COMMODORE 64 WHOLE MEMORY GUIDE

Tim Arnot

COMMODORE 64 WHOLE MEMORY GUIDE

Tim Arnot

MELBOURNE HOUSE
PUBLISHERS

©1985 Tim Arnot

All rights reserved. This book is copyright and no part may be copied or stored by electromagnetic, electronic, photographic, mechanical or any other means whatsoever except as provided by national law. All enquiries should be addressed to the publishers:

IN THE UNITED KINGDOM —
Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

IN AUSTRALIA —
Melbourne House (Australia) Pty Ltd
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

Melbourne House acknowledges the help of Commodore Business Machines (UK) Ltd, for their permission to reproduce disassembled code from the CBM 64. All of the work in this book belongs to Tim Arnot and Commodore Business Machines is in no way responsible for its accuracy.

The ROM contents are © Commodore Business Machines.

ISBN 0 86161 194 2

Printed by The Whitefriars Press Ltd, Tonbridge, Kent

Edition: 7 6 5 4 3 2 1
Printing: F E D C B A 9 8 7 6 5 4 3 2 1
Year: 90 89 88 87 86 85

CONTENTS

INTRODUCTION 1

SECTION 1 Memory Maps 3
RAM Memory Map 4
BASIC ROM Memory Map 12
Kernal ROM Memory Map 16
Input/Output Memory Map 21

SECTION 2 RAM Guide 28

SECTION 3 ROM Guide 78

SECTION 4 Kernal Guide 318

SECTION 5 Input/Output Guide 341

APPENDIX 1
The version 3 Kernal ROM 395

APPENDIX 2
HEX/Decimal Converter 397

APPENDIX 3
CBM ASCII codes 398

INDEX 400

INTRODUCTION

This is not just a memory map of the Commodore 64 micro computer, it's a memory guide. So, what's the difference? A memory map is a list of all the memory locations used within a computer. A memory guide is much more than this. It's a detailed description of each location, explaining what it's for, how it is used by the computer, and, more importantly, how it can be used by the programmer.

The memory guide to the Commodore 64 is split up into three main sections; the RAM guide, the I/O guide, and the ROM guide. The ROM guide also includes a complete and annotated disassembly of the Commodore 64 ROMs.

If you are a machine code programmer, this book will be invaluable in helping you to write programs that incorporate the subroutines contained within the Commodore 64 ROM. It explains how to pass any parameters that may be required to the routines, and how to recover the results of calculations etc. that are returned by these routines. Also covered is the procedure used by the computer to cope with errors in the data operated on by the routines.

If you are a BASIC programmer, this book will enable you to manipulate the system variables used by the Commodore 64 to your own ends. This will allow you to use the advanced features of the Commodore 64 to their full.

This book does not, however, teach you how to program in machine code, or give explanations of the 6502/6510 instruction set. I have assumed that you possess, or have access to some other book which can do this for you.

NUMBER FORMATS

In general, the hexadecimal (base 16) number system has been used throughout this book. Decimal numbers are sometimes used, though, and to distinguish these from hex numbers, and also to distinguish data from pointers, vectors etc., a system of prefixes has been used.

If the number has no prefix, then it is a decimal number.

If the number is hex data, then it is prefixed with a hash (#). This is mainly to conform to the 6502 immediate addressing mode format, where data is prefixed in this way. Examples are #60, #1F, #CA.

If the number is a hex address, then it is prefixed with a dollar sign ($). Examples are $FF, $A000, $B506.

If the number is a pointer or vector, then it is enclosed in parentheses (). For example ($C1) represents a pointer or vector that is held in locations $C1 and $C2.

Many locations are referred to by a label. A label is simply a mnemonic device to aid the programmer in remembering which locations perform which functions. The labels used in this book are those widely accepted as 'Commodore standard'.

SECTION 1.
MEMORY MAPS

RAM memory map

LABEL		ADDRESS	COMMENTS
D6510	$00	0	6510 onboard Data Direction Register
R6510	$01	1	6510 onboard I/O port
	$02	2	Unused
ADRAY1	$03-$04	3-4	Jump vector: convert flpt to integer
ADRAY2	$05-$06	5-6	Jump vector: convert integer to flpt
CHARAC	$07	7	Search character
ENDCHR	$08	8	Flag: scan for quote at end of string
TRMPOS	$09	9	Screen column before last TAB
VERCK	$0A	10	Flag: 0 = LOAD, 1 = VERIFY
COUNT	$0B	11	Input buffer pointer, No. of subscripts
DIMFLG	$0C	12	Flag: default array DIM
VALTYP	$0D	13	Data type: #FF = string, #00 = numeric
INTFLG	$0E	14	Data type: #80 = integer, #00 = flpt
GARBFL	$0F	15	Flag: DATA scan/LIST quote/garbage collect
SUBFLG	$10	16	Flag: subscript ref/ user function call
INPFLG	$11	17	Flag: #00 = INPUT, #40 = GET, #98 = READ
TANSGN	$12	18	Flag: TAN sign / comparison result
CHANNL	$13	19	Flag: Current I/O channel
LINNUM	$14-$15	20-21	Integer value
TEMPPT	$16	22	Pointer: temporary string stack
LASTPT	$17-$18	24-24	Last temporary string address
TEMPST	$19-$21	25-33	Stack for temporary strings
INDEX	$22-$25	34-37	Utility pointer area
RESHO	$26-02A	38-42	Temporary fac for product of multiply
TXTTAB	$2B-$2C	43-44	Pointer: start of BASIC text
VARTAB	$2D-$2E	45-46	Pointer: start of BASIC variables
ARYTAB	$2F-$30	47-48	Pointer: start of BASIC arrays
STREND	$31-$32	49-50	Pointer: end of BASIC arrays +1
FRETOP	$33-$34	51-52	Pointer: bottom of string

			storage
FRESPC	$35-$36	53-54	Utility string pointer
MEMSIZ	$37-$38	55-56	Pointer: highest address used by BASIC
CURLIN	$39-$3A	57-58	Current BASIC line number
OLDLIN	$3B-$3C	59-60	Previous BASIC line number
OLDTXT	$3D-$3E	61-62	Pointer: BASIC statement for CONT
DATLIN	$3F-040	63-64	Current DATA line number
DATPTR	$41-$42	65-66	Pointer: current data item address
INPPTR	$43-$44	67-68	Vector: INPUT routine
VARNAM	$45-$46	69-70	Current BASIC variable name
VARPNT	$47-$48	71-72	Pointer: start of current variable data
FORPNT	$49-$4A	73-74	Pointer: index variable for FOR/NEXT
OPPTR	$4B-$4C	75-76	Operator table displacement
OPMASK	$4D	77	Mask for comparison
DEFPNT	$4E-$4F	78-79	Pointer: Current FN descriptor
DSCPNT	$50-$52	80-82	Pointer: Current string descriptor
FOUR6	$53	83	Constant for garbage collection
JMPER	$54-$56	84-86	6510 JMP to function
	$57-$60	87-96	Temp data area
FACEXP	$61	97	Fac#1: exponent
FACHO	$62-$65	98-101	Fac#1: mantissa
FACSGN	$66	102	Fac#1: sign
SGNFLG	$67	103	Pointer: constant for series evaluation
BITS	$68	104	Fac#1: overflow digit
ARGEXP	$69	105	Fac#2: exponent
ARGHO	$6A-$6D	106-109	Fac#2: mantissa
ARGSGN	$6E	110	Fac#2: sign
ARISGN	$6F	111	Fac#1 v fac#2 sign comparison result
FACOV	$70	112	Fac#1: low order (rounding)
FBUFPT	$71-$72	113-114	Pointer: cassette buffer
CHRGET	$73-$8A	115-138	Subroutine: get next byte of BASIC text
CHRGOT	$79	121	Entry to get current byte of BASIC text
TXTPTR	$7A-$7B	122-123	Pointer: current byte of BASIC text
RNDX	$8B-$8F	139-143	Seed value for RND
STATUS	$90	144	Kernal I/O status word (ST)
STKEY	$91	145	Flag: <STOP> key pressed
SVXT	$92	146	Tape timing constant

VERCK	$93	147	KERNAL flag: #00 = LOAD, #01 = VERIFY
C3PO	$94	148	Flag: buffered character for serial bus
BSOUR	$95	149	Buffered character for serial bus
SYNO	$96	150	Cassette sync no.
XSAV	$97	151	Temp data area
LDTND	$98	152	No. of open files, index to file table
DFLTN	$99	153	Default input device (0)
DFLTO	$9A	154	Default output (CMD) device (3)
PRTY	$9B	155	Tape character parity
DPSW	$9C	156	Flag: tape byte received
MSGFLG	$9D	157	Flag: Control KERNAL message output
PTR1	$9E	158	Tape pass 1 error log
PTR2	$9F	159	Tape pass 2 error log
TIME	$A0-$A2	160-162	Real-time jiffy clock (approx) 1/60 sec.
	$A3	163	Serial bit count / EOI flag
	$A4	164	Cycle count
CNTDN	$A5	165	Cassette sync countdown
BUFPNT	$A6	166	Pointer: tape I/O buffer
INBIT	$A7	167	RS-232 in bits, tape leader short count
BITC1	$A8	168	RS-232 in bit count, tape write new byte, read error
RINONE	$A9	169	RS-232 check for start bit, tape count zeros
RIDATA	$AA	170	RS-232 input byte buffer, tape function mode (bits 6 & 7), sync countdown
RIPRTY	$AB	171	RS-232 input parity, tape short count
SAL	$AC-$AD	172-173	Pointer: tape buffer, screen scrolling
EAL	$AE-$AF	174-175	Tape end address, end of program
CMPO	$B0-$B1	176-177	Tape timing constants
TAPE1	$B2-$B3	178-179	Pointer: start of tape buffer
BITTS	$B4	180	RS-232 out bit count, #1 = tape timer enabled
NXTBIT	$B5	181	RS-232 next bit to send, tape EOT flag
RODATA	$B6	182	RS-232 out byte buffer, tape read character error
FNLEN	$B7	183	Length of current file name

LA	$B8	184	Current logical file number
SA	$B9	185	Current secondary address
FA	$BA	186	Current device number
FNADR	$BB-$BC	187-188	Pointer to current file name
ROPRTY	$BD	189	RS-232 out parity, tape receive I/P character
FSBLK	$BE	190	Number of blocks to read/write to tape
MYCH	$BF	191	Serial word buffer
CAS1	$C0	192	Tape motor interlock
STAL	$C1-$C2	193-194	I/O start address
MEMUSS	$C3-$C4	195-196	Data start address
LSTX	$C5	197	Current key pressed
NDX	$C6	198	No. of characters in keyboard buffer
RVS	$C7	199	Flag: print reverse characters. #00 = no
INDX	$C8	200	Pointer: end of logical line for INPUT
LXSP	$C9-$CA	201-202	Cursor X-Y position at start of INPUT
SFDX	$CB	203	Keyboard matrix coordinate
BLNSW	$CC	204	Cursor blink enable: #00 = flash cursor
BLNCT	$CD	205	Timer: countdown to toggle cursor
GDBLN	$CE	206	Character under cursor
BLNON	$CF	207	Flag: cursor blink phase
CRSW	$D0	208	Flag: INPUT / GET from keyboard
PNT	$D1-$D2	209-210	Pointer: current screen line address
PNTR	$D3	211	Current cursor column on line
QTSW	$D4	212	Flag: editor in quotes mode, #00 = no
LNMX	$D5	213	Screen line length
TBLX	$D6	214	Current cursor line number
	$D7	215	Tape: most recent dipole bit value
INSRT	$D8	216	Flag: INST mode. >#00 = # inserts
LDTB1	$D9-$F2	217-242	Screen line link table
USER	$F3-$F4	243-244	Pointer: current screen colour location
KEYTAB	$F5-$F6	245-246	Vector: keyboard decode table
RIBUF	$F7-$F8	247-248	RS-232 in buffer pointer
ROBUF	$F9-$FA	249-250	RS-232 out buffer pointer
FREKZP	$FB-$FE	251-254	Free 0-page space for user programs

BASZPT	$FF	255	Temp data area

PAGE 1

	$100-$1FF	256-511	Microprocessor system stack space
	$100-$10A	256-266	Flpt to ASCII string work area
BAD	$100-$13E	256-318	Tape input error log

PAGE 2

BUF	$200-$258	512-600	BASIC input buffer
LAT	$259-$262	601-610	Table of active file numbers
FAT	$263-$26C	611-620	Table of device numbers for open files
SAT	$26D-$276	621-630	Table of secondary addresses for open files
KEYD	$277-$280	631-640	Keyboard buffer queue (FIFO)
MEMSTR	$281-$282	641-642	Pointer: bottom of BASIC memory
MEMSIZ	$283-$284	643-644	Pointer: top of BASIC memory
TIMOUT	$285	645	IEEE timeout flag
COLOR	$286	646	Current character colour code
GDCOL	$287	647	Character colour under cursor
HIBASE	$288	648	Top of screen memory (page)
XMAX	$289	649	Size of keyboard buffer
RPTFLG	$28A	650	Flag: repeat keypress, #80 = all keys
KOUNT	$28B	651	Repeat speed counter
DELAY	$28C	652	Repeat delay counter
SHFLG	$28D	653	Flag <SHIFT>/<CTRL>/<CBM> key
LSTSHF	$28E	654	Last keyboard shift pattern
KEYLOG	$28F-$290	655-656	Vector: keyboard table setup
MODE	$291	657	#00 = disable <SHIFT>/<CBM>, #80 = enable
AUTODN	$292	658	#00 = auto scroll down on
M51CTR	$293	659	RS-232 6551 control register image
M51CDR	$294	660	RS-232 6551 command register image
M51AJB	$295-$296	661-662	RS-232 non-standard bit time
RSSTAT	$297	663	RS-232 6551 status register image
BITNUM	$298	664	RS-232 number of bits still to send
BAUDOF	$299-$29A	665-666	RS-232 baud rate (bit time)
RIDBE	$29B	667	RS-232 index to end of in buffer
RIDBS	$29C	668	RS-232 start page of in buffer

RODBS	$29D	669	RS-232 start page of out buffer
RODBE	$29E	670	RS-232 index to end of out buffer
IRQTMP	$29F-$2A0	671-672	IRQ vector store during tape I/O
ENABL	$2A1	673	RS-232 enables (NMI interrupt control)
	$2A2	674	TOD sense during tape I/O
	$2A3	675	Temp store for tape read
	$2A4	676	Temp D1IRQ indicator for tape read
	$2A5	677	Temp screen line index
	$2A6	678	#00 = NTSC, #01 = PAL
SPRT11	$2A7-$2FF	679-767	Unused (sprite block 11)

PAGE 3

IERROR	$300-$301	768-769	Vector: print BASIC error message ($E38B)
IMAIN	$302-$303	770-771	Vector: BASIC warm start ($A483)
ICRNCH	$304-$305	772-773	Vector: tokenise BASIC text ($A57C)
IQPLOP	$306-$307	774-775	Vector: list BASIC text ($A71A)
IGONE	$308-$309	776-777	Vector: BASIC character dispatch ($A7E4)
IEVAL	$30A-$30B	778-779	Vector: evaluate BASIC token ($AE86)
SAREG	$30C	780	(A) register for SYS
SXREG	$30D	781	(X) register for SYS
SYREG	$30E	782	(Y) register for SYS
SPREG	$30F	783	(P) register for SYS
USRPOK	$310	784	JMP (#4C) instruction for USR function
USRADD	$311-$312	785-786	Lo-hi address for USR user code
	$313	787	Unused
CINV	$314-$315	788-789	Vector: hardware IRQ interrupt ($EA31)
CBINV	$316-$317	790-791	Vector: software BRK interrupt ($FE66)
NMINV	$318-$319	792-793	Vector: hardware NMI interrupt ($FE47)
IOPEN	$31A-$31B	794-795	Vector: KERNAL OPEN routine ($F34A)
ICLOSE	$31C-$31D	796-797	Vector: KERNAL CLOSE routine ($F291)

ICHKIN	$31E-$31F	798-799	Vector: KERNAL CHKIN routine ($F20E)
ICKOUT	$320-$321	800-801	Vector: KERNAL CHKOUT routine ($F250)
ICLRCH	$322-$323	802-803	Vector: KERNAL CLRCHN routine ($F333)
IBASIN	$324-$325	804-805	Vector: KERNAL CHRIN routine ($F157)
IBSOUT	$326-$327	806-807	Vector: KERNAL CHROUT routine ($F1CA)
ISTOP	$328-$329	808-809	Vector: KERNAL STOP routine ($F6ED)
IGETIN	$32A-$32B	810-811	Vector: KERNAL GETIN routine ($F13E)
ICLALL	$32C-$32D	821-813	Vector: KERNAL CLALL routine ($F32F)
USRCMD	$32E-$32F	814-815	User-defined vector ($FE66)
ILOAD	$330-$331	816-817	Vector: KERNAL LOAD routine ($F49E)
ISAVE	$332-$333	818-819	Vector: KERNAL SAVE routine ($F5DD)
	$334-$33B	820-827	Unused
TBUFFR	$33C-$3FB	828-1019	Cassette I/O buffer
SPRT13	$340-$37E	832-894	Unused (sprite block 13)
	$37F	895	Unused
SPRT14	$380-$3BE	896-958	Unused (sprite block 14)
	$3BF	959	Unused
SPRT15	$3C0-$3FE	960-1022	Unused (sprite block 15)
	$3FF	1023	Unused

PAGE 4 UPWARDS

VICSCN	$400-$7E7	1024-2023	Video matrix: 25 lines by 40 columns
	$7F8-$7FF	2040-2047	Sprite data pointers
	$800-$9FFF	2048-40959	BASIC RAM program area
	$1000-$1FFF	4096-8191	Bank 0 character ROM image
	$8000-$9FFF	32768-40959	External 8K plug-in ROM area
	$9000-$9FFF	36864-40959	Bank 2 character ROM image
	$A000-$BFFF	40960-49151	BASIC ROM (or 8k RAM)
	$C000-$CFFF	49152-53247	4k RAM
	$D000-$DFFF	53248-57343	Character ROM / 4k RAM
	$D000-$D02E	53248-53294	

Video Interface Controller (VIC)
$D400-$D41C 54272-54300
Sound Interface Device (SID)
$D800-$DBFF 55296-56319
Colour RAM (nybbles)
$DC00-$DC0F 56320-56335
Complex Interface Adaptor #1
$DD00-$DD0F 56576-56591
Complex Interface Adaptor #2
$E000-$FFFF 57344-65535
Kernal ROM (or 8k RAM)

BASIC interpreter ROM ($A000 — $BFFF)

$A000	40960		Restart vectors
$A00C	40972	STMDSP	BASIC command vectors
$A052	41042	FUNDSP	BASIC function vectors
$A080	41088	OPTAB	BASIC operator vectors
$A09E	41118	RESLST	BASIC command keyword table
$A129	41257	MSCLST	BASIC misc. keyword table
$A140	41280	OPLIST	BASIC operator keyword table
$A14D	41293	FUNLST	BASIC function keyword table
$A19E	41374	ERRTAB	Error message table
$A328	41768	ERRPTR	Error message pointers
$A364	41828	OKK	Misc. messages
$A38A	41866	FNDFOR	Find FOR/GOSUB entry on stack
$A3B8	41912	BLTU	Open space in memory
$A3FB	41979	GETSTK	Check stack depth
$A408	41992	REASON	Check memory overlap
$A435	42037	OMERR	Output ?OUT OF MEMORY error
$A437	42039	ERROR	Error routine
$A469	42089	ERRFIN	Break entry
$A474	42100	READY	Restart BASIC
$A480	42112	MAIN	Input & identify BASIC line
$A49C	42140	MAIN1	Get line number & tokenise text
$A4A2	42146	INSLIN	Insert BASIC text
$A533	42291	LINKPRG	Rechain lines
$A560	42336	INLIN	Input line into buffer
$A579	42361	CRUNCH	Tokenise input buffer
$A613	42515	FNDLIN	Search for line number
$A642	42562	SCRTCH	Perform NEW
$A65E	42590	CLEAR	Perform CLR
$A68E	42638	STXPT	Reset TXTPTR
$A69C	42652	LIST	Perform LIST
$A717	42775	QPLOP	Handle LIST character
$A742	42818	FOR	Perform FOR
$A7AE	42926	NEWSTT	BASIC warm start
$A7C4	42948	CKEOL	Check end of program
$A7E1	42977	GONE	Prepare to execute statement
$A7ED	42989	GONE3	Perform BASIC keyword
$A81D	43037	RESTOR	Perform RESTORE
$A82C	43052	STOP	Perform STOP, END, BREAK
$A857	43095	CONT	Perform CONT
$A871	43121	RUN	Perform RUN
$A883	43139	GOSUB	Perform GOSUB
$A8A0	43168	GOTO	Perform GOTO
$A8D2	43218	RETURN	Perform RETURN
$A8F8	43256	DATA	Perform DATA
$A906	43270	DATAN	Search for next statement / line

$A928	43304	IF	Perform IF
$A93B	43323	REM	Perform REM
$A94B	43339	ONGOTO	Perform ON
$A96B	43371	LINGET	Fetch LINNUM from BASIC
$A9A5	43429	LET	Perform LET
$A9C4	43460	PUTINT	Assign integer
$A9D6	43478	PTFLPT	Assign floating point
$A9D9	43481	PUTSTR	Assign string
$A9E3	43491	PUTTIM	Assign TI$
$AA2C	43564	GETSPT	Add digit to fac#1
$AA80	43648	PRINTN	Perform PRINT#
$AA86	43654	CMD	Perform CMD
$AA9A	43674	STRDON	Print string from memory
$AAA0	43680	PRINT	Perform PRINT
$AAB8	43704	VAROP	Output variable
$AAD7	43735	CRDO	Output CR/LF
$AAE8	43752	COMPRT	Handle comma, TAB(, SPC(
$AB1E	43806	STROUT	Output string
$AB3B	43835	OUTSPC	Output format character
$AB4D	43853	DOAGIN	Handle bad data
$AB7B	43899	GET	Perform GET
$ABA5	43941	INPUTN	Perform INPUT#
$ABBF	43967	INPUT	Perform INPUT
$ABEA	44010	BUFFUL	Read input buffer
$ABF9	44025	QINLIN	Do input prompt
$AC06	44038	READ	Perform READ
$AC35	44085	RDGET	General purpose read routine
$ACFC	44284	EXINT	Input error messages
$AD1E	44318	NEXT	Perform NEXT
$AD61	44385	DONEXT	Check valid loop
$AD8A	44426	FRMNUM	Confirm result
$AD9E	44446	FRMEVL	Evaluate expression in text
$AE83	44675	EVAL	Evaluate single term
$AEA8	44712	PIVAL	Constant - pi
$AEAD	44717	QDOT	Continue expression
$AEF1	44785	PARCHK	Expression in brackets
$AEF7	44791	CHKCLS	Confirm character
$AF08	44808	SYNERR	Output ?SYNTAX error
$AF0D	44813	DOMIN	Set up NOT function
$AF14	44820	RSVVAR	Identify reserved variable
$AF28	44840	ISVAR	Search for variable
$AF48	44872	TISASC	Convert TI to ascii string
$AFA7	44967	ISFUN	Identify function type
$AFB1	44977	STRFUN	Evaluate string function
$AFD1	45009	NUMFUN	Evaluate numeric function
$AFE6	45030	OROP	Perform OR, AND
$B016	45078	DOREL	Perform <, =, >
$B01B	45083	NUMREL	Numeric comparison
$B02E	45102	STRREL	String comparison

$B07E	45182	DIM	Perform DIM
$B08B	45195	PTRGET	Identify variable
$B0E7	45286	ORDVAR	Locate ordinary variable
$B113	45331	ISLETC	Does (A) hold an alphabetic character?
$B11D	45341	NOTFNS	Create new variable
$B128	45352	NOTEVL	Create variable
$B194	45460	ARYGET	Allocate array pointer space
$B1A5	45477	N32768	Constant 32768 in flpt
$B1AA	45482	FACINX	Fac#1 to integer in (A/Y)
$B1B2	45490	INTIDX	Evaluate text for integer
$B1BF	45503	AYINT	Fac#1 to positive integer
$B1D1	45521	ISARY	Get array parameters
$B218	45592	FNDARY	Find array
$B245	45637	BSERR	?BAD SUBSCRIPT/?ILLEGAL QUANTITY
$B261	45665	NOTFDD	Create array
$B30E	45838	INLPN2	Locate element in array
$B34C	45900	UMULT	Number of bytes in subscript
$B37D	45949	FRE	Perform FRE
$B391	45969	GIVAYF	Convert integer in (A/Y) to flpt
$B39E	45982	POS	Perform POS
$B3A6	45990	ERRDIR	Confirm program mode
$B3B3	46003	DEF	Perform DEF
$B3E1	46049	GETFNM	Check syntax of FN
$B3F4	46068	FNDOER	Perform FN
$B465	46181	STRD	Perform STR$
$B487	46215	STRLIT	Set up string
$B4D5	46293	PUTNW1	Save string descriptor
$B4F4	46324	GETSPA	Allocate space for string
$B526	46374	GARBAG	Garbage collection
$B5BD	46525	DVARS	Search for next string
$B606	46598	GRBPAS	Collect a string
$B63D	46653	CAT	Concatenate two strings
$B67A	46714	MOVINS	Store string in high RAM
$B6A3	46755	FRESTR	Perform string househeeping
$B6DB	46811	FREFAC	Clean descriptor stack
$B6EC	46828	CHRD	Perform CHR$
$B700	46848	LEFTD	Perform LEFT$
$B72C	46892	RIGHTD	Perform RIGHT$
$B737	46903	MIDD	Perform MID$
$B761	46945	PREAM	Pull string parameters
$B77C	46972	LEN	Perform LEN
$B782	46978	LEN1	Exit string mode
$B78B	46987	ASC	Perform ASC
$B79B	47003	GTBYTC	Evaluate text to 1 byte in (X)
$B7AD	47021	VAL	Perform VAL
$B7B5	47029	STRVAL	Convert ascii string to flpt
$B7EB	47083	GETNUM	Get parameters for POKE, WAIT
$B7F7	47095	GETADR	Convert fac#1 to integer in LINNUM
$B80D	47117	PEEK	Perform PEEK

$B824	47140	POKE	Perform POKE
$B82D	47149	WAIT	Perform WAIT
$B849	47177	FADDH	Add 0.5 to fac#1
$B850	47184	FSUB	Perform subtraction
$B862	47202	FADD5	Normalise addition
$B867	47207	FADD	Perform addition
$B947	47431	NEGFAC	2's complement fac#1
$B97E	47486	OVERR	Output ?OVERFLOW error
$B983	47491	MULSHF	Multiply by zero byte
$B9BC	47548	FONE	Table of flpt constants
$B9EA	47594	LOG	Perform LOG
$BA28	47656	FMULT	Perform multiply
$BA59	47705	MULPLY	Multiply by a byte
$BA8C	47756	CONUPK	Load fac#2 from memory
$BAB7	47799	MULDIV	Test both accumulators
$BAD4	47828	MLDVEX	Overflow/underflow
$BAE2	47842	MUL10	Multiply fac#1 by 10
$BAF9	47865	TENC	Constant 10 in flpt
$BAFE	47870	DIV10	Divide fac#1 by 10
$BB07	47879	FDIV	Divide fac#2 by flpt at (A/Y)
$BB0F	47887	FDIVT	Divide fac#2 by fac#1
$BBA2	48034	MOVFM	Load fac#1 from memory
$BBC7	48071	MOV2F	Store fac#1 in memory
$BBFC	48124	MOVFA	Copy fac#2 into fac#1
$BC0C	48140	MOVAF	Copy fac#1 into fac#2
$BC1B	48155	ROUND	Round fac#1
$BC2B	48171	SIGN	Check sign of fac#1
$BC39	48185	SGN	Perform SGN
$BC58	48216	ABS	Perform ABS
$BC5B	48219	FCOMP	Compare fac#1 with memory
$BC9B	48283	QINT	Convert fac#1 to integer
$BCCC	48332	INT	Perform INT
$BCF3	48371	FIN	Convert ASCII string to a number in fac#1
$BDB3	48563	N0999	String conversion constants
$BDC2	48578	INPRT	Output 'IN' and line number
$BDDD	48605	FOUT	Convert fac#1 to ASCII string
$BE68	48744	FOUTIM	Convert TI to string
$BF11	48913	FHALF	Table of constants
$BF71	49009	SQR	Perform SQR
$BF7B	49019	FPWRT	Perform power ^
$BFB4	49076	NEGOP	Negate fac#1
$BFBF	49087	LOGEB2	Table of constants
$BFED	49133	EXP	Perform EXP

KERNAL ROM ($E000 — $FFFF)

$E000	57344		EXP continued from BASIC ROM
$E043	57411	POLYX	Series evaliation
$E08D	57485	RMULC	Constants for RND
$E097	57495	RND	Perform RND
$E0F9	57593	BIOERR	Handle I/O error in BASIC
$E10C	57612	BCHOUT	Output character
$E112	57618	BCHIN	Input character
$E118	57624	BCKOUT	Set up for output
$E11E	57630	BCKIN	Set up for input
$E124	57636	BGETIN	Get one character
$E12A	57642	SYS	Perform SYS
$E156	57686	SAVET	Perform SAVE
$E165	57701	VERFYT	Perform VERIFY/LOAD
$E1BE	57790	OPENT	Perform OPEN
$E1C7	57799	CLOSET	Perform CLOSE
$E1D4	57812	SLPARA	Get parameters for LOAD/SAVE
$E200	57856	COMBYT	Get next one byte parameter
$E206	57862	DEFLT	Check default parameters
$E20E	57870	CMMERR	Check for comma
$E219	57881	OCPARA	Get parameters for OPEN/CLOSE
$E264	57956	COS	Perform COS
$E26B	57963	SIN	Perform SIN
$E2B4	58036	TAN	Perform TAN
$E2E0	58080	PI2	Table of trig constants
$E30E	58126	ATN	Perform ATN
$E33E	58174	ATNCON	Table of ATN constants
$E37B	58235	BASSFT	BASIC warm restart
$E394	58260	INIT	BASIC cold restart
$E3A2	58274	INITAT	CHRGET for zero-page
$E3BA	58298	RNDSED	RND seed for zero-page
$E3BF	58303	INITCZ	Initialise BASIC RAM
$E422	58402	INITMS	Output power-up message
$E447	58439	BVTRS	Table of BASIC vectors
$E453	58451	INITV	Initialise vectors
$E45F	58463	WORDS	Power-up message
$E4AD	58541		Patch for BASIC call to CHKOUT
$E4B7	58551		Unused bytes for future patches
$E4DA	58586		Reset character colour
$E4E0	58592		Pause after finding tape file
$E4EC	58604		RS-232 timing table - PAL
$E500	58624	IOBASE	Get I/O address
$E505	58629	SCREEN	Get screen size
$E50A	58634	PLOT	Put/get row and column
$E518	58648	CINT1	Initialise I/O
$E544	58692		Clear screen
$E566	58726		Home cursor
$E56C	58732		Set screen pointers

$E59A	58778		Set I/O defaults
$E5B4	58804	LP2	Get character from keyboard buffer
$E5CA	58826		Input from keyboard
$E632	58930		Input from screen or keyboard
$E684	59012		Quotes test
$E691	59025		Set up screen print
$E6B6	59062		Advance cursor
$E6ED	59117		Retreat cursor
$E701	59137		Back on to previous line
$E716	59158		Output to screen
$E87C	59516		Go to next line
$E891	59537		Output <carriage return>
$E8A1	59553		Check line decrement
$EACB	59595		Set colour code
$E8DA	59610		Colour code table
$E8EA	59626		Scroll screen
$E965	59749		Open a space on the screen
$E9C8	59848		Move a screen line
$E9E0	59872		Synchronise colour transfer
$E9F0	59888		Set start of line
$E9FF	59903		Clear screen line
$EA13	59923		Print to screen
$EA24	59940		Synchronise colour pointer
$EA31	59953		Main IRQ entry point
$EA87	60039	SCNKEY	Scan keyboard
$EADD	60125		Process key image
$EB79	60281		Keyboard select vectors
$EB81	60289		Keyboard 1 - unshifted
$EBC2	60354		Keyboard 2 - shifted
$EC03	60419		Keyboard 3 - commodore
$EC44	60484		Graphics / text control
$EC78	60536		Keyboard 4 - control
$ECB9	60601		Video chip setup table
$ECE7	60647		Shift-run equivalent
$ECF0	60656		Low byte screen line addresses
$ED09	60681	TALK	Send TALK / LISTEN
$ED40	60736		Send data on serial bus
$EDAD	60845		Flag errors
$EDB9	60857	SECOND	Send LISTEN SA
$EDBE	60862		Clear ATN
$EDC7	60871	TKSA	Send TALK SA
$EDCC	60876		Wait for clock
$EDDD	60893	CIOUT	Send serial deferred
$EDEF	60911	UNTLK	Send UNTALK / UNLISTEN
$EE13	60947	ACPTR	Receive from serial bus
$EE85	61061		Serial clock on
$EE8E	61070		Serial clock off
$EE97	61079		Serial output 1
$EEA0	61088		Serial output 0

$EEA9	61097		Get serial data and clock in
$EEB3	61107		Delay 1 mS
$EEBB	61115		RS-232 send
$EF06	61190		Send new RS-232 byte
$EF2E	61230		No DSR / CTS error
$EF39	61241		Disable timer
$EF4A	61258		Compute bit count
$EF59	61273		RS-232 receive
$EF7E	61310		Set up to receive
$EF90	61328		Process RS-232 byte
$EFE1	61409		Submit to RS-232
$F00D	61453		No DSR error
$F017	61463		Send to RS-232 buffer
$F04D	61517		Input from RS-232
$F086	61574		Get from RS-232
$F0A4	61604		Serial bus idle
$F0BD	61629		Table of kernal I/O messages
$F12B	61739		Print message if direct
$F13E	61758	GETIN	Get a byte...
$F157	61783	CHRIN	Input a byte...
$F199	61849		Get from tape/serial/RS-232
$F1CA	61898	CHROUT	Output one character
$F20E	61966	CHKIN	Set input device
$F250	62032	CHKOUT	Set output device
$F291	62097	CLOSE	Close file
$F30F	62223		Find file
$F31F	62239		Set file values
$F32F	62255	CLALL	Abort all files
$F333	62259	CLRCHN	Restore default I/O
$F34A	62282	OPEN	Open file
$F3D5	62421		Send SA
$F409	62473		Open RS-232
$F49E	62622	LOAD	Load RAM
$F4B8	62648		Load from serial bus
$F533	62771		Load from tape
$F5AF	62927		Print "SEARCHING"
$F5C1	62913		Print filename
$F5D2	62930		Print "LOADING/VERIFYING"
$F5DD	62941	SAVE	Save RAM
$F5FA	62970		Save to serial bus
$F659	63065		Save to tape
$F68F	63119		Print "SAVING"
$F69B	63131	UDTIM	Bump clock
$F6BC	63164		Log CIA key reading
$F6DD	63197	RDTIM	Get time
$F6E4	63204	SETTIM	Set time
$F6ED	63213	STOP	Check stop key
$F6FB	63227		Output error messages
$F72D	63277		Find any tape header

$F76A	63338		Write tape header
$F7D0	63440		Get buffer address
$F7D7	63447		Set buffer start/end pointers
$F7EA	63466		Find specific tape header
$F80D	63501		Bump tape pointer
$F817	63511		Print "PRESS PLAY"
$F82E	63534		Check tape status
$F838	63544		Print "PRESS RECORD"
$F841	63553		Initiate tape read
$F864	63588		Initiate tape write
$F875	63605		Common tape code
$F8D0	63696		Check tape stop
$F8E2	63714		Set read timing
$F92C	63788		Read tape bits
$FA60	64096		Store tape characters
$FB8E	64398		Reset tape pointer
$FB97	64407		New character setup
$FBA6	64422		Send tone to tape
$FBC8	64456		Write data to tape
$FBCD	64461		IRQ entry point
$FC57	64599		Write tape leader
$FC93	64659		Restore normal IRQ
$FCB8	64696		Set IRQ vector
$FCCA	64714		Kill tape motor
$FCD1	64721		Check read/write pointer
$FCDB	64731		Bump read/write pointer
$FCE2	64738		Power reset entry
$FD02	64770		Check for 8-ROM
$FD12	64786		8-ROM mask
$FD15	64789	RESTOR	Kernal reset
$FD1A	64794	VECTOR	Kernal move
$FD30	64816		Kernal reset vectors
$FD50	64848	RAMTAS	Initialise system constants
$FD9B	64923		IRQ vectors for tape I/O
$FDA3	64931	IOINIT	Initialise I/O
$FDDD	64989		Enable timer
$FDF9	65017	SETNAM	Save filename data
$FE00	65024	SETLFS	Save file details
$FE07	65031	READST	Get STATUS
$FE18	65048	SETMSG	Flag STATUS
$FE21	65057	SETTMO	Set IEEE timeout
$FE25	65061	MEMTOP	Read/set top of memory
$FE34	65076	MEMBOT	Read/set bottom of memory
$FE43	65091		NMI entry point
$FE66	65126		Warm start BASIC
$FEC2	65218		RS-232 timing table - NSTC
$FED6	65238		NMI RS-232 in
$FF07	65287		NMI RS-232 out
$FF43	65347		Fake IRQ entry

$FF48	65352		IRQ entry
$FF5B	65371	CINT	Initialise screen editor
$FF80	65408		KERNAL version I.D.
$FF81	65409		Kernal jump table
$FFFA	65530		System hardware vectors

I/O memory map

```
$0000  0      D6510   Onboard data direction register
                      (xx101111).
$0001  1      R6510   Onboard I/O port. Bits are assigned as
                      follows:
                        0 LORAM signal (0 = switch BASIC ROM
                      out)
                        1 HIRAM signal (0 = switch kernal
                      ROM out)
                        2 CHAREN signal (0 = switch
                      character ROM out)
                        3 Cassette data out line
                        4 Cassette switch sense (1 = switch
                      pressed)
                        5 Cassette motor control (1 = motor
                      ON)
                      6-7 UNDEFINED
```

<u>6566/6567 VIDEO INTERFACE CONTROLLER (VIC II)</u>

```
$D000  53248  Sprite #0, X position
$D001  53249  Sprite #0, Y position
$D002  53250  Sprite #1, X position
$D003  53251  Sprite #1, Y position
$D004  53252  Sprite #2, X position
$D005  53253  Sprite #2, Y position
$D006  53254  Sprite #3, X position
$D007  53255  Sprite #3, Y position
$D008  53256  Sprite #4, X position
$D009  53257  Sprite #4, Y position
$D00A  53258  Sprite #5, X position
$D00B  53259  Sprite #5, Y position
$D00C  53260  Sprite #6, X position
$D00D  53261  Sprite #6, Y position
$D00E  63262  Sprite #7, X position
$D00F  53263  Sprite #7, Y position
$D010  53264  Sprites 0-7, MSB of X position
$D011  53265  VIC control register
                  7 Raster compare (MSB of $D012)
                  6 Extended colour text (1 = ON)
                  5 Bit map mode (1 = ON)
                  4 Blank video display (0 = BLANK)
                  3 Row select (1 = 25, 0 = 24)
                2-0 Smooth scroll vertical bit position
$D012  53266  Raster Read/Write
$D013  53267  Light pen X (latch)
$D014  53268  Light pen Y (latch)
```

```
$D015  53269  Sprites 0-7, enable (1 = ENABLE)
$D016  53270  VIC control register
                  5 Reset VIC (1 = RESET)
                  4 Multicolour mode (1 = ON)
                  3 Column select (1 = 40, 0 = 38)
                2-0 Smooth scroll horizontal bit position
$D017  53271  Sprites 0-7, expand  Y direction
$D018  53272  Memory control register
                7-4 Video matrix base address (in VIC
window)
                3-1 Character ROM base address (in VIC
window)
$D019  53273  IRQ flag register
                  7 Set on any VIC IRQ condition
                  3 Light pen IRQ
                  2 Sprite to sprite collision
                  1 Sprite to background collision
                  0 Raster compare IRQ
$D01A  53274  IRQ mask reg.
                7-4 UNUSED
                3-0 As above (1 = interrupt enabled)
$D01B  53275  Sprite priority (0 = bckground priority)
$D01C  53276  Sprite multicolour (1 = M.C.M.)
$D01D  53277  Expand Sprite in X direction
$D01E  53278  Sprite-Sprite collision
$D01F  53279  Sprite-data collision
$D020  53280  Border colour
$D021  53281  Screen colour 0
$D022  53282  Screen colour 1
$D023  53283  Screen colour 2
$D024  53284  Screen colour 3
$D025  53285  Sprite multicolour 0
$D026  53286  Sprite multicolour 1
$D027  53287  Sprite 0 colour
$D028  53288  Sprite 1 colour
$D029  53289  Sprite 2 colour
$D02A  53290  Sprite 3 colour
$D02B  53291  Sprite 4 colour
$D02C  53292  Sprite 5 colour
$D02D  53293  Sprite 6 colour
$D02E  53294  Sprite 7 colour
```

6581 SOUND INTERFACE CHIP (SID)

```
$D400  54272  Voice 1 frequency control low byte
$D401  54273  Voice 1 frequency control high byte
$D402  54274  Voice 1 pulse width low byte
$D403  54275  Voice 1 pulse width high nybble (bits 3-0)
$D404  54276  Voice 1 control
```

```
                   7 Select random noise (1 = ON)
                   6 Select pulse waveform (1 = ON)
                   5 Select sawtooth waveform (1 = ON)
                   4 Select triangle waveform (1 = ON)
                   3 Test bit (1 = disable voice 1)
                   2 Ring modulate voice 1 with voice 3 O/P (1
                = ON)
                   1 Synchronise voice 1 with voice 3 freq. (1
                = ON)
                   0 Gate bit (1 = start AD, 0 = start
                release)
$D405   54277   Voice 1 envelope
                 7-4 Select ATTACK cycle duration
                 3-0 Select DECAY cycle duration
$D406   54278   Voice 1 envelope
                 7-4 Select SUSTAIN cycle duration
                 3-0 Select RELEASE cycle duration
$D407   54279   Voice 2 frequency control low byte
$D408   54280   Voice 2 frequency control high byte
$D409   54281   Voice 2 pulse width low byte
$D40A   54282   Voice 2 pulse width high nybble (bits 3-0)
$D40B   54283   Voice 2 control
                   7 Select random noise (1 = ON)
                   6 Select pulse waveform (1 = ON)
                   5 Select sawtooth waveform (1 = ON)
                   4 Select triangle waveform (1 = ON)
                   3 Test bit (1 = disable voice 2)
                   2 Ring modulate voice 2 with voice 1 O/P (1
                = ON)
                   1 Synchronise voice 2 with voice 1 freq. (1
                = ON)
                   0 Gate bit (1 = start AD, 0 = start
                release)
$D40C   54284   Voice 2 envelope
                 7-4 Select ATTACK cycle duration
                 3-0 Select DECAY cycle duration
$D40D   54285   Voice 2 envelope
                 7-4 Select SUSTAIN cycle duration
                 3-0 Select RELEASE cycle duration
$D40E   54286   Voice 3 frequency control low byte
$D40F   54287   Voice 3 frequency control high byte
$D410   54288   Voice 3 pulse width low byte
$D411   54289   Voice 3 pulse width high nybble (bits 3-0)
$D412   54290   Voice 3 control
                  7 Select random noise (1 = ON)
                  6 Select pulse waveform (1 = ON)
                  5 Select sawtooth waveform (1 = ON)
                  4 Select triangle waveform (1 = ON)
                  3 Test bit (1 = disable voice 3)
```

```
                2 Ring modulate voice 3 with voice 2 O/P (1
              = ON)
                1 Synchronise voice 3 with voice 2 freq. (1
              = ON)
                0 Gate bit (1 = start AD, 0 = start release)
$D413  54291  Voice 3 envelope
              7-4 Select ATTACK cycle duration
              3-0 Select DECAY cycle duration
$D414  54292  Voice 3 envelope
              7-4 Select SUSTAIN cycle duration
              3-0 Select RELEASE cycle duration
$D415  54293  Filter cutoff frequency low nybble (bits 2-0)
$D416  54294  Filter cutoff High byte
$D417  54295  Filter resonance control
              7-4 Select filter resonance
                3 Filter external input (1 = YES)
                2 Filter voice 1 O/P (1 = yes)
                1 Filter voice 2 O/P (1 = yes)
                0 Filter voice 3 O/P (1 = yes)
$D418  54296  Filter/volume
                7 Cutoff voice 3 O/P (1 = OFF)
                6 Filter high pass mode (1 = ON)
                5 Filter band pass mode (1 = ON)
                4 Filter low pass mode (1 = ON)
              3-0 Select output volume
$D419  54297  A/D converter 1
$D41A  54298  A/D converter 2
$D41B  54299  Voice 3 random Number generator
$D41C  54300  Envelope 3 output
```

6526 COMPLEX INTERFACE ADAPTOR 1 (CIA #1)

```
$DC00  56320  Data port A
              7-0 Keyboard column select (write)
              7-6 Read paddles on port (01 = A, 10 = B)
                5 UNUSED
                4 Joystick 2 fire button (read) (1 = fire)
              3-0 Joystick 2 direction (read)
              3-2 Paddle fire buttons
$DC01  56321  Data port B
              7-0 Keyboard row value (read)
                7 Timer B toggle/pulse output
                6 Timer A toggle/pulse output
                5 UNUSED
                4 Joystick 1 fire button (read) (1 =fire)
              3-0 Joystick 1 direction (read)
              3-2 Paddle fire buttons
$DC02  56322  D.D.R.A (Bit = 1 for O/P, 0 for I/P)
$DC03  56323  D.D.R.B (Bit = 1 for O/P, 0 for I/P)
```

```
$DC04  56324  Timer A low
$DC05  56325  Timer A high
$DC06  56326  Timer B low
$DC07  56327  Timer B high
$DC08  56328  T.O.D. clock 1/10 secs register
$DC09  56329  T.O.D. clock Seconds register
$DC0A  56330  T.O.D. clock Minutes register
$DC0B  56331  T.O.D. clock
                7 AM/PM flag
              6-0 Hours register
$DC0C  56332  Serial data reg
$DCOD  56333  Interrupt control register
                7 IRQ occurred flag
              6-5 UNUSED
                4 FLAG1 - cassette read / serial SRQ input
                3 Serial port IRQ
                2 TOD clock alarm IRQ
                1 Timer A IRQ
                0 Timer B IRQ
$DC0E  56334  Control register A
                7 TOD frequency (1 = 50 Hz, 0 = 60 Hz)
                6 Serial port mode (1 = O/P, 0 = I/P)
                5 Timer A count (1 = CNT pulses, 0 = o2
              clock)
                4 Force load timer A? (1 = YES)
                3 Timer A run (1 = one-shot, 0 = continuous)
                2 Timer A O/P to PB6 (1 = toggle, 0 = pulse)
                1 Timer A output on PB6? (1 = YES)
                0 Start/stop timer A (1 = START, 0 = STOP)
$DC0F  56335  Control register B
                7 Set alarm/TOD clock (1 = ALARM, 0 = CLOCK)
              6-5 Timer B mode select:
                  00 = Count system o2 clock pulses
                  01 = Count +ve CNT transitions
                  10 = Count timer A underflow pulses
                  11 = Count timer A underflow while CNT +ve
                4 Force load timer B? (1 = YES)
                3 Timer B run (1 = one-shot, 0 = continuous)
                2 Timer B O/P to PB7 (1 = toggle, 0 = pulse)
                1 Timer B output on PB7? (1 = YES)
                0 Start/stop timer B (1 = START, 0 = STOP)
```

6526 COMPLEX INTERFACE ADAPTOR 2 (CIA #2)

```
$DD00  56576  Data port A
                7 Serial bus DATA INPUT
                6 Serial bus CLK pulse INPUT
                5 Serial bus DATA OUTPUT
```

```
                 4 Serial bus CLK pulse OUTPUT
                 3 Serial bus ATN OUTPUT
                 2 RS-232 DATA OUTPUT
               1-0 VIC memory bank select
$DD01   56577  Data port B
                 7 USER PORT / RS-232 DSR
                 6 USER PORT / RS-232 CTS
                 5 USER PORT
                 4 USER PORT / RS-232 Carrier Detect
                 3 USER PORT / RS-232 Ring Indicator
                 2 USER PORT / RS-232 DTR
                 1 USER PORT / RS-232 RTS
                 0 USER PORT / RS-232 Received Data
$DD02   56578  D.D.R.A (Bit = 1 for O/P, 0 for I/P)
$DD03   56579  D.D.R.B (Bit = 1 for O/P, 0 for I/P)
$DD04   56580  Timer A low
$DD05   56581  Timer A high
$DD06   56582  Timer B low
$DD07   56583  Timer B high
$DD08   56584  T.O.D. clock 1/10 secs register
$DD09   56585  T.O.D. clock Seconds register
$DD0A   56586  T.O.D. clock Minutes register
$DD0B   56587  T.O.D. clock
                 7 AM/PM flag
               6-0 Hours register
$DD0C   56588  Serial data reg
$DDOD   56589  Interrupt control
                 7 NMI occurred flag
               6-5 UNUSED
                 4 FLAG1 - USER PORT / RS-232 Received Data
                 3 Serial port NMI
                 2 TOD clock alarm NMI
                 1 Timer A NMI
                 0 Timer B NMI
$DD0E   56590  Control register A
                 7 TOD frequency (1 = 50 Hz, 0 = 60 Hz)
                 6 Serial port mode (1 = O/P, 0 = I/P)
                 5 Timer A count (1 = CNT pulses, 0 = o2
               clock)
                 4 Force load timer A? (1 = YES)
                 3 Timer A run (1 = one-shot, 0 = continuous)
                 2 Timer A O/P to PB6 (1 = toggle, 0 = pulse)
                 1 Timer A output on PB6? (1 = YES)
                 0 Start/stop timer A (1 = START, 0 = STOP)
$DD0F   56591  Control register B
                 7 Set alarm/TOD clock (1 = ALARM, 0 = CLOCK)
               6-5 Timer B mode select:
                   00 = Count system o2 clock pulses
                   01 = Count +ve CNT transitions
```

```
  10 = Count timer A underflow pulses
  11 = Count timer A underflow while CNT +ve
4 Force load timer B? (1 = YES)
3 Timer B run (1 = one-shot, 0 = continuous)
2 Timer B O/P to PB7 (1 = toggle, 0 = pulse)
1 Timer B output on PB7? (1 = YES)
0 Start/stop timer B (1 = START, 0 = STOP)
```

SECTION 2.
RAM GUIDE

HEX DEC

$00 D6510 6510 ONBOARD DATA DIRECTION REGISTER 0

This location is used to define the hardware I/O port at $01. It is discussed in greater detail in the I/O guide.

$01 R6510 6510 ONBOARD I/O PORT 1

This location is a hardware bi-directional I/O port. It is discussed in greater detail in the I/O guide.

$02 UNUSED 2

$03-$04 ADRAY1 JUMP VECTOR: CONVERT FLPT TO INTEGER 3-4

This vector points to the routine at $B1AA (FACINX) to convert a number from flpt into an integer. This vector, however is not actually used by the BASIC interpreter. It can still be used by the programmer who needs to perform the conversion in a machine code program that interacts with BASIC, as the vector will point to the routine regardless of changes that may have been made to the BASIC ROM by Commodore. See also ($311) USRADD.

$05-$06 ADRAY2 VECTOR: CONVERT INTEGER TO FLPT 5-6

This vector points to the routine at $B391 (GIVAYF) to convert a signed integer into flpt. This vector, however is not actually used by the BASIC interpreter. It can still be used by the programmer to convert numbers into flpt, as the vector will point to the routine regardless of changes that may have been made to the BASIC ROM by Commodore. See also ($311) USRADD.

$07 CHARAC SEARCH CHARACTER 7

CHARAC is used by the BASIC routines that scan the input buffer at $0200. Its purpose is to detect certain significant characters such as quotes, commas, and colons by holding their ASCII values during the search. This location is also used by other BASIC routines.

$08 ENDCHR FLAG:SCAN FOR QUOTE AT END OF STRING 8

ENDCHR is used extensively as a work byte during the tokenisation of BASIC text. It is also used as a search character for quotes and colon terminators in the same way as CHARAC.

$09 TRMPOS SCREEN COLUMN BEFORE LAST TAB 9

TRMPOS is used by both TAB and SPC. The cursor position is moved here from $D3 (PNTR), and is used to calculate the final cursor position. This value represents the cursor position on a logical line, and so can range from 0 to 79.

0A VERCK FLAG: 0 = LOAD, 1 = VERIFY 10

The BASIC interpreter uses a single KERNAL routine for both LOAD and VERIFY. Which is performed depends on the state of (A) on entry to the routine. This is either 0 for LOAD or 1 for VERIFY. VERCK is set by the BASIC routine, and another, similar flag at $93 is set by the KERNAL routine.

$0B COUNT INPUT BUFF POIN./NUMBER OF SUBSC. 11

COUNT is used by two routines. Firstly by the routines to process text from the input buffer at $0200, as a pointer to the current working position in the buffer. Once text processing has been completed, COUNT is equal to the length of the text.

COUNT is also used by the routines that deal with arrays to calculate the number of DIMensions required and the amount of memory needed for a newly created array. It is also used to hold the number of subscripts during the referencing of an element within an array.

$0C DIMFLG FLAG: DEFAULT ARRAY DIM 12

DIMFLG is used by the routines that create or reference an array to determine, firstly, that the variable concerned is an array, secondly, whether it has already been DIMensioned, and thirdly, whether a new array assumes specified or default DIMensions.

$0D VALTYP DATA TYPE (STRING OR NUMERIC) 13

VALTYP is used whenever a variable is created or located to determine whether the data is string or numeric. A value of #00 indicates numeric data, and #FF indicates string data. The data type is decoded from the information stored with the variable in high RAM (see $B08B, IDENTIFY VARIABLE).

$0E INTFLG DATA TYPE (INTEGER OR FLPT) 14

If a test on VALTYP has determined data to be numeric, then a further test is performed on this location to determine

the type of numeric data. A value of #00 indicates a flpt number, and a value of #80 indicates an integer.

$0F GARBFL FLAG:DATA SCAN QUOTE/GARBAGE COLLECT 15

GARBFL is used by three major sections of the BASIC interpreter. The LIST routine uses it as a flag to indicate whether or not quotes mode is on. If it is, then the BASIC text string is output directly. If it is not, then the string is scanned for keyword tokens.

GARBFL is used by the garbage collection routines to indicate that garbage collection has already been attempted. If there is still insufficient memory to create a new dynamic string then an ?OUT OF MEMORY error results.

GARBFL is finally used as a general work area in converting a line of BASIC text from the input buffer at $0200 into a tokenised and linked program line.

$10 SUBFLG FLAG: SUBSCRIPT REF/USER FUNCTION CALL 16

SUBFLG is used during the process of finding or creating a variable. The flag is set if an open parenthesis is found immediately after the variable name. This indicates that the variable is either an array or a user-defined function (FN).

$11 INPFLG FLAG:DIRECT DATA TO INPUT/GET/READ 17

INPUT, READ and GET all perform similar functions, thus the BASIC interpreter uses some of the same routines for all three keywords. However, they do need to be separated for the areas in which they differ. This flag indicates which of the keywords is currently being executed (#00 = INPUT, #40 = READ, #98 = GET).

The main areas of difference are:- INPUT displays a ? prompt and echos typed characters on the screen. It also waits for a carriage return (CHR$(13)) before processing the text. GET accepts whatever is the first character in the keyboard buffer without waiting for a keypress (it will even accept null values). READ takes its data from within the BASIC program, and so must search for the next valid DATA statement.

$12 TANSGN FLAG: TAN SIGN / COMPARISON RESULT 18

TANSGN indicates whether the result of a TAN or SIN operation is positive or negative.

TANSGN is also used by the comparison routines ($B016 - $B07D) to indicate the result of a comparison between two variables. Note that the two variables being compared must be of the same type, or ?TYPE MISMATCH error occurs. If the two variables being compared are A and B, then if A > B, the result is #01. If A = B, the result is #02. If A < B, the result is #04. When more than one operator is being used, then the value of TANSGN is a combination of all the results.

$13 CHANNL CURRENT I/O CHANNEL 19

During the input and output of data from BASIC, CHANNL is examined to see which I/O device is to be used. This is not to determine where the data is to be sent ($B8 is used for this purpose), but to determine what is needed in terms of prompting and output of control characters.

When the default input device (keyboard) or default output device (screen) is being used then the value in CHANNL = #00.

For all other input or output devices, the value in CHANNL is the logical file number (or CMD channel) being used. BASIC is thus informed that, for instance, where TAB would output cursor right characters to the screen, it must output spaces to other devices. During input, the ? prompt is suppressed, as is the printing of ?EXTRA IGNORED or ?REDO FROM START messages.

The flag is cleared to zero whenever an error occurs or GET, GET#, INPUT, INPUT# or PRINT# commands have been completed. This effectively redirects output to the screen, and CMD will have to be used again to redirect the I/O. Note that PRINT and LIST do not reset this location.

$14-$15 LINNUM INTEGER VALUE 20-21

LINNUM is used to hold the destination line number for ON, GOTO, GOSUB, LIST, and RUN. The line number is a two byte integer in lo-hi format. It is also used to point to the address for use in the PEEK, POKE, WAIT and SYS commands. The LIST routine also stores the highest line number to be listed here, or #FFFF if the whole program is listed.

GOTO makes a test to see if the target line number is

greater than the current line number. If so, then text is searched from its current position, otherwise it must be searched from the beginning. This test, however is only performed on the most significant byte.

$16 TEMPPT POINTER: TEMPORARY STRING STACK 22

TEMPPT points to the next available space in the temporary string descriptor stack held at $19 - $21 (For detail on the construction of a string descriptor, see the notes on the descriptor stack). There is room for three descriptors on the stack. When the stack is empty, TEMPPT = #19. Its value increases by three whenever a new descriptor is added, until when it is full, its value is #22. Should further attempts be made by BASIC to add to this stack, then a ?FORMULA TOO COMPLEX error is generated.

$17-$18 LASTPT LAST TEMPORARY STRING ADDRESS 23-24

This is a pointer to the location of the last temporary string descriptor in the descriptor stack. The address is in the normal lo-hi format, but since the string stack is wholly contained in zero page, the MSB ($18) will always be zero. The LSB will be 3 less than the value in TEMPPT.

$19-$21 TEMPST STACK FOR TEMPORARY STRINGS 25-33

The temporary string stack contains the descriptors of strings that have not been assigned to a variable. These may be literal strings for printing, e.g. PRINT "ERIK", or the intermediate product of a string manipulation, e.g. X$= MID$(LEFT$(A$,2)).

The data held on each string consists of its two byte absolute start address and one byte indicating its length. Thus, the descriptor stack can contain the descriptors of 3 temporary strings.

$22-$25 INDEX UTILITY POINTER AREA 34-37

INDEX consists of four locations, used by the majority of BASIC routines to hold temporary pointers and the results of calculations. ($22) is often referred to as INDEX1, and ($24) as INDEX2.

$26-$2A RESHO TEMP FAC FOR PRODUCT OF MULTIPLY 38-42

These 5 bytes form a floating point accumulator (fac) which is used as a store for intermediate results during the flpt multiply routines. It is also used by the array creation

routine. See also $61-$70.

$2B-$2C TXTTAB POINTER: START OF BASIC TEXT 43-44

TXTTAB points to the first byte of BASIC text. This is normally $0801. This pointer may be changed to set the start of BASIC to some other location. Reasons for doing this include:

1. PET emulation (BASIC text here starts at $0401). This is helpful in transferring programs from the 64 to a PET, since the PET always loads programs at their absolute start address and not at the start of BASIC as does the 64.

2. Saving an area of memory to tape or disk that is not a normal BASIC program. This pointer should be set to the start of the area of memory, and ($45) must be set to the end address +1 of the area of memory to be saved.

3. Raising the start of BASIC text so that a safe area is created in low RAM. This can then be used for sprite data, graphics screens etc.

4. Storing more than one BASIC program in memory at one time. Each program can be accessed by setting the pointers to its start and end. Also programs can be merged or appended using this technique.

It is important to set the byte preceeding the start of BASIC text to zero ($0800 in the 64, $0400 in the PET) before trying to use the new text area.

$2D-$2E VARTAB POINTER: START OF BASIC VAR. 45-46

VARTAB points to the end address +1 of BASIC text, and the start address of variable data storage. All variables are stored from this address, except for arrays, which are stored in their own separate area. The descriptors to strings are stored here, but the actual strings are stored right at the top end of RAM. Each variable or string descriptor is 7 bytes long, consisting of 2 bytes for the variable name and 5 bytes for the variable data. A detailed description of variable structure can be found under $B08B in the ROM guide.

Variables are stored in the order in which they are created. They are also searched for in the same order. Variables that are used frequently should be assigned at the start of the program in order to achieve the fastest possible

execution speed. Try not to assign a variable and then only use it once. The variable cannot be deleted from the table and must be searched past to find other variables.

Because arrays are stored immediately after the variable table, any arrays that have been created must be moved up by 7 bytes every time a new variable is created.

When the commands CLR, NEW, RUN or LOAD are executed, VARTAB is reset to one byte following the end of BASIC text. Modifying a program will cause this pointer to be moved to the new end of BASIC text. This will result in the loss of the variable table.

When LOAD is executed from within a program, VARTAB is not reset. This allows chained programs to share the same variables. This assumes that the chained program is shorter than the calling program, and does not overwrite the variable table. It is not possible to use previously defined functions from chained programs.

$2F-$30 ARYTAB POINTER: START OF BASIC ARRAYS 47-48

ARYTAB points to the end of the ordinary variable table and the start of the array variable table. The format of array variables is detailed under $B194 in the ROM guide. The actual data contained in the array is stored after the array header in the same format as ordinary variables. The major difference, however is that they only take up the space required. Thus, flpt variables use 5 bytes, integers use 2 bytes, and string descriptors use 3 bytes. As with ordinary strings, array strings are actually stored in the dynamic string area, with only the descriptor in the array table.

$31-$32 STREND POINTER: END OF BASIC ARRAYS +1 49-50

STREND points to the end of the BASIC array table +1 and the start of free RAM. The function FRE(n) returns the difference between the end of arrays (STREND) and the bottom of strings (FRETOP).

Should the addition of a new dynamic string cause FRETOP to be lower than STREND (or the addition of an array or ordinary variable), then garbage collection is performed to remove all strings that are no longer pointed to by a descriptor. If FRETOP is still lower than STREND, then an ?OUT OF MEMORY error results.

$33-$34 FRETOP POINTER: BOTT OF STRING STORAGE 51-52

FRETOP points to the end of the dynamic string storage area and the top of free RAM. When a new string is created, it is added to the bottom of the dynamic string area, and

FRETOP adjusted down to accomodate it. This may cause the garbage collect routines to be called (see STREND).

$35-$36 FRESPC UTILITY STRING POINTER 53-54

FRESPC is used by the routines that manipulate strings as a pointer to the most recent string to be worked on.

$37-$38 MEMSIZ POIN: HIGHEST ADD. USED BY BAS. 55-56

MEMSIZ contains the highest address of RAM available to the BASIC interpreter. It is normally $9FFF, but is set to $7FFF by the power-up routine if there is an external ROM cartridge present.
This pointer may be lowered by the user to create a safe area of RAM that will not be disturbed by BASIC. CLR must be issued after MEMTOP is lowered, to ensure that strings do not overwrite the protected area.

When a file is opened to the RS-232 port, MEMTOP is lowered by 512 bytes in order to create the input and output buffers for the port. The computer automatically performs CLR at this point, so any variables created will be lost.

$39-$3A CURLINCURRENT BASIC LINE NUMBER 57-58

CURLIN holds the current line number that is being executed. If the computer is in direct mode, ie. no program is currently being run, then $3A holds #FF. This is used by certain keywords to check the mode in case of an ?ILLEGAL DIRECT error.

CURLIN is updated as each new line is executed. Valid line numbers are in the range 0 to 63999. Error messages use CURLIN to indicate the location of an error.

$3B-$3C OLDLIN PREVIOUS BASIC LINE NUMBER 59-60

When STOP, END or break are performed, the contents of CURLIN are copied into this location. $3A is then set to #FF to indicate direct mode. If CONT is performed, then the contents of OLDLIN are copied into CURLIN, and execution continues.

$3D-$3E OLD TXTPOINTER: ADD. OF STAT. FOR CONT 61-62

OLDTXT holds the address (not line number) of the BASIC statement being exexcuted. Each time a new keyword is executed, the value of TXTPTR ($7A) is copied into OLDTXT. When CONT is performed, then OLDTXT is copied back into TXTPTR. If LOAD is executed, or the program was stopped by an error, or the program has been modified, then $3E is set to #00, and CONT produces a ?CAN'T CONTINUE error.

$3F-$40 DATLIN CURRENT DATA LINE NUMBER 63-64

DATLIN holds the line number of the current DATA statement being READ. This is only used as a reference for when an ?OUT OF DATA error occurs, as the actual datum is referred to by DATPTR ($41).

$41-$42 DATPTR POINTER: CURRENT DATA ITEM ADD. 65-66

DATPTR holds the address (not line number) of the current datum being accessed by READ. If this pointer becomes greater than the contents of VARTAB during a READ operation, then ?OUT OF DATA results. DATPTR is reset to the start of BASIC text by the RESTORE command.

$43-$44 INPPTR VECTOR: INPUT ROUTINE 67-68

This vector points to the source of the data being read by the INPUT. GET and READ routines. For INPUT and GET, this vector points to the input buffer at $0200. For READ, it points to the current DATA statement.

$45-$46 VARNAM CURRENT BASIC VARIABLE NAME 69-70

When a variable is being searched for, its name is placed in VARNAM. The format of the variable name is firstly an alphabetic character, then an optional alphanumeric character. The ASCII values of these characters are modified by the variable type concerned. Flpt variables are unmodified, integer variables have #80 added to both characters, string variables have #80 added to the second character, and defined functions have #80 added to the first character.

$47-$48 VARPNT POIN: START OF CURRENT VAR DATA 71-72

VARPNT points to the third byte of the descriptor to the current variable being accessed. For numeric variables, this is the start of the actual data, but for string variables, this is the string descriptor.

$49-$4A FORPNT POINTER: INDEX VAR. FOR NEXT 73-74

FORPNT holds the address of the variable being used to index

a FOR/NEXT loop. The address is pushed onto the stack by the FOR/NEXT routines so that FORPNT can be used as a work area by other BASIC keywords.

$4B-$4C OPPTR OPERATOR TABLE DISPLACEMENT 75-76

OPPTR holds the displacement of the current mathematical operator being used in the operator table at $A080.

$4D OPMASK MASK FOR COMPARISON 77

During the evaluation of a BASIC expression, OPMASK is used to indicate the type of comparison being used (<=>).

$4E-$4F DEFPNT POINTER: CURRENT FN DESCRIPTOR 78-79

DEFPNT is used as a pointer to the address of the variable data in which the results of an FN operation is stored. Its operation is much the same as for VARPNT.

$50-$52 DSCPNT POINTER: CURR STRING DESCRIPTOR 80-82

When a string is being assigned, the first two bytes of DSCPNT are used as a temporary pointer to the string descriptor, and the third byte holds the string length.

$53 FOUR6 CONSTANT FOR GARBAGE COLLECTION 83

FOUR6 is used by the garbage collect routines to indicate whether a 3 byte array string descriptor, or a 7 byte ordinary string descriptor is being collected.

$54-$56 JMPER 6510 JMP TO FUNCTION 84-86

This area consists of the 6510 JMP instruction (#4C) followed by a two byte function address taken from the table at $A052. It is used as a linking subroutine by the function evaluation routines at $AFB1 and $AFD1.

$57-$60 TEMP DATA AREA 87-96

This area is used by many BASIC routines as a temporary work area.

$61 FACEXP FAC#1: EXPONENT 97

The two flpt accumulators (fac) are used by the BASIC interpreter for all of its calculations. The results of the calculations are held in fac#1 ($61-$66). A flpt number consists of three parts: mantissa, exponent, and sign. The mantissa is the normalised value of the flpt number (between 1.00000 and 1.99999). The exponent is a power of 2. When the mantissa is multiplied by 2 raised to the power of the exponent, then the result is the actual value of the flpt number.

FACEXP is the exponent value for FAC#1. In order to take into account negative numbers, #81 is added to the value in FACEXP. Thus #80 represents 0, #81 (2^0) = 1, #82 (2^1) = 1, #83 (2^2) = 4 etc.

When the value in FACEXP indicates a zero number, the contents of the mantissa and sign byte are ignored by the BASIC interpreter.

$62-$65 FACHO FAC#1: MANTISSA 98-101

FACHO is the 4 byte numeric part of fac#1. Since the value of the number is normalised before being stored here, it will always be in the range 1.00000 to 1.99999. The most significant digit of the number is therefore always going to be 1. Knowing this, the most significant digit is not actually needed, thus when a flpt number is stored as a variable, the most significant bit is used to represent the sign of the number, with the remaining 31 bits representing the number.

When a flpt number is converted into an integer, the result is stored in the first two bytes of FACHO ($62).

$66 FACSGN FAC#1: SIGN 102

FACSGN represents the sign of the flpt number in fac#1. #00 indicates a positive number, while #FF represents a negative number.

$67 SGNFLG CONSTANT FOR SERIES EVALUATION 103

SCNFLG represents the number of terms to be used in the series calculations for evaluating LOG, EXP and trig functions.

$68 BITS FAC#1: OVERFLOW DIGIT 104

BITS is the overflow byte used in converting integers or ASCII strings into flpt numbers.

$69 ARGEXP FAC#2: EXPONENT 105

ARGEXP is the exponent of the second flpt accumulator, fac#2. This accumulator is used whenever flpt operations are undertaken that involve more than one flpt number. The result of such operations is always left in fac#1. The format of this accumulator is identical to fac#1.

$6A-$6D ARGHO FAC#2: MANTISSA 106-109

ARGHO is the 4 byte mantissa for fac#2. It is identical to FACHO.

$6E ARGSGN FAC#2: SIGN 110

ARGSGN is the sign byte for fac#2. It is identical to FACSGN.

$6F ARISGN FAC#1 Vs FAC#2 SIGN COMPARISON RESULT 111

ARISGN is used to indicate whether or not the two flpt accumulators have like sign bytes. #00 indicates like signs, and #FF, unlike signs.

$70 FACOV FAC#1 LOW ORDER (ROUNDING) 112

FACOV is used to extend the accuracy of intermediate flpt mathematical operations when the mantissa of fac#1 is too large to fit into 32 bits. This byte will then hold the low order bits of the number. FACOV is also used for rounding of the final result to be placed in FACHO.

$71-$72 FBUFPT POINTER: CASSETTE BUFFER 113-114

FBUFPT does not appear to be used by any of the BASIC routines to point to the cassette buffer. Instead, it is used as a work byte during formula evaluation, string setup and TI$ manipulation.

$73-$7A CHRGET SUB.: GET NEXT BYTE OF BASIC 115-138

CHRGET is a machine language subroutine which is copied from ROM at $E3A2 into this location at power-up. It is used by the BASIC interpreter to read characters of text from the BASIC text area or the input buffer at $0200. It is discussed in detail at the beginning of the ROM guide.

$79 CHRGOT SUBROUTINE: GET CURR BYTE OF BASIC 121

CHRGOT is an entry point to the CHRGET subroutine which places into (A) the byte of BASIC text currently pointed to by TXTPTR. It is used by BASIC routines to re-read the byte of text being worked on.

$7A-$7B TXTPTR POINTER: CURRENT BYTE OF BASIC 122-123

TXTPTR holds the address of the current byte of BASIC text being executed. It forms an integral part of the CHRGET and CHRGOT subroutines, and is incremented every time that CHRGET is called.

$8B-$8F RNDX SEED VALUE FOR RND 139-143

RNDX is a 5 byte flpt value representing the seed for the RND function random number generator. It is copied into RAM from ROM along with the CHRGET subroutine at power-up. Its initial value is #80 4F C7 52 58. See also $E097 in the ROM guide.

$90 STATUS KERNAL I/O STATUS WORD 144

STATUS contains the current status of I/O operations being undertaken by KERNAL routines. It is identical in its structure to the BASIC reserved variable, ST. RS-232 routines use RSSTAT ($297) as a status byte. See also the KERNAL routine READST.

$91 STKEY FLAG: STOP KEY PRESSED 145

STKEY is updated 60 times each second by the IRQ interrupt service routine. Its function is to indicate whether the STOP key has been pressed. During the IRQ service routine, the value of the row of the keyboard matrix that holds the STOP key is stored in STKEY. This has the additional benefit that the 7 other keys on this row of the matrix can be detected. The values returned for the 8 possible keys pressed (and no key) are detailed below:

```
255 #FF = NO KEY
254 #FE = <1>    PRESSED      239 #EF = <SPACE> PRESSED
253 #FD = < >    PRESSED      223 #DF = <CBM>   PRESSED
251 #FB = <CTRL> PRESSED      191 #BF = <Q>     PRESSED
247 #F7 = <2>    PRESSED      127 #7F = <STOP>  PRESSED
```

$92 SVXT TAPE TIMING CONSTANT 146

This is an 'adjustable constant', used during tape read operations. It is adjusted in order to compensate for

variations in the tape speed and for variations in the speed of tapes recorded on different cassette units.

$93 VERCK KERNAL FLAG: LOAD/VERIFY 147

The same KERNAL routine is used for both LOAD and VERIFY operations. This location is used to determine whether the data being loaded from tape or disk is to be stored in RAM or compared against RAM.

$94 C3PO FLAG: CHARACTER READY FOR SERIAL BUS 148

C3PO is used to indicate when there is a character in the serial bus data buffer (BSOUR) awaiting output to the bus.

$95 BSOUR BUFFERED CHARACTER FOR SERIAL BUS 149

This is the character waiting to be sent to the serial bus. #FF in this location indicates that there is no character awaiting output.

$96 SYNO CASSETTE SYNC NUMBER 150

SYNO is a constant used by the cassette handling routines.

$97 XSAV TEMP DATA AREA 151

XSAV is used as a temporary (X) register save area during many of the KERNAL I/O routines.

$98 LDTND NUM OF OPEN FILES/INDEX TO FILE TABLE 152

LDTND holds the number of currently open I/O files. This number is also used as an index to the end of the logical file number, device and secondary address tables held at $259-$277.

OPEN causes this value to be incremented to its maximum value of 10, while CLOSE causes this value to be decremented. The KERNAL CLALL routine sets this location to zero, effectively closing all files.

$99 DFLTN DEFAULT INPUT DEVICE 153

DFLTN is normally set to #00 to indicate the keyboard as input device. The device number can be changed by the KERNAL routine CHKIN which opens an input channel.

The BASIC routines INPUT and INPUT# call CHKIN to set DFLTN to the required device, however the device number is reset on termination of the routines.

$9A DFLTO DEFAULT OUTPUT (CMD) DEVICE 154

DFLTO is normally set to #03 to indicate the screen as output device. The device number is normally changed by the KERNAL routine CHKOUT which opens an output channel.

The BASIC routines PRINT# and CMD call CHKOUT to set DFLTN to the required device, however the device number is reset on termination of the PRINT# routine.

$9B PRTY TAPE CHARACTER PARITY 155

PRTY is used to determine the parity check performed when tape read/write operations are performed. Parity is calculated by determining the number of 1s or 0s present in the byte. The parity bit is set depending on whether this number is odd or even.

$9C DPSW FLAG: TAPE BYTE RECEIVED 156

DPSW is set once all 8 bits of a byte have been received from the tape.

$9D MSGFLG FLAG: CONTROL KERNAL MESSAGE OUTPUT 157

MSGFLG is set by the KERNAL routine SETMSG, and it determines whether or not the KERNAL control or error messages are enabled. The possible values for this flag are:
#00 - All KERNAL messages are suppressed. This is the value set when
0 BASIC RUN mode is entered.

#40 - Enable KERNAL error messages only. These consist of "I/O ERROR
64 #" and a number. Since they are not self explanatory, BASIC
prefers to use its own set of error messages.

#80 - Enable KERNAL control messages. These are messages such as
128 "SAVING", "SEARCHING", "FOUND" etc. They are enabled by BASIC
in direct mode but not in RUN mode.

#C0 - Enable both KERNAL error and control messages.
192

$9E PTR1 TAPE PASS 1 ERROR LOG 158

PTR1 is used to set up a log of any bytes in which a tape read error has occurred during the first pass of the data block. (Each block of data is recorded twice to minimise data loss.)

$9F PTR2 TAPE PASS 2 ERROR LOG 159

PTR2 is an index used in correcting those bytes in which an error occurred on the first pass of the tape data block.

$A0-$A2 TIME REAL-TIME JIFFY CLOCK 160-162

The three bytes form a software clock that is incremented by the IRQ service routine 60 times a second. It is therfore a count of thenumber of 60ths of a second (jiffies) that have occurred since the computer was switched on. After 24 hours, the jiffy clock is reset to zero.

The jiffy clock may be accessed via the reserved variables TI and TI$. TI$ may also be used indirectly to change these three bytes (TI$ is effectively TI put into HHMMSS format).

The jiffy clock is updated by the IRQ service routine, so anything that affects the operation of this routine (such as tape or serial bus I/O) will stop the clock until the IRQ routine is restored. This will cause the jiffy clock to become inaccurate.

$A3 SERIAL BIT COUNT / EOI FLAG 163

This location is a counter used by the routines that input and output data to the serial bus. It is first set to the value 8 in readiness to send or receive a byte. As each bit is cycled to or from the bus, this value is decremented until it reaches zero, when it is reset to 8 for the next byte.

This location is also used as a flag to indicate when the EOI (End-Or-Identify) handshake has been received from the bus. EOI is sent on the bus along with the final byte of the message to indicate the end of data transmission.

$A4 CYCLE COUNT 164

This location is used by tape I/O as a temporary counter.

$A5 CNTDN CASSETTE SYNC COUNTDOWN 165

CNTDN is used to count the number of sync pulses sent on the tape before the actual data block.

$A6 BUFPNT POINTER: TAPE I/O BUFFER 166

BUFPNT does not point to the tape buffer directly, but is an index to the last character to be written to the buffer. The tape buffer is 192 bytes long, so the value in BUFPNT can range from 0 to 191 (#BF). When this location reaches #BF, the contents of the tape buffer are written to the tape.

$A7 INBIT RS-232 INPUT BITS/TAPE LEADER SHORT COUNT 167

INBIT stores each bit of serial data as it is received from the RS-232 port. These bits are later shifted together to form the final byte value. The location is also used by the tape I/O routines to detect short block errors.

$A8 BITC1 RS232 IN BIT CNT/TAPE READ ERR/WRT NEW BYTE 168

BITC1 counts the number of bits received from the RS-232 port. This is so that the RS-232 routines know when a complete word has been received. BITC1 is also used to flag read errors and the start of a new byte to be written by the tape I/O routines.

$A9 RINONE RS-232 FLAG: CHK FOR STR BIT/TAPE CNT ZEROS 169

RINONE is used as a flag by the RS-232 routines to indicate whether or not a start bit was received. A value here of #90 indicates that no start bit has been received, while #00 indicates a start bit has been received.
RINONE is also used by the tape I/O handling routines for counting zeros.

$AA RIDATA RS-232 INPUT BYTE/TAPE FUNC MODE/SYNC CNTDN 170

RIDATA is used by the RS-232 input routines to build up each byte as it is received in bit form from the RS-232 port. The byte, once completed is stored in the RS-232 in-buffer, which is pointed to by RIBUF ($F7).
Bits 6 and 7 of RIDATA are used by the tape routines to indicate the current function mode being used. This

includes a flag to indicate whether the current character should be treated as data or a sync character. Bits 0-5 form a counter for sync characters to indicate when the end of a sync block has been reached.

$AB RIPRTY RS-232 INPUT PRTY/CASSETTE SHORT CNT 171

RIPRTY is used by the RS-232 input routines as a flag to detect when a parity error has occurred. For more detail, see the RS-232 section.

RIPRTY is also a flag used by the tape routines to indicate the end of a header or leader block. The flag is set when the block ends before the computer is expecting it to.

$AC-$AD SAL POINTER: TAPE BUF/SCRN SCROLLING 172-173

SAL points to the start address of a block of memory that is going to be saved. During the save, this pointer is incremented until it points to the end address of the block of memory to be saved as indicated by EAL. Once the save is completed, SAL is restored to its start value. When data is being written to tape, SAL points to the start of the tape buffer. It is incremented to the end of the tape buffer in the same way.

SAL is also used as a temporary pointer during screen scrolling and other screen management routines.

$AE-$AF EAL TAPE END ADDS/END OF PROGRAM 174-175

EAL points to the end address +1 of a block of memory to be saved. The save is completed once the contents of SAL are the same as EAL. When data is being written to tape, EAL points to the end of the tape buffer.

$B0-$B1 CMP0 TAPE TIMING CONSTANTS 176-177

These two bytes are sometimes used together as a constant, and are used separately at other times. $B0 is used to calculate the value of the adjustable constant SVXT at $92. $B1 is just used as a timing constant during tape read.

$B2-$B3 TAPE1 POINTER: START OF TAPE BUFFER 178-179

TAPE1 is set to the start address of the cassette tape buffer ($033C). Since the buffer is always referenced through this pointer, it is possible to change the location of the tape buffer by merely adjusting this pointer. It is

not possible to set the buffer to start below $0200, since doing this not only overwrites the system variables, but it causes an ?ILLEGAL DEVICE NUMBER message when used.

$B4 BITTS RS-232 OUT BIT COUNT/TAPE TIMER ENBL 180

BITTS is used by the RS-232 output routines to count the number of bits being transmitted to the port. This is to adjust for the number of stop bits and parity bit.

During tape input operations, BITTS is used as a flag to indicate when the tape timer is enabled. A value of #01 indicates that the timer is enabled and the system is ready to receive a byte.

$B5 NXTBIT RS-232 NEXT BIT TO SEND/TAPE EOT FLAG 181

NXTBIT holds the value of the next bit to be sent on the RS-232 port (#00 or #01). It is also used as a flag by the tape read routines to indicate that an End-Of-Tape (EOT) header has been read.

$B6 RODATA RS-232 OUT BYTE BUF/TAPE READ CHRCTER ERR 182

RODATA holds the byte of data currently being sent on the RS-232 port. Each bit of data is consecutively shifted out into the system carry flag, and then shifted into NXTBIT before being sent.

RODATA is also used as a flag by the tape routines to indicate that a character read error has occurred.

$B7 FNLEN LENGTH OF CURRENT FILENAME 183

FNLEN indicates the number of characters in the current filename. This can be from 0 to 187 for a tape file, from 1 to 16 for a disk file, or 4 characters for an RS-232 file.

Disk files always require a filename to be specified. Tape files do not require a filename to be specified, and so this location may contain #00.

Only the first 16 characters of a tape filename will be displayed on the screen by the SEARCHING FOR, SAVING, and FOUND messages, but the whole filename will be written or searched for. The length of the filename field on the tape header allows short machine language programs to be saved as filenames.

$B8 LA CURRENT LOGIGAL FILE NUMBER 184

LA holds the logical file number of the I/O file that is currently in use. Valid file numbers can range from #01 to #FF. #00 is used to indicate that there is no file currently open. A file number greater than #7F will cause a line feed character (#0A) to be sent with each carriage return.

LA is set up by the KERNAL routine OPEN.

$B9 SA CURRENT SECONDARY ADDRESS 185

SA holds the secondary address of the I/O file that is currently in use. Valid secondary addresses can range from #00 to #7F, although #1F is the usual upper limit for serial devices. Secondary addresses are used to send commands to a device, say to change character set. Since each device has its own set of secondary addresses, each meaning something different, it is not possible to give a comprehensive list here.

SA is set up by the KERNAL routine OPEN.

$BA FA CURRENT DEVICE NUMBER 186

FA holds the number associated with the device currently being used. Device numbers can range from #00 to #1F, but not all of them have been assigned to a particular device. All devices greater than 3 are assumed to be on the serial bus. The devices currently assigned are as follows:

0 - KEYBOARD
1 - TAPE RECORDER
2 - RS-232 PORT
3 - SCREEN
4 - PRINTER
5 - ALTERNATIVE PRINTER
6 - PRINTER/PLOTTER
8 - DISK DRIVE
9 - ALTERNATIVE DISK DRIVE

$BB-$BC FNADR POINTER: CURRENT FILE NAME 187-188

FNADR is a pointer to the start of the filename associated with the current open file. The length of the filename is held in FNLEN. If a null filename is specified, then FNLEN is set to #00, and this location is not used.

$BD ROPRTY RS-232 OUT PRTY/TAPE RECVE INPUT CHRCTR 189

ROPRTY is used by the RS-232 input routines to calculate the output parity bit. For more detail, see the RS-232 section.

It is also used as a temporary store for the character being received from tape.

$BE FSBLK NUMBER OF BLOCKS TO READ/WRITE TO TAPE 190

FSBLK is a counter, used by the tape I/O routines to indicate the number of blocks outstanding in a tape I/O operation.

$BF MYCH SERIAL WORD BUFFER 191

MYCH is used by both the tape and serial bus input routines as a temporary work area for assembling the bits received into the final byte.

$C0 CAS1 TAPE MOTOR INTERLOCK 192

CAS1 holds the current status of the cassette unit motor. When a button is pressed on the cassette unit, this byte is checked. If it holds #00, the motor is turned on and this location is set to #01. Similarly, when a button is released on the cassette unit, this byte is checked, and if it holds #01, the motor is switched off.

By setting the value in this location, the tape motor can be turned on or off. However, the IRQ service routine, which processes this byte will reset the value to its initial value if it is not #00 with no buttons pressed, or #01 with a button pressed.

$C1-$C2 STAL I/O START ADDRESS 193-194

STAL points to the start address of a block of data to be saved or loaded. If the device concerned is tape, then this location points to the start of the tape buffer as indicated by ($B2). This is because the tape buffer is always the first block saved or loaded from tape. The start address of the actual RAM block is indicated by ($C3).

$C3-$C4 MEMUSS DATA START ADDRESS 195-196

MEMUSS points to the start address of a block of memory to be loaded or saved to tape. It is only used for tape operations, since the first block to be recorded must be the

header contained in the tape buffer. Thus, the normal pointer, ($C1) points to the start of the tape buffer.

$C5 LSTX CURRENT KEY PRESSED 197

LSTX holds the value of the current key being pressed. This is not its ASCII value, but a value based on the position of the key in the keyboard matrix. Shift, CTRL and CBM keys are not indicated in this location, neither do they modify the value of any key pressed. RESTORE is not accounted for in the keyboard matrix, since it is connected to the microprocessor NMI line. A value of #40 indicates no key.

$C6 NDX NUMBER OF CHRCTR IN KYBRD BUF (QUEUE) 198

NDX holds the number of characters waiting to be processed in the keyboard buffer, which starts at $0227. Manipulation of this byte can be used to create a 'dynamic keyboard' effect. For example, any keypresses can be removed from the keyboard buffer by the command POKE 198,0. The command RUN can be executed by POKEing the ASCII values of the characters R, U, N and <CARRIAGE RETURN> into the start of the keyboard buffer at $0227 (649 decimal), and setting NDX equal to 4.

$C7 RVS FLAG: PRINT REVERSE CHARACTERS 199

When RVS holds #00, characters are printed to the screen normally. When it contains #12, all characters are printed to the screen in reverse mode. This involves the screen editor adding #80 to the screen code of each character. Pressing CTRL-RVS sets this flag, and CTRL-OFF clears it. Note that the flag is also cleared whenever a carriage return (#0D) is encountered.

$C8 INDX POINTER: END OF LOGICAL LINE FOR INPUT 200

INDX points to the final valid character on the logical screen line that is to be INPUT. Since a logical screen line can be two physical lines long, the pointer can range in value from #00 to #4F (0-79).

$C9-$CA LXSP CURSOR X-Y POS AT START OF INPUT 201-202

LXSP keeps track of the position of the cursor on its current logical line. The format is row in the first byte, and column in the second. Since a logical screen line may be either 40 or 80 columns long, there can be anything between 13 and 25 rows on the screen. See also $D9-$F2.

$CB SFDX KEYBOARD MATRIX COORDINATE 203

The keyboard scanning routine uses SFDX to indicate the current key being pressed. The value here is used as an index to the keyboard decode matrices, from which the ASCII value of the key is derived. This value is the same as that in $C5.

$CC BLNSW CURSOR BLINK ENABLE 204

When BLNSW is set to #00, the cursor blink facility is switched on. A non-zero value will turn the cursor off. The cursor is automatically turned off when the keyboard buffer contains a keypress, or when program execution commences.

$CD BLNCT TIMER: COUNTDOWN TO TOGGLE CURSOR 205

BLNCT is used as a counter by the IRQ service routine to toggle the cursor on or off. Firstly, the location is set to #14, then every IRQ, it is decremented until it finally reaches zero. At this point, the cursor is toggled and the process is started all over again. Normally, the cursor will blink 3 times in each second.

$CE GDBLN CHARACTER UNDER CURSOR 206

GDBLN holds the normal screen code of the character occupying the current cursor position. This is so that when the cursor moves on, the character can be restored to its original value. (The cursor works by switching the character between normal and reversed modes).

$CF BLNON FLAG: CURSOR BLINK PHASE 207

BLNON indicates whether the cursor is currently on (the character under the cursor is reversed), or off. It contains the value #00 if the cursor is on, and #01 if it is off.

$D0 CRSW FLAG: INPUT FROM SCRN OR GET FROM KYBRD 208

The KERNAL routine CHRIN uses CRSW to indicate whether data should be input from the screen or the keyboard.

$D1-$D2 PNT PNTER: CURRENT SCRN LINE ADDRESS 209-210

PNT indicates the start address of the first column of the screen line which currently contains the cursor.

$D3 PNTR CURRENT CURSOR COLUMN ON LINE 211

PNTR holds the column in which the cursor is currently positioned. The value can range from 0 to 79, since each logical screen line can be up to two physical screen lines long. This location holds the number returned by the BASIC POS function. It is possible to change the cursor horizontal position by altering the value contained here.

$D4 QTSW FLAG: EDITOR IN QUOTES MODE 212

A zero in QTSW indicates that quotes mode is off. Any non-zero number indicates that quotes mode is on. Quotes mode is toggled on or off every time the quotes character is typed. The effect of quotes mode is to cause cursor control keys, and other non-printing characters to print a reversed character instead of performing their normal function. The only exception to this is DELETE, which operates normally in quotes mode.

It is possible to escape from quotes mode by pressing RETURN, or SHIFT-RETURN.

$D5 LNMX SCREEN LINE LENGTH 213

LNMX is a flag used by the screen editor to indicate whether the current logical screen line is 40 or 80 columns long. The flag is then referred to when the cursor reaches the end of a line to determine whether the logical line can be extended, or a new logical line must be started.

$D6 TBLX CURRENT CURSOR LINE NUMBER 214

TBLX holds the current physical (not logical) line number on which the cursor is placed. It can range in value from 0 to 24. The cursor vertical position can be altered by changing the value contained here.

$D7 TAPE: MOST RECENT DIPOLE BIT VALUE 215

This location is used by the tape routines to hold bit data used in the building and unpacking of data bytes. It is also used by the screen editor as a temporary store for the last character to be printed.

$D8 INSRT FLAG: INST MODE 216

INSRT is used to indicate whether or not the screen editor is in insert mode. Any number greater than zero indicates

that insert mode is on. Additionally, the value indicates the number of inserts that have been opened.

When INST is pressed, the screen line from the current cursor position is shifted right by one character, another screen line is added to the logical line if necessary, the screen line link table is updated, LNMX is adjusted, and INSRT incremented. The editor is now in quotes mode and all cursor etc. keys (including DELETE) will print a reversed character to the screen. As each character is printed, INSRT is decremented, until it reaches zero, when insert mode is terminated.

$D9-$F2 LDTB1 SCREEN LINE LINK TABLE 217-242

This is a table consisting of 25 bytes. Each byte represents one physical line of the screen. The table is split up into several functions, each performed by different bits within the 1 byte entries. Each function will be looked at separately here.

BITS 0-3. Screen memory on the Commodore 64 is 1000 bytes long. This means that it occupies four pages of computer memory. These three bits indicate on which of these four pages the line starts. The pointer PNT ($D1) can then be calculated by adding this number to the start page of screen RAM, held at $0288, and also adding to it the entry for the particular line in the table of screen address low bytes held at $ECF0.

BIT 7. This is a flag used by the screen editor to indicate the position of the physical line in a logical line. If it is the first line, then it is set to 1. If it is the second line, it is 0.

$F3-$F4 USER PNTR: CURRENT SCRN COLOUR LOC 243-244

USER points to the first byte of the line of colour RAM that corresponds to the current physical screen line. The pointer is synchronised with PNT ($D1).

$F5-$F6 KEYTAB VECTOR: KYBRD DECODE TABLE 245-246

KEYTAB points to the start of the keyboard matrix table currently being used. There are four tables, each returning a unique ASCII code for each of the 64 keys on the keyboard. This is so that different key values can be obtained for the SHIFT, CTRL and CBM keys. The addresses of the keyboard tables are as follows:

$EB81 - DEFAULT UNSHIFTED CHARACATERS
$EBC2 - SHIFTED CHARACTERS
$EC03 - CBM LOGO CHARACTERS
$EC78 - CTRL CHARACTERS

These keyboard matrix tables should not be confused with the toggling of character sets brought about by SHIFT-CBM. The toggling serves only to change the portion of the character shape ROM used to display characters on the screen.

$F7-$F8 RIBUF RS-232 IN BUFFER POINTER 247-248

RIBUF points to the start of the 256 byte input buffer which is set up at the top of memory whenever an RS-232 file is opened.

$F9-$FA ROBUF RS-232 OUT BUFFER POINTER 249-250

ROBUF points to the start of the 256 byte output buffer which is set up at the top of memory whenever an RS-232 file is opened.

$FB-$FE FREKZP FREE 0-PAGE SPACE FOR USER PROG 251-254

These are locations that it is guaranteed BASIC will not alter for use in user written machine code programs. However, these locations are often used by commercial BASIC extension packages.

$FF BASZPT TEMP DATA AREA 255

BASZPT is used as a temporary store during the conversion of flpt numbers into ASCII strings.

PAGE 1

$100-$1FF   MICROPROCESSOR SYSTEM STACK SPACE   256-511

This whole page of memory is reserved for the hardware stack of the 6510 microprocessor. The stack is organised on a Last In, First Out (LIFO) basis, rather like placing cards onto the top of a deck and then removing them, again from the top.

The stack is controlled by a 9-bit register within the microprocessor, called the 'stack pointer' (SP). The low 8 bits of this register point to the last stack location to be used, called 'top of stack'. The high bit indicates the page of memory on which the stack is to be found. Since it cannot be changed, the stack is limited to the range $0100 - $01FF. For most purposes, the high bit of (SP) is ignored, and its contents referred to as being in the range #00 - #FF.

The first number to be placed on the stack will be at $01FF, the second at $01FE and so on. Should more than 256 bytes be pushed onto the stack, (SP) will reset to #FF and an overflow error results. Similarly, if too many bytes are pulled from the stack, (SP) becomes #00, and an underflow error results. These errors do not cause the program to halt, but execution continues using the new but erroneous value of (SP). As a result, the system will go haywire and cause nothing to operate correctly until the system is reset or powered on again.

Most of the BASIC and KERNAL routines make heavy use of the stack for GOSUBs, FOR-NEXT loops, DEF FNs, the storing of return addresses from subroutines, the saving of the processor internal registers during interrupt servicing, etc. Part of the stack space is also used separately by BASIC as a temporary work area.

$100-$10A   FLPT TO ASCII STRING WORK AREA   256-266

This part of the stack is used by BASIC routines in the conversion of numbers into their equivalent ASCII digits ready for printing to the screen etc. This area is protected by BASIC from being overwritten by the stack.

$100-$13E   BAD   TAPE INPUT ERROR LOG   256-318

During tape I/O, these 62 bytes are used as indices to which bytes in a tape block were not received correctly during the

first pass of the data, so that on the second pass, any corrections can be made. For more detail, see the section on the cassette port in the I/O guide.

?PAGE 2

$200-$258 BUF BASIC INPUT BUFFER 512-600

In BASIC direct mode, this buffer is used to process text that is typed onto the screen. When the return key is pressed, the contents of the logical screen line which the cursor is currently on are placed into this buffer. The line is then scanned, and all recognised keywords are converted into 1 byte tokens. Finally, the first byte of the buffer is checked. If it is an ASCII number, the contents of the buffer are stored in memory as part of a BASIC program. If it is any other character, the line of text is passed to the keyword execution routine, which then performs the typed command.

This input buffer is also used for storing the text typed during the INPUT and GET commands. INPUT transfers characters from the screen on the pressing of return, and processes them via the INPUT routines. GET simply processes the first character that it finds in the buffer. As a result, INPUT and GET cannot be used in direct mode, as they would both be trying to use the same buffer for different things at the same time.

So that the processing routines know where the valid text ends, and do not try to process text from a previous line that was not overwritten, a #00 character is used as a terminator. Thus, when #00 is reached, the processing routine knows that it is time to stop and return control to the editor. It is interesting to note that although a limit of 80 characters exists on typing lines of BASIC text (88 characters for disk I/O), the maximum length of a line of BASIC text is 252 characters.

$259-$262 LAT TABLE OF ACTIVE FILE NUMBERS 601-610

This is a 10 byte table of the file numbers of currently open I/O files. All of the information needed to operate a file is held in this and the following two tables. The number of valid entries in the table is held at $98. Each time the OPEN command is used, the file number, device number and secondary address are added to this table, and $98 is incremented.

When a file is CLOSEd, $98 is decremented. If the file

being closed is not the last one in the table, then all the entries below it are moved up by 1 byte to overwrite that entry. When a file is written to, its details are read from these tables, and used to determine the current I/O device.

Since each table is 10 bytes long, only 10 logical files can be open at a time.

$263-$26C FAT TABLE OF DEV NUM FOR OPEN FILES 611-620

This is a 10 byte table containing the device numbers associated with each of the open files held in the active files table.

$26D-$276 SAT TABLE OF SEC. ADD. FOR OPEN FILES 621-630

This is a 10 byte table containing the secondary addresses associated with each of the open files held in the active files table.

$227-$280 KEYD KEYBOARD BUFFER QUEUE (FIFO) 631-640

KEYD is a 10 byte buffer that holds the ASCII value of characters typed at the keyboard. During the IRQ interrupt service routine, the keyboard is scanned, and if a key was pressed, its ASCII value is added to the end of the queue, and the buffer pointer at $C6 is incremented.

When the screen editor sees that there are characters in the keyboard buffer, they are removed, in the order they were typed in, and displayed on the screen. The maximum size of the keyboard buffer is held in $289, and can be varied from 0 to the absolute maximum of 10. The buffer does not 'wrap around' once it is full, so all characters typed when the buffer is full will be ignored.

The BASIC INPUT and GET commands transfer the contents of this buffer to the BASIC input buffer at $0200 for processing. Thus any characters already in the buffer will be treated as part of the input. There are two ways of preventing this from happening, both involving the emptying of the keyboard buffer. The first reads any characters present and discards them using a loop: FOR I=1 TO 10:GET Z$:NEXT I.
The second sets the keyboard buffer pointer to zero,

indicating no characters present: POKE 198,0.

By POKEing characters into the keyboard buffer, and setting the buffer pointer to the number of characters POKEd, a 'dynamic keyboard' effect can be achieved. Thus commands that are required to be executed in direct mode, but must be executed from within a BASIC program, can be executed in the following way.

Firstly, print the commands to be executed onto the screen, paying very careful attention to their layout and position. End the command sequence with a GOTO statement, so that the BASIC program will continue execution afterwards.

Secondly, place the required number of return characters and any other characters needed into the keyboard buffer.

Thirdly, set the buffer pointer to the number of characters in the buffer.

Finally, perform END. This activates the screen editor, and causes the contents of the keyboard buffer to be read. The direct commands will be executed, and the BASIC program will continue at the line indicated by the GOTO.

It is also possible to achieve a similar effect by directly POKEing the commands into the keyboard buffer.

$281-$282 MEMSTR POINTER: BOTTOM OF BASIC MEMORY 641-642

MEMSTR is set to point to $0800 by the hardware reset routine. It is then used by the BASIC interpreter to set its own start of BASIC pointer at ($2B). This pointer can be read or changed by using the KERNAL routine MEMBOT.

$282-$284 MEMSIZ POINTER: TOP OF BASIC MEMORY 643-644

MEMSIZ is used by the KERNAL operating system to point to the highest byte of RAM that is directly accessible to BASIC text. On power up or reset, the KERNAL routine RAMTAS is called, which performs a non-destructive test on RAM from $0400 upwards, stopping only when the test fails due to the presence of a ROM. This is normally at $A000, but can be $8000 if an extension ROM is fitted. This location is then set according to that result.

This location is changed automatically when an RS-232 file is opened, to allow space for the two 256 byte buffers that are created at the top of memory. The pointer can also be

set or read using the KERNAL routine MEMTOP.

$285	TIMOUT	IEEE TIMEOUT FLAG	645

TIMOUT is used by the operating system to signal that a timeout has occurred on the IEEE bus.

$286	COLOR	CURRENT CHARACTER COLOUR CODE	646

COLOR indicates the colour that characters printed to the screen will appear in. During the print operation, the operating system stores the printed characters in screen RAM, and stores the contents of this location in the equivalent parts of colour RAM. Thus the text appears to be printed in this colour. The colour of the characters printed can be changed in three ways.

1. Selecting a new colour directly from the keyboard. This is done by holding down the CTRL or CBM key, and pressing a number from 1 to 8. In this way, all 16 colours can be obtained.

2. Printing the CBM ASCII code for a particular colour to the screen.

3. Directly POKEing the value of the new colour into this location. The table below shows the POKE, CHR$ code and keypress for each colour.

COLOUR	POKE	CHR$	KEYBOARD	COLOUR	POKE	CHR$	KEYBOARD
BLACK	0	144	CTRL 1	ORANGE	8	129	CBM 1
WHITE	1	5	CTRL 2	BROWN	9	149	CBM 2
RED	2	28	CTRL 3	PINK	10	150	CBM 3
CYAN	3	159	CTRL 4	DK GREY	11	151	CBM 4
PURPLE	4	156	CTRL 5	MID GREY	12	152	CBM 5
GREEN	5	30	CTRL 6	LT GREEN	13	153	CBM 6
BLUE	6	31	CTRL 7	LT BLUE	14	154	CBM 7
YELLOW	7	158	CTRL 8	LT GREY	15	155	CBM 8

$287	GDCOL	CHARACTER COLOUR UNDER CURSOR	647

GDCOL holds the colour code of the character currently under the cursor. This is because the cursor blinks in the current character colour, which may not be the same as the colour of the character on the screen. Thus, when the cursor moves on, the character is returned to its original colour.

$288 HIBASE TOP OF SCREEN MEMORY (PAGE) 648

HIBASE indicates the logical page on which the 40 by 25 video screen matrix starts. On power up, this is set to the value 4, indicating that the screen starts at location $0400. The purpose of this location is to tell the screen editor where in memory to store printed characters.

The screen display can be moved around in memory by changing the VIC II control register ($D018) and the VIC II memory bank select register ($DD00). The only limitation on moving the screen is that it must always start on a 1K boundary ($0400, $0800, $0C00 etc.). The screen editor will not be able to print to this new screen until HIBASE is altered to point to it.

HIBASE can be used to print characters to other areas of memory that are not to be directly displayed. For example, by setting HIBASE to point to the start of a block of sprite data, the data can be printed as a string straight into memory, instead of a whole series of POKEs.

$289 XMAX SIZE OF KEYBOARD BUFFER 649

XMAX indicates the maximum number of characters that can be held in the keyboard buffer (located at $0277). The absolute maximum size of the keyboard buffer is 10 characters, but the only restraint on extending the buffer beyond this value is the fact that it would cause the buffer to overwrite the pointers to the top and bottom of memory and screen. If needed, the pointers can be saved and restored by using KERNAL routines.

Note that when the keyboard buffer pointer reaches the same value as is stored here, all keypresses will be ignored, until a character has been read from the buffer.

$28A RPTFLG FLAG: REPEAT KEYPRESS 650

RPTFLG is used by the keyboard scan routine to determine what keys, if any, should repeat. When a key is repeated, its ASCII value is written into the keyboard buffer continuously, at a rate determined by KOUNT and DELAY, until the key is released. If a key is not repeated, it is placed into the keyboard buffer once and then ignored until the key is released.

The default value in this location is 0, which repeats only the space bar, cursor and INST/DEL keys. A value of #80

will cause all keys to repeat, and a value of #40 will prevent any keys from repeating. Other values will cause different keys to repeat.

$28B KOUNT REPEAT SPEED COUNTER 651

KOUNT is a counter, used by the keyboard reading routine to determine how long to wait between placing each successive repeat 'keypress' into the keyboard buffer. The IRQ service routine initially sets this location to 6, and, once $28C indicates that the key should repeat, it is decremented once every 60th of a second. When it reaches zero, it is set to 4, and the process repeated. The rate of repeat on each key is 15 repeats per second.

$28C DELAY REPEAT DELAY COUNTER 652

DELAY is used by the keyboard scan routine to determine how long a key must be held down before it is repeated. It is set to #10, and, once a key is pressed, decremented every 60th of a second. Once this counter reaches zero, the counter at $028B, to time the delay between repeats is enabled. The initial value is only restored once the key has been released. The delay between pressing the key and it repeating is approximately 1/3 of a second.

$28D SHFLG FLAG SHIFT / CTRL / CBM KEY 653

SHFLG holds a flag to indicate to the operating system which, if any, of the SHIFT, CTRL or CBM logo keys are being pressed. The value 1 indicates SHIFT, while the values 2 and 4 indicate CBM and CTRL respectively. The values are cumulative when more than one key is being held down.

SHFLG is used by two routines. Firstly, by the keyboard decode routine, to indicate which of the four keyboard ASCII lookup tables will be used, and secondly, by the screen editor, which uses SHIFT/CBM to toggle between the upper case/graphics and upper/lower case character sets. This last use is completely separate from the keyboard SHIFT decoding, and only changes the part of the character ROM that is displayed on the screen.

$28E LSTSHF LAST KEYBOARD SHIFT PATTERN 654

LSTSHF is used by the keyboard scan routine in conjunction with SHFLG ($028D) to debounce the SHIFT, CTRL and CBM keys. It prevents the screen editor from repeatedly toggling the upper/lower case character sets during one pressing of the

SHIFT and CBM keys.

$28F-$290 KEYLOG VECTOR: KEYBOARD TABLE SETUP 655-656

KEYLOG is a pointer to the keyboard decode routine, which takes the value of the keypress from the keyboard scan routine and converts it into a CBM ASCII character. This involves using one of four keyboard lookup tables (one for straight keypresses, and one each for SHIFT, CTRL and CBM). The routine is situated at $EADD.

$291 MODE FLAG: ENABLE SHIFT/CBM 657

The MODE flag enables or disables the screen character set toggling feature of the SHIFT and CBM keys. By setting this location to #00, the feature is disabled. Setting the value #80 will enable the feature. This flag does not affect the normal operation of either the SHIFT or CBM key when used on its own. The effect of the flag is identical to PRINTing CHR$(8) or CHR$(9).

$292 AUTODN FLAG: AUTO SCROLL DOWN ENABLED 658

AUTODN is used as a flag by the screen editor to determine whether or not moving the cursor beyond the 40th column of a screen line will cause another physical line to be added to the logical line. When this location is set to #00, the lines below the current one will be scrolled down in order to add the new physical line. Any non-zero value will prevent the scroll.

$293 M51CTR RS-232 6551 CONT REGISTER IMAGE 659

M51CTR is used to control the baud rate, word length and the number of stop bits applied to characters being transmitted on the RS-232 port. The format used is identical to that of the 6551 ACIA chip (table 2.1). When a file is opened to the RS-232 port, the first character of the file name is stored here. See also the section on RS-232 in the I/O ports guide.

$294 M51CDR RS-232 6551 COMMAND REG IMAGE 660

M51CDR is used to control the parity, duplex mode and the handshaking protocol used on the RS-232 port. The format

used is identical to that of the 6551 ACIA chip (table 3.1). When a file is opened to the RS-232 port, the second character of the file name is stored here. See also the section on RS-232 in the I/O ports guide.

$295-$296 M51AJB RS-232 NON-STANDARD BIT TIME 661-662

M51AJB is used to store the non-standard baud rate when this option is selected in $0293. This was probably put here to conform to the 6551 ACIA, however the non-standard bit timing is not implemented in the Commodore software version of the device. Commodore have specified that the value stored here should be the system o2 clock frequency, divided by 2, and minus 100. This result is stored in the lo-hi format. o2 clock rates are, for NTSC systems, 1.02273 MHz, and, for PAL systems, 0.98525 MHz.

$297 RSSTAT RS-232 6551 STATUS REG IMAGE 663

RSSTAT is used to indicate the current error status of the RS-232 port. Apart from direct PEEKing, this location can be read by calling the KERNAL routine READST, or by referring to the BASIC reserved variable, ST. Both READST and ST reset this location to zero after use, so it is important to preserve the original value if more than one test is to be made. The detail of the status register is set out in table 2.1. It is up to the user to take any action on the detection of an error in data transmission, since no automatic action is taken by the computer.

$298 BITNUM RS-232 No OF BITS STILL TO SEND 664

BITNUM holds the number of bits of the current byte that have not been transmitted onto the RS-232 port.

$299-$29A BAUDOF RS-232 BAUD RATE (BIT TIME) 665-666

BAUDOF holds the baud rate (number of bits sent per second) currently being used on the RS-232 port. The value here is used as a basis for setting the two timers on the CIA#2 chip. When these timers reach zero, an NMI interrupt is generated. The NMI service routine then handles data transmission to and from the port. The actual value (called a prescaler) that is stored in the CIA timers can be

calculated as follows. PRESCALER = ((o2 CLOCK / BAUD RATE) / 2) - 100
The o2 system clock frequency is 1.02273 MHz for NTSC systems, and 0.98525 MHz for PAL systems. A table of prescaler values for the valid baud rates is held at $FEC2 for the American NTSC version, and $E4EC for the European PAL version.

CONTROL REGISTER

7	6 5	4	3 2 1 0
STOP BITS	**WORD LENGTH**	**UNUSED**	**BAUD RATE**
0 = 1 bit	00 = 8 bits		0000 = USER RATE (NI)
1 = 2 bits	01 = 7 bits		0001 = 50 BAUD
	10 = 6 bits		0010 = 75
	11 = 5 bits		0011 = 110
			0100 = 134.5
			0101 = 150
			0110 = 300
			0111 = 600
			1000 = 1200
			1001 = (1800) 2400
			1010 = 2400
			1011 = 3600 (NI)
			1100 = 4800 (NI)
			1101 = 7200 (NI)
			1110 = 9600 (NI)
			1111 = 19200 (NI)

NI = Not Implemented

COMMAND REGISTER

7 6 5	4	3 2 1	0
PARITY OPTIONS	**DUPLEX**	**UNUSED**	**HAND-SHAKE**
XX0 parity disabled — none generated / received	0 = FULL		0=3 LINE
001 Odd parity Receiver/transmitter	1 = HALF		1=X LINE
011 Even parity Receiver/transmitter			
101 Mark transmitted Parity check disabled			
111 Space transmitted Parity check disabled			

STATUS REGISTER

BIT
0 = PARITY ERROR
1 = FRAMING ERROR
2 = RECEIVER BUFFER OVERRUN
3 = RECEIVER BUFFER EMPTY
4 = CTS SIGNAL MISSING
5 = UNUSED
6 = DSR SIGNAL MISSING
7 = BREAK DETECTED

TABLE 2.1

$29B RIDBE RS-232 INDEX TO END OF IN BUFFER 667

RIDBE points to the final byte of the RS-232 input buffer. Both RS-232 buffers operate on a wrap-around principle, ie. once data reaches one end of the buffer, it starts to write again from the other end. Thus, the start and end points may be anywhere within the 256 byte buffer.

$29C RIDBS RS-232 START PAGE OF IN BUFFER 668

This location is an index to the start of the RS-232 input buffer, and is used as a pointer for reading data from the buffer. Both RS-232 buffers operate on a wrap-around principle, ie. once data reaches one end of the buffer, it starts to write again from the other end. Thus, the start and end points may be anywhere within the 256 byte buffer.

$29D RODBS RS-232 START PAGE OF OUT BUFFER 669

RODBS is an index to the start of the RS-232 output buffer, and is used as a pointer for reading data from the buffer. Both RS-232 buffers operate on a wrap-around principle, ie. once data reaches one end of the buffer, it starts to write again from the other end. Thus, the start and end points may be anywhere within the 256 byte buffer.

$29B RODBE RS-232 INDEX TO END OF OUT BUFFER 670

RODBE points to the final byte of the RS-232 output buffer. It is used as a pointer for writing data to the buffer. Both RS-232 buffers operate on a wrap-around principle, ie. once data reaches one end of the buffer, it starts to write again from the other end. Thus, the start and end points may be anywhere within the 256 byte buffer.

$29F-$2A0 IRQTMP IRQ VEC STORE DURING TAPE I/O 671-672

When data is read or written to tape, the normal IRQ interrupt vector at ($0314) is replaced with a series of special vectors that point to the tape I/O routines. IRQTMP is used to store the normal vector during this time, so that once tape I/O has finished, the normal vector can be restored.

The vector is saved and replaced in this way, instead of just copying it back from the ROM vectors list, so that any user IRQ vector (for, say, IRQ driven music or sprites) will be preserved. When tape I/O is taking place, all the normal functions performed by the IRQ service routine (reading the keyboard, updating the clock etc.) are suspended.

$2A1 ENABL RS-232 ENABLES (NMI INTERRUPT CONTROL) 673

ENABL holds the value which is written into the CIA#2 Interrupt Control register (ICR) at $DD0D. It is used as an indicator to what the system is currently doing. Three bits are significant here, and all are active 1.

BIT 0 - SYSTEM IS TRANSMITTING DATA
BIT 1 - SYSTEM IS RECEIVING DATA
BIT 4 - SYSTEM IS WAITING FOR RECEIVER EDGE

$2A2 TOD SENSE DURING TAPE I/O 674

This location is used as a store of the CIA#1 Control Register B during tape I/O.

$2A3 TEMP STORE FOR TAPE READ 675

This location is used as a store of the CIA#1 Control Register A during tape I/O.

$2A4 TEMP D1IRQ INDICATOR FOR TAPE READ 676

This location is used as a store of the CIA#1 Interrupt Control Register during tape I/O.

$2A5 TEMP SCREEN LINE INDEX 677

This is a temporary pointer, used by the screen editor to indicate the start of the next 40 column screen line during screen scrolling.

$2A6 PAL / NTSC FLAG 678

This location is set to 0 if the computer is an NTSC

(American TV) standard, and 1 if it is PAL (European TV) standard.

A test is performed on power up to determine the type used. This consists of setting the Raster Interrupt flag for raster line 311. If the interrupt occurs, then the computer is a PAL system (NTSC monitors only have 262 raster scan lines).

The difference between PAL and NTSC is important, because they both operate on a different system o2 clock frequency (NTSC = 1.02273 MHz, PAL = 0.98525 MHz). This clock frequency is used as a basis for generating the IRQ interrupts every 60th of a second, and also for the RS-232 baud rate timing. A prescaler is used to offset the differences in frequency when calculating the values to use in the timers for these functions.

$2A7-$2FF SPRT11 UNUSED (SPRITE BLOCK 11) 679-767

This area is unused by both BASIC and the KERNAL operating system. It can therefore be used by the programmer for short machine language programs, or for storing sprite data. When being used for sprite data, this area is designated as block 11.

$300-$301 IERROR VEC: PRINT BASIC ERROR MESSAGE 768-769

IERROR points to the start of the BASIC error handling routine at $E38B. In order to generate a particular error message, the (X) register must be loaded with the error code, and this routine called. It is important to note that once the message has been printed, a BASIC warm start is performed. A table of error codes and their related messages is given with the notes on this routine in the ROM GUIDE.

$302-$303 IMAIN VECTOR: BASIC WARM START 770-771

IMAIN points to the start of the main BASIC input, identify and execute loop at $A483. This routine performs all the BASIC functions in direct mode.

$304-$305 ICRNCH VECTOR: TOKENISE BASIC TEXT 772-773

ICRNCH points to the routine at $A57C which takes a line of BASIC text from the system input buffer at $0200 and converts it into 1 byte tokens.

$306-$307 IQPLOP VECTOR: LIST BASIC TEXT 774-775

IQPLOP points to the routine at $A7A1 which converts BASIC keyword tokens into their full length keywords, and lists the current BASIC program to the screen.

$308-$309 IGONE VECTOR: BASIC CHARACTER DISPATCH 776-777

IGONE points to the routine at $A7E4 which executes a BASIC keyword token.

$30A-$30B IEVAL VECTOR: EVALUATE BASIC TOKEN 778-779

IEVAL points to the routine at $AE86 which evaluates a single term of an arithmetic expression.

$30C SAREG (A) REGISTER FOR SYS 780

SAREG can be used to set the processor accumulator to a predetermined value on entry to a SYS call. Once the machine language routine has finished, the final value of (A) is returned here.

$30D SXREG (X) REGISTER FOR SYS 781

SXREG can be used to set the processor X index register to a predetermined value on entry to a SYS call. Once the machine language routine has finished, the final value of (X) is returned here.

$30E SYREG (Y) REGISTER FOR SYS 782

SYREG can be used to set the processor Y index register to a predetermined value on entry to a SYS call. Once the machine language routine has finished, the final value of (X) is returned here.

$30F SPREG (P) REGISTER FOR SYS 783

SPREG can be used to set the processor status register to a predetermined value on entry to a SYS call. Once the machine language routine has finished, the final value of (P) is returned here.

$310 USRPOK JMP INSTRUCTION FOR USR FUNCTION 784

USRPOK holds the 6510 machine language op-code for JMP (#4C). It is used in conjunction with the following two bytes, which form the operand, to execute a USR function machine language routine.

$311-$312 USRADD LO-HI ADDRESS FOR USR ROUTINE 785-786

USRADD contains the target address of a machine language routine for use with the USR function. The programmer must set up the address here before calling the function, since the default address generates an ?ILLEGAL QUANTITY error.

The parameter in parentheses in the X = USR(Y) statement is placed into the flpt accumulator fac#1 ($61-$66). At the end of the routine, the contents of fac#1 are assigned to the preceeding variable (X in this case).

$313 UNUSED 787

$314-$315 CINV VECTOR: HARDWARE IRQ INTERRUPT 788-789

CINV points to the routine at $EA31 which services and IRQ interrupt request. Normally an IRQ is generated 60 times each second by timer B of CIA#1. During the servicing of each IRQ, the system jiffy clock is updated, the keyboard is scanned, the cursor is blinked, the tape motor interlock is

maintained and the STOP key tested for.

By changing this vector, it is possible to substitute another machine language routine to be executed every 60th of a second. This routine must either perform the normal IRQ service functions itself, or call the normal IRQ routine once it has finished. Several factors must be borne in mind when changing the IRQ vector.

An IRQ request may occur while the vector is being changed. This would cause a fatal error, and recovery could only be made by resetting the machine. This can be overcome by using the 6510 SEI instruction before changing the vector. This has the effect of preventing an IRQ from being serviced. Once the vector has been changed, the IRQ can be enabled again with the CLI instruction.

It is possible for an IRQ to be generated from sources other than CIA#1 timer B. There is the remainder of CIA#1, and the VIC II chip to take into consideration when handling an IRQ. Thus the source of the IRQ should be established before it is processed.

The 6510 BRK instruction also generates an IRQ. This is tested for by the ROM IRQ routine before being directed to this vector. A BRK IRQ is directed to the vector CBINV at ($316).

$316-$317 CBINV VECTOR: SOFTWARE BRK INTERRUPT 790-791

CBINV points to the routine which is executed every time the 6510 BRK (#00) instruction is encountered. This defaults to the BASIC warm start routine called by pressing RUN/STOP and RESTORE. This vector is often used by machine language monitors to call the monitor warm start routine.

$318-$319 NMINV VECTOR: HARDWARE NMI INTERRUPT 792-793

NMINV points to the routine at $FE47 which is executed whenever an NMI interrupt request is generated. There are two sources of an NMI interrupt, and the routine pointed to by this vector checks to see which of them caused it, and acts accordingly.

1. CIA#2. If the NMI was generated here, the routine checks to see if one of the RS-232 routines should be called.

2. The RESTORE key is connected directly to the processor

NMI line. When the NMI comes from this source, the STOP key is checked. If it was pressed simultaneously with RESTORE, a BASIC warm start is performed, unless an external ROM cartridge is present, in which case that is warm started. If the STOP key was not pressed, the NMI routine exits without taking any further action.

Note that by changing this vector to point to the RTI instruction at the end of the service routine, the STOP/RESTORE keys can be disabled. This, however does have the effect of disabling all NMIs.

$31A-$31B IOPEN VECTOR: KERNAL OPEN ROUTINE 794-795

IOPEN points to the KERNAL OPEN routine located at $F34A. The OPEN entry at $FFC0 in the KERNAL jump table is directed here.

$31C-$31D ICLOSE VECTOR: KERNAL CLOSE ROUTINE 796-797

ICLOSE points to the KERNAL CLOSE routine located at $F291. The CLOSE entry at $FFC3 in the KERNAL jump table is directed here.

$31E-$31F ICHKIN VECTOR: KERNAL CHKIN ROUTINE 798-799

ICHKIN points to the KERNAL CHKIN routine located at $F20E. The CHKIN entry at $FFC6 in the KERNAL jump table is directed here. CHKIN is used to open a channel for input from a device.

$320-$321 ICKOUT VECTOR: KERNAL CHKOUT ROUTINE 800-801

ICKOUT points to the KERNAL CHKOUT routine located at $F250. The CHKOUT entry at $FFC9 in the KERNAL jump table is directed here. CHKOUT is used to open a channel for output to a device.

$322-$323 ICLRCH VECTOR: KERNAL CLRCHN ROUTINE 802-803

ICLRCH points to the KERNAL CLRCHN routine located at $F333. The CLRCHN entry at $FFCC in the KERNAL jump table is directed here. CLRCHN is used to close all currentluy open channels.

$324-$325 IBASIN VECTOR: KERNAL CHRIN ROUTINE 804-805

IBASIN points to the KERNAL CHRIN routine located at $F157. The CHRIN entry at $FFCF in the KERNAL jump table is

directed here. CHRIN is used to input a character from a channel opened by CHKIN.

$326-$327 IBSOUT VECTOR: KERNAL CHROUT ROUTINE 806-807

IBSOUT points to the KERNAL CHROUT routine located at $F1CA. The CHROUT entry at $FFD2 in the KERNAL jump table is directed here. CHROUT is used to output a character to a channel opened by CHKOUT.

$328-$329 ISTOP VECTOR: KERNAL STOP ROUTINE 808-809

ISTOP points to the KERNAL STOP routine located at $F6ED. The STOP entry at $FFE1 in the KERNAL jump table is directed here. STOP is used to scan the <STOP> key. This routine can be disabled by incrementing the vector by three bytes, ie. POKE 808,239. Note that this does not stop the STOP/RESTORE sequence. POKE 808,234 will cause the BASIC LIST function to be disabled. Both instances can be returned to normal by POKE 808,237.

$32A-$32B IGETIN VECTOR: KERNAL GETIN ROUTINE 810-811

IGETIN points to the KERNAL GETIN routine located at $F13E. The GETIN entry at $FFE4 in the KERNAL jump table is directed here. GETIN is used to get a character from a currently open file.

$32C-$32D ICLALL VECTOR: KERNAL CLALL ROUTINE 812-813

ICLALL points to the KERNAL CLALL routine located at $F32F. The CLALL entry at $FFE7 in the KERNAL jump table is directed here. CLALL is used to close all open channels and files.

$32E-$32F USRCMD USER-DEFINED VECTOR 814-815

USRCMD points to the software BRK routine located at $FE66. This vector is used by machine language monitors (MLMs) when they encounter a command that they don't understand. It can thus be used to add new commands to an MLM. Since there is no MLM built into the Commodore 64, this vector is mostly redundant.

$330-$331 ILOAD VECTOR: KERNAL LOAD ROUTINE 816-817

ILOAD points to the KERNAL LOAD routine located at $F49E. The LOAD entry at $FFD5 in the KERNAL jump table is directed here. LOAD is used to load or verify a file into RAM.

$330-$331 ISAVE VECTOR: KERNAL SAVE ROUTINE

ISAVE points to the KERNAL SAVE routine located at $F5DD. The SAVE entry at $FFD8 in the KERNAL jump table is directed here. SAVE is used to save RAM to an external device.

$334-$338 UNUSED 820-827

$33C-$3FB TBUFFR CASSETTE I/O BUFFER 828-1019

TBUFFR is the 192 byte buffer used for reading and writing data to tape. When LOADing or SAVEing programs, the buffer is only used for the tape header block, the rest of the I/O directly accessing the RAM concerned. When accessing data files, the data is read and written from the buffer.

The format of a tape header block is as follows. Byte 1 is a header block type id. This indicates to the operating system which type of file is currently being accessed. The header types are shown below.

#01 - RELOCATABLE PROGRAM FILE (SA = 0)
#02 - DATA FILE (ACTUAL DATA BLOCK)
#03 - ABSOLUTE PROGRAM FILE (SA = 1)
#04 - DATA FILE HEADER
#05 - END-OF-TAPE HEADER

The second and third bytes of the header indicate the file start address. This is the start address of the program for program files, and the start of the data within the buffer for data files.

The next two bytes of the header indicate the file end address. This is the end address of the program for program files, and the end of the data within the buffer for data files.

The remainder of the header is used to store either the actual data bytes, or the file name. The file name can be up to 187 bytes long, and can include things like sprite data and short machine language programs that need to run in the cassette buffer.

When the buffer is not being used for tape I/O, it is possible to use it for storing data for sprite blocks 13,14 and 15. It is also a favourite place for storing short machine language programs.

$3FC-$3FF UNUSED 1020-1023

PAGE 4 UPWARDS

$400-$7FF VICSCN VIDEO MATRIX: 25 LNS * 40 COL 1024-2047

VICSCN holds the video screen matrix and the sprite data pointers. Screen memory can be relocated to start at the beginning of any 2K boundary by setting the VIC II memory control registers at $D018 and $DD00.

The video matrix is used to store characters for display on the screen. Each byte stored here represents a number, letter or graphical symbol to be displayed on the screen (A=#01, B=#02 etc.). The VIC-II chip takes these screen codes and uses them as an index to the position of the character in the character dot ROM.

Characters may be stored here either by printing them through the BASIC PRINT statement and the KERNAL CHROUT routines, or by directly POKEing the character screen codes into screen memory. This second method requires a value to be POKEd into the equivalent byte of colour RAM ($D800-$D8FF), since the character colour is set to the background colour whenever the screen is cleared.

The final eight bytes of the video area are used as sprite data pointers. These locations point to the start of the 63 byte block of data that defines a particular sprite. The final byte of each sprite block is unused, so that they can be divided into convenient blocks of 64 bytes.

$800-$9FFF BASIC RAM PROGRAM AREA 2048-40959

This is the area where BASIC programs and variables are stored. A BASIC program is made up of linked lines of tokenised BASIC keywords. The format of each BASIC line is as follows:

1. A two byte lo-hi link address, pointing to the start of the next BASIC line. In the final line of the program, this link address is set to $0000.

2. A two byte lo-hi line number (lo-hi). This is the BASIC line number
given the line by the programmer. It must be in the range 0 to 63999.

3. The tokenised BASIC keywords and data. This section can be up to 250 bytes long, although only 80 bytes can be written directly from the keyboard.

4. The line terminator character (#00). This informs the BASIC interpreter that the end of the line has been reached. The very first line of BASIC must also be #00, as this is treated by the interpreter as an end of line.

$1000-$1FFF BANK 0 CHARACTER ROM IMAGE 4096-8191

This is the default character ROM image that is seen by the VIC II chip when looking at bank 0 (the 1st 16K of RAM). The actual character ROM is located at $C000, but due to the construction of the machine, the VIC II chip can see it here. The ROM is used for holding the dot images of all the characters that can be displayed on the screen. There are two complete 256 character character sets. Both sets can be toggled between by pressing the <SHIFT> and <CBM> keys together.

SECTION 3.
ROM GUIDE

CHRGET AND CHRGOT

This is probably the most important part of the BASIC interpreter. When the computer is switched on, it is copied from ROM into RAM at $0073 to $008A. The purpose of CHRGET is to find the next byte of BASIC text from either the input buffer ($0200-$0258) or from BASIC text ($0800-$9FFF).

The index TXTPTR ($7A) is in the middle of the routine. This points to the current byte of text and is modified by the routine as it is executed. This is the reason it is in RAM and not ROM.

On exit, the byte of text is held in (A) and the carry flag is set according to the ASCII code of the character: If the ASCII code is between #30 and #39, (ie a decimal number), then carry is clear, otherwise it is set. If the Zero flag is set, then a terminator has been found - either End-Of-Line or colon.

A popular way of adding new commands to BASIC is to modify CHRGET by using a 'wedge' to point to a user replacement routine. An example of this is the DOS support program supplied by Commodore.

```
., 0073  e6 7a     inc $7a      ;increment TXTPTR
., 0075  d0 02     bne $0079
., 0077  e6 7b     inc $7b
., 0079  ad 08 02  lda $0208    ;CHRGOT entry. read TXTPTR
., 007c  c9 3a     cmp #$3a     ;ASCII colon (terminator) -
                                sets Z flag
., 007e  b0 0a     bcs $008a
., 0080  c9 20     cmp #$20     ;ASCII space - get next
                                character
., 0082  f0 ef     beq $0073
., 0085  38        sec
., 0085  e9 30     sbc #$30     ;ASCII zero
., 0087  38        sec
., 0088  e9 d0     sbc #$d0
., 008a  60        rts
```

40960 RESTART VECTORS

These are vectors to cold ($E394) and warm (E37b) reset routines, and the ASCII string 'CBMBASIC'. The cold restart vector is used to initialise BASIC when the computer is first switched on, and the warm restart vector is used when <STOP/RESTORE> is pressed.

```
.:a000 94 e3 7b e3 43 42 4d 42
.:a008 41 53 49 43
```

40972 STMDSP: BASIC COMMAND VECTORS

These are vectors to the routines indicated by the BASIC keyword table. The vector is pushed onto the stack and RTS performed via the CHRGET routine. As a result, the vector points to the start address
-1 of the routine.

```
.:a00c 30 a8 41 a7 1d ad f7 a8
.:a014 a4 ab be ab 80 b0 05 ac
.:a01c a4 a9 9f a8 70 a8 27 a9
.:a024 1c a8 82 a8 d1 a8 3a a9
.:a02c 2e a8 4a a9 2c b8 67 e1
.:a034 55 e1 64 e1 b2 b3 23 b8
.:a03c 7f aa 9f aa 56 a8 9b a6
.:a044 5d a6 85 aa 29 e1 bd e1
.:a04c c6 e1 7a ab 41 a6
```

41042 FUNDSP: BASIC FUNCTION VECTORS

These are vectors to the functions indicated by the BASIC keyword table. Functions are distinguished from commands and operators by a following argument in parentheses ().

```
.:a052 39 bc cc bc 58 bc 10 03
.:a05a 7d b3 9e b3 71 bf 97 e0
.:a062 ea b9 ed bf 64 e2 6b e2
.:a06a b4 e2 0e e3 0d b8 7c b7
.:a072 65 b4 ad b7 8b b7 ec b6
.:a07a 00 b7 2c b7 37 b7
```

41088 OPTAB: BASIC OPERATOR VECTORS

Here each operator vector is preceeded by a priority code. This is used to determine the order in which operators are performed. The vectors point to the start address of the routine -1. When the operator is called, its vector is

pushed onto the stack and RTS performed from the CHRGET routine.

```
.:a080 79 69 b8 79 52 b8 7b 2a
.:a088 ba 7b 11 bb 7f 7a bf 50
.:a090 e8 af 46 e5 af 7d b3 bf
,:a098 5a d3 ae 64 15 b0
```

41118 RESLIST: BASIC COMMAND KEYWORD TABLE

This is a table of BASIC keywords. The keywords are written in token order, ie. END = #80, FOR = #81 etc. To calculate the token value for a keyword, add 127 to its position in this table. Each keyword has bit 7 of the last character set to 1. The end of the table is denoted by a zero byte. This table is used by the routines to convert BASIC text into compact 1 byte tokens, and also by the LIST routine to expand the tokenised text. The keyword GO (#CB) is included to allow the use of GO TO as well as GOTO. Note that there is no separate token for GET#.

```
.:a09e 45 4e c4 46 4f d2 4e 45  enDfoRne
.:a0a6 58 d4 44 41 54 c1 49 4e  xTdatAin
.:a0ae 50 55 54 a3 49 4e 50 55  put#inpu
.:a0b6 d4 44 49 cd 52 45 41 c4  TdiMreaD
.:a0be 4c 45 d4 47 4f 54 cf 52  leTgotOr
.:a0c6 55 ce 49 c6 52 45 53 54  uNiFrest
.:a0ce 4f 52 c5 47 4f 53 55 c2  orEgosuB
.:a0d6 52 45 54 55 52 ce 52 45  returNre
.:a0de cd 53 54 4f d0 4f ce 57  MstoPoNw
.:a0e6 41 49 d4 4c 4f 41 c4 53  aiTloaDs
.:a0ee 41 56 c5 56 45 52 49 46  avEverif
.:a0f6 d9 44 45 c6 50 4f 4b c5  YdeFpokE
.:a0fe 50 52 49 4e 54 a3 50 52  print#pr
.:a106 49 4e d4 43 4f 4e d4 4c  inTconTl
.:a10e 49 53 d4 43 4c d2 43 4d  isTclRcm
.:a116 c4 53 59 d3 4f 50 45 ce  DsySopeN
.:a11e 43 4c 4f 53 c5 47 45 d4  closEgeT
.:a126 4e 45 d7 54 41 42 a8 54  neWtab(t
.:a12e cf 46 ce 53 50 43 a8 54  OfNspc(t
.:a136 48 45 ce 4e 4f d4 53 54  heNnoTst
.:a13e 45 d0 ab ad aa af de 41  ePanDoRs
.:a146 4e c4 4f d2 be bd bc 53  gNinTabS
.:a14e 47 ce 49 4e d4 41 42 d3  usRfrEpo
.:a156 55 53 d2 46 52 c5 50 4f  SsqRrnDl
.:a15e d3 53 51 d2 52 4e c4 4c  oGexPcoS
.:a166 4f c7 45 58 d0 43 4f d3  siNtaNat
.:a16e 53 49 ce 54 41 ce 41 54  NpeeKleN
.:a176 ce 50 45 45 cb 4c 45 ce  str$vaLa
```

```
.:a17e 53 54 52 a4 56 41 cc 41  sCchr$le
.:a186 53 c3 43 48 52 a4 4c 45  ft$right
.:a18e 46 54 a4 52 49 47 48 54  $mid$gO
.:a196 a4 4d 49 44 a4 47 cf 00
```

41374 ERRTAB: ERROR MESSAGE TABLE

This is the table of error messages used by BASIC. Each message has bit 7 of the last character set to 1.

```
.:a19e 54 4f 4f 20 4d 41 4e 59  too many
.:a1a6 20 46 49 4c 45 d3 46 49   fileSfi
.:a1ae 4c 45 20 4f 50 45 ce 46  le opeNf
.:a1b6 49 4c 45 20 4e 4f 54 20  ile not
.:a1be 4f 50 45 ce 46 49 4c 45  opeNfile
.:a1c6 20 4e 4f 54 20 46 4f 55   not fou
.:a1ce 4e c4 44 45 56 49 43 45  nDdevice
.:a1d6 20 4e 4f 54 20 50 52 45   not pre
.:a1de 53 45 4e d4 4e 4f 54 20  senTnot
.:a1e6 49 4e 50 55 54 20 46 49  input fi
.:a1ee 4c c5 4e 4f 54 20 4f 55  lEnot ou
.:a1f6 54 50 55 54 20 46 49 4c  tput fil
.:a1fe c5 4d 49 53 53 49 4e 47  Emissing
.:a206 20 46 49 4c 45 20 4e 41   file na
.:a20e 4d c5 49 4c 4c 45 47 41  mEillega
.:a216 4c 20 44 45 56 49 43 45  l device
.:a21e 20 4e 55 4d 42 45 d2 4e   numbeRn
.:a226 45 58 54 20 57 49 54 48  ext with
.:a22e 4f 55 54 20 46 4f d2 53  out foRs
.:a236 59 4e 54 41 d8 52 45 54  yntaXret
.:a23e 55 52 4e 20 57 49 54 48  urn with
.:a246 4f 55 54 20 47 4f 53 55  out gosu
.:a24e c2 4f 55 54 20 4f 46 20  Bout of
.:a256 44 41 54 c1 49 4c 4c 45  datAille
.:a25e 47 41 4c 20 51 55 41 4e  gal quan
.:a266 54 49 54 d9 4f 56 45 52  titYover
.:a26e 46 4c 4f d7 4f 55 54 20  floWout
.:a276 4f 46 20 4d 45 4d 4f 52  of memor
.:a27e d9 55 4e 44 45 46 27 44  Yundef'd
.:a286 20 53 54 41 54 45 4d 45   stateme
.:a28e 4e d4 42 41 44 20 53 55  nTbad su
.:a296 42 53 43 52 49 50 d4 52  bscripTr
.:a29e 45 44 49 4d 27 44 20 41  edim'd a
.:a2a6 52 52 41 d9 44 49 56 49  rraYdivi
.:a2ae 53 49 4f 4e 20 42 59 20  sion by
.:a2b6 5a 45 52 cf 49 4c 4c 45  zerOille
.:a2be 47 41 4c 20 44 49 52 45  gal dire
.:a2c6 43 d4 54 59 50 45 20 4d  cTtype m
.:a2ce 49 53 4d 41 54 43 c8 53  ismatcHs
```

```
.:a2d6 54 52 49 4e 47 20 54 4f  tring to
.:a2de 4f 20 4c 4f 4e c7 46 49  o lonGfi
.:a2e6 4c 45 20 44 41 54 c1 46  le datAf
.:a2ee 4f 52 4d 55 4c 41 20 54  ormula t
.:a2f6 4f 4f 20 43 4f 4d 50 4c  oo compl
.:a2fe 45 d8 43 41 4e 27 54 20  eXcan't
.:a306 43 4f 4e 54 49 4e 55 c5  continuE
.:a30e 55 4e 44 45 46 27 44 20  undef'd
.:a316 46 55 4e 43 54 49 4f ce  functioN
.:a31e 56 45 52 49 46 d9 4c 4f  verifYlo
.:a326 41 c4                    aD
```

41768 ERRPTR: ERROR MESSAGE POINTERS

This is a table of vectors to the start address of each message in the error messages table. There are 30 error messages including BREAK, which is in the table below.

```
.:a328 9e a1 ac a1 b5 a1 c2 a1
.:a330 d0 a1 e2 a1 f0 a1 ff a1
.:a338 10 a2 25 a2 35 a2 3b a2
.:a340 4f a2 5a a2 6a a2 72 a2
.:a348 7f a2 90 a2 9d a2 aa a2
.:a350 ba a2 c8 a2 d5 a2 e4 a2
.:a358 ed a2 00 a3 0e a3 1e a3
.:a360 24 a3 83 a3
```

41828 OKK: MISC MESSAGES

This is a table of miscellaneous messages used by BASIC. Each message in the table has a zero byte terminator.

```
.:a364 0d 4f 4b 0d 00 20 20 45   ok    e
.:a36c 52 52 4f 52 00 20 49 4e  rror  in
.:a374 20 00 0d 0a 52 45 41 44      read
.:a37c 59 2e 0d 0a 00 0d 0a 42  y.     b
.:a384 52 45 41 4b 00           reak
```

41866 FNDFOR: FIND FOR/GOSUB ENTRY ON STACK

This routine is called by NEXT and RETURN. The stack is searched for a FOR token. If not found then ?NEXT WITHOUT FOR, unless called from RETURN when further tests are performed by that routine.

```
., a38a  ba       tsx
., a38b  e8       inx
., a38c  e8       inx
., a38d  e8       inx
```

```
., a38e  e8        inx          ;(X) points to where token
                                should be
., a38f  bd 01 01  lda $0101,x
., a392  c9 81     cmp #$81     ;token FOR
., a394  d0 21     bne $a3b7    ;no - RTS
., a396  a5 4a     lda $4a      ;>FORPNT - pointer to index
                                variable
., a398  d0 0a     bne $a3a4    ;index exists
., a39a  bd 02 01  lda $0102,x
., a39d  85 49     sta $49      ;recover FORPNT from stack
., a39f  bd 03 01  lda $0103,x
., a3a2  85 4a     sta $4a
., a3a4  dd 03 01  cmp $0103,x  ;check same index variable
., a3a7  d0 07     bne $a3b0    ;if different then check next
                                entry
., a3a9  a5 49     lda $49
., a3ab  dd 02 01  cmp $0102,x
., a3ae  f0 07     beq $a3b7    ;same index so end (Z=0)
., a3b0  8a        txa
., a3b1  18        clc
., a3b2  69 12     adc #$12     ;FOR entry is 18 bytes long
., a3b4  aa        tax
., a3b5  d0 d8     bne $a38f    ;start again
., a3b7  60        rts
```

41912 BLTU: OPEN SPACE IN MEMORY

This routine enables text to be inserted into a BASIC program. A check is made to ensure that sufficient RAM is available, then an upwards memory move takes place. This is done by subtracting the block pointers to find the length of the block then moving memory up by the required amount. The following locations must be set on entry: ($58)=Top of destination +1, ($5A)=Top of source +1, ($5F)=Bottom of source.

```
., a3b8  20 08 a4  jsr $a408    ;check memory space
., a3bb  85 31     sta $31      ;<STREND - end of arrays +1
., a3bd  84 32     sty $32
., a3bf  38        sec
., a3c0  a5 5a     lda $5a      ;<top of source+1
., a3c2  e5 5f     sbc $5f      ;<bottom of source
., a3c4  85 22     sta $22      ;<number of bytes to move
., a3c6  a8        tay
., a3c7  a5 5b     lda $5b      ;repeat for hi bytes
., a3c9  e5 60     sbc $60
., a3cb  aa        tax          ;>number of bytes to move
., a3cc  e8        inx
., a3cd  98        tya
```

```
., a3ce  f0 23     beq $a3f3     ;lo byte of counter = 0
., a3d0  a5 5a     lda $5a
., a3d2  38        sec
., a3d3  e5 22     sbc $22
., a3d5  85 5a     sta $5a
., a3d7  b0 03     bcs $a3dc
., a3d9  c6 5b     dec $5b
., a3db  38        sec
., a3dc  a5 58     lda $58
., a3de  e5 22     sbc $22
., a3e0  85 58     sta $58
., a3e2  b0 08     bcs $a3ec
., a3e4  c6 59     dec $59
., a3e6  90 04     bcc $a3ec
., a3e8  b1 5a     lda ($5a),y   ;move source to destination
., a3ea  91 58     sta ($58),y
., a3ec  88        dey           ;next byte
., a3ed  d0 f9     bne $a3e8
., a3ef  b1 5a     lda ($5a),y   ;move source to destination
., a3f1  91 58     sta ($58),y
., a3f3  c6 5b     dec $5b       ;adjust block pointers
., a3f5  c6 59     dec $59
., a3f7  ca        dex           ;next page
., a3f8  d0 f2     bne $a3ec
., a3fa  60        rts
```

41979 GETSTK: CHECK STACK DEPTH

This routine checks whether a given number of bytes will fit onto the stack. (A) must be set to HALF the required number. If there is insufficient room then ?OUT OF MEMORY error. The stack does not occupy all of page 1 since 62 bytes are used as a workspace for other routines.

```
., a3fb  0a        asl           ;double test quantity
., a3fc  69 3e     adc #$3e      ;bottom 62 bytes are used as
                                 RAM
., a3fe  b0 35     bcs $a435     ;out of stack space
., a400  85 22     sta $22
., a402  ba        tsx
., a403  e4 22     cpx $22       ;check stack pointer is ok
., a405  90 2e     bcc $a435     ;?OUT OF MEMORY error
., a407  60        rts
```

41992 REASON: CHECK MEMORY OVERLAP

This routine is used to check to see if there is enough space in RAM for a program addition. (A/Y) must hold the address to be tested. This is compared against bottom of

strings. If they overlap, then garbage is collected. If there is still no room then ?OUT OF MEMORY error.

```
., a408  c4 34    cpy $34       ;>FRETOP - bottom of strings
., a40a  90 28    bcc $a434     ;ok
., a40c  d0 04    bne $a412     ;not - garbage collect
., a40e  c5 33    cmp $33       ;<FRETOP
., a410  90 22    bcc $a434     ;ok
., a412  48       pha           ;push test address
., a413  a2 09    ldx #$09
., a415  98       tya
., a416  48       pha
., a417  b5 57    lda $57,x     ;push temp pointers $57 - $60
., a419  ca       dex
., a41a  10 fa    bpl $a416
., a41c  20 26 b5 jsr $b526     ;garbage collect
., a41f  a2 f7    ldx #$f7
., a421  68       pla
., a422  95 61    sta $61,x     ;pull pointers
., a424  e8       inx
., a425  30 fa    bmi $a421
., a427  68       pla           ;pull test address
., a428  a8       tay
., a429  68       pla
., a42a  c4 34    cpy $34       ;repeat test on FRETOP
., a42c  90 06    bcc $a434     ;ok
., a42e  d0 05    bne $a435     ;?OUT OF MEMORY error
., a430  c5 33    cmp $33
., a432  b0 01    bcs $a435     ;?OUT OF MEMORY error
., a434  60       rts
```

42037 OMERR: OUTPUT ?OUT OF MEMORY ERROR

This routine sets the error pointer to ?OUT OF MEMORY, then enters the next routine.

```
., a435  a2 10    ldx #$10
```

42039 ERROR: ERROR ROUTINE

This routine prints an error message from the error messages table. (X) must hold the error number on entry. A table is given below of messages and their corresponding numbers.

```
., a437  6c 00 03 jmp ($0300)   ;vector IERROR - points to
                                 $A43A
., a43a  8a       txa
., a43b  0a       asl           ;double to provide offset
., a43c  aa       tax
```

```
., a43d  bd 26 a3 lda $a326,x  ;pointers to message
., a440  85 22    sta $22      ;<INDEX1
., a442  bd 27 a3 lda $a327,x
., a445  85 23    sta $23
., a447  20 cc ff jsr $ffcc    ;CLRCHN - close I/O channels
., a44a  a9 00    lda #$00
., a44c  85 13    sta $13      ;input prompt flag
., a44e  20 d7 aa jsr $aad7    ;output CR/LF
., a451  20 45 ab jsr $ab45    ;output question mark
., a454  a0 00    ldy #$00
., a456  b1 22    lda ($22),y  ;load byte of message into
                               (A)
., a458  48       pha
., a459  29 7f    and #$7f     ;zero bit 7
., a45b  20 47 ab jsr $ab47    ;output character in (A)
., a45e  c8       iny
., a45f  68       pla
., a460  10 f4    bpl $a456
., a462  20 7a a6 jsr $a67a    ;disable CONT
., a465  a9 69    lda #$69
., a467  a0 a3    ldy #$a3     ;$A369 points to message
                               'ERROR
```

ERROR MESSAGES AND THEIR CORRESPONDING NUMBERS

```
#01   1   TOO MANY FILES
#02   2   FILE OPEN
#03   3   FILE NOT OPEN
#04   4   FILE NOT FOUND
#05   5   DEVICE NOT PRESENT
#06   6   NOT INPUT FILE
#07   7   NOT OUTPUT FILE
#08   8   MISSING FILENAME
#09   9   ILLEGAL DEVICE NUMBER
#0A  10   NEXT WITHOUT FOR
#0B  11   SYNTAX
#0C  12   RETURN WITHOUT GOSUB
#0D  13   OUT OF DATA
#0E  14   ILLEGAL QUANTITY
#0F  15   OVERFLOW
#10  16   OUT OF MEMORY
#11  17   UNDEF'D STATEMENT
#12  18   BAD SUBSCRIPT
#13  19   REDIM'D ARRAY
#14  20   DIVISION BY ZERO
#15  21   ILLEGAL DIRECT
#16  22   TYPE MISMATCH
#17  23   STRING TOO LONG
#18  24   FILE DATA
```

```
#19  25   FORMULA TOO COMPLEX
#1A  26   CAN'T CONTINUE
#1B  27   UNDEF'D FUNCTION
#1C  28   VERIFY
#1D  29   LOAD
```

42089 ERRFIN: BREAK ENTRY

This routine prints "ERROR" or "BREAK" depending on whether program or direct mode is engaged. In program mode, it also prints "IN" and CURLIN.

```
., a469  20 1e ab jsr $ab1e    ;output string at (A/Y)
., a46c  a4 3a    ldy $3a      ;>CURLIN - line number or #FF
                               if direct
., a46e  c8       iny
., a46f  f0 03    beq $a474    ;direct mode - restart BASIC
., a471  20 c2 bd jsr $bdc2    ;output 'IN and CURLIN
```

42100 READY: RESTART BASIC

This routine prints "READY.", sets direct mode, and then waits for the next BASIC line or direct command to be entered.

```
., a474  a9 76    lda #$76     ;point to 'READY at $A376
., a476  a0 a3    ldy #$a3
., a478  20 1e ab jsr $ab1e    ;output string at (A/Y)
., a47b  a9 80    lda #$80
., a47d  20 90 ff jsr $ff90    ;SETMSG - control KERNAL
                               messages
```

42112 MAIN: INPUT & IDENTIFY BASIC LINE

This is the major BASIC interpreter routine, executed when in direct mode. BASIC text is placed in the input buffer with a zero terminator and processed. If the first character is numeric then it is treated as a program line. If not then it is treated as a direct command.

```
., a480  6c 02 03 jmp ($0302)  ;vector IMAIN points to $A483
., a483  20 60 a5 jsr $a560    ;input line to input buffer
., a486  86 7a    stx $7a      ;TXTPTR
., a488  84 7b    sty $7b
., a48a  20 73 00 jsr $0073    ;CHRGET
., a48d  aa       tax
., a48e  f0 f0    beq $a480    ;terminator - start again
., a490  a2 ff    ldx #$ff
```

```
., a492  86 3a     stx $3a      ;>CURLIN - set direct mode
., a494  90 06     bcc $a49c    ;number so set up text
., a496  20 79 a5  jsr $a579    ;crunch keywords
., a499  4c e1 a7  jmp $a7e1    ;execute statement
```

42140 MAIN1: GET LINE NUMBER & TOKENISE TEXT

This part of the MAIN routine reads the line number at the start of the BASIC line. It then searches the existing program for that line. If it is found, the line is deleted and the replacement inserted. All the keywords are crunched into tokens.

```
., a49c  20 6b a9  jsr $a96b    ;read LINNUM from text
., a49f  20 79 a5  jsr $a579    ;crunch keywords to tokens
```

42146 INSLIN: INSERT BASIC TEXT

If the line number already exists in text then memory is moved down to overwrite the line. CLR is performed and the link pointers are rebuilt. Memory is moved up to make room for a new (or replacement line if given) which is written into memory. Finally CLR is performed, the link pointers are rebuilt and the main loop re-entered.

```
., a4a2  84 0b     sty $0b      ;COUNT - input buffer pointer
., a4a4  20 13 a6  jsr $a613    ;search for LINNUM in text
., a4a7  90 44     bcc $a4ed    ;not found - insert new line
., a4a9  a0 01     ldy #$01
., a4ab  b1 5f     lda ($5f),y  ;($5F) points to line header
., a4ad  85 23     sta $23
., a4af  a5 2d     lda $2d      ;<VARTAB - start of variables
., a4b1  85 22     sta $22
., a4b3  a5 60     lda $60
., a4b5  85 25     sta $25
., a4b7  a5 5f     lda $5f
., a4b9  88        dey
., a4ba  f1 5f     sbc ($5f),y
., a4bc  18        clc
., a4bd  65 2d     adc $2d      ;<VARTAB
., a4bf  85 2d     sta $2d
., a4c1  85 24     sta $24
., a4c3  a5 2e     lda $2e      ;>VARTAB
., a4c5  69 ff     adc #$ff
., a4c7  85 2e     sta $2e      ;>VARTAB
., a4c9  e5 60     sbc $60
., a4cb  aa        tax
., a4cc  38        sec
```

```
., a4cd  a5 5f     lda $5f
., a4cf  e5 2d     sbc $2d       ;<VARTAB
., a4d1  a8        tay
., a4d2  b0 03     bcs $a4d7
., a4d4  e8        inx
., a4d5  c6 25     dec $25
., a4d7  18        clc
., a4d8  65 22     adc $22
., a4da  90 03     bcc $a4df
., a4dc  c6 23     dec $23
., a4de  18        clc
., a4df  b1 22     lda ($22),y   ;move memory to delete line
., a4e1  91 24     sta ($24),y   ;($22) = source
., a4e3  c8        iny           ;($24) = destination
., a4e4  d0 f9     bne $a4df     ;(X/Y) = number of bytes
., a4e6  e6 23     inc $23
., a4e8  e6 25     inc $25
., a4ea  ca        dex
., a4eb  d0 f2     bne $a4df
., a4ed  20 59 a6  jsr $a659     ;reset execution and do CLR
., a4f0  20 33 a5  jsr $a533     ;rebuild link pointers
., a4f3  ad 00 02  lda $0200     ;input buffer
., a4f6  f0 88     beq $a480     ;terminator - input &
                                 identify BASIC
., a4f8  18        clc
., a4f9  a5 2d     lda $2d       ;VARTAB
., a4fb  85 5a     sta $5a
., a4fd  65 0b     adc $0b       ;COUNT
., a4ff  85 58     sta $58
., a501  a4 2e     ldy $2e
., a503  84 5b     sty $5b
., a505  90 01     bcc $a508
., a507  c8        iny
., a508  84 59     sty $59
., a50a  20 b8 a3  jsr $a3b8     ;move block of memory up
., a50d  a5 14     lda $14       ;LINNUM - temp integer value
., a50f  a4 15     ldy $15
., a511  8d fe 01  sta $01fe
., a514  8c ff 01  sty $01ff
., a517  a5 31     lda $31       ;STREND - end of arrays +1
., a519  a4 32     ldy $32
., a51b  85 2d     sta $2d       ;VARTAB
., a51d  84 2e     sty $2e
., a51f  a4 0b     ldy $0b       ;COUNT
., a521  88        dey
., a522  b9 fc 01  lda $01fc,y   ;write line of text into
                                 memory
., a525  91 5f     sta ($5f),y
., a527  88        dey
```

```
., a528  10 f8     bpl $a522
., a52a  20 59 a6  jsr $a659      ;reset execution and do CLR
., a52d  20 33 a5  jsr $a533      ;rebuild link pointers
., a530  4c 80 a4  jmp $a480      ;input & identify BASIC
```

42291 LINKPRG: RECHAIN LINES

Starting from TXTTAB, BASIC text is searched for a zero byte terminator denoting the end of a line (EOL). Once found, the link address for that line is calculated. INDEX1 is updated to point to the end of this line and the search is repeated. Note that lines longer than 255 characters cause the routine to hang.

```
., a533  a5 2b     lda $2b        ;TXTTAB - start of BASIC text
., a535  a4 2c     ldy $2c
., a537  85 22     sta $22        ;INDEX1 - temp pointer to
                                  text
., a539  84 23     sty $23
., a53b  18        clc
., a53c  a0 01     ldy #$01
., a53e  b1 22     lda ($22),y    ;examine link address
., a540  f0 1d     beq $a55f      ;zero - end of program
., a542  a0 04     ldy #$04       ;by-pass link addr and line
                                  number
., a544  c8        iny
., a545  b1 22     lda ($22),y    ;get next byte of text
., a547  d0 fb     bne $a544      ;not EOL so repeat
., a549  c8        iny
., a54a  98        tya
., a54b  65 22     adc $22        ;calculate link addr
., a54d  aa        tax
., a54e  a0 00     ldy #$00
., a550  91 22     sta ($22),y    ;store link addr for this
                                  line
., a552  a5 23     lda $23
., a554  69 00     adc #$00
., a556  c8        iny
., a557  91 22     sta ($22),y    ;store hi byte
., a559  86 22     stx $22        ;set INDEX1 to point to new
                                  line
., a55b  85 23     sta $23
., a55d  90 dd     bcc $a53c      ;restart search
., a55f  60        rts
```

42336 INLIN: INPUT LINE INTO BUFFER

This routine inputs a character from the keyboard. If it is a carriage return (#0D) then the zero byte input buffer

terminator is set up. If not, then the character is placed into the queue in the input buffer. If the length of the queue exceeds 80 (#59) characters, then ?STRING TOO LONG error.

```
., a560  a2 00     ldx #$00
., a562  20 12 e1  jsr $e112      ;BCHIN - input character from
                                  keyboard
., a565  c9 0d     cmp #$0d       ;<CR>?
., a567  f0 0d     beq $a576      ;yes - set terminator and
                                  exit
., a569  9d 00 02  sta $0200,x    ;store character in input
                                  buffer
., a56c  e8        inx
., a56d  e0 59     cpx #$59       ;buffer full?
., a56f  90 f1     bcc $a562      ;no - repeat process
., a571  a2 17     ldx #$17       ;flag ?STRING TOO LONG
., a573  4c 37 a4  jmp $a437      ;do error
., a576  4c ca aa  jmp $aaca      ;add zero terminator to
                                  string
```

42361 CRUNCH: TOKENISE INPUT BUFFER

The input buffer is scanned until its zero byte terminator is found. Characters are compared with the keyword table and each recognised keyword is converted into a 1 byte token consisting of the position of the keyword in the table ORed with #80. This routine can be fooled by such as eN, dA, nE etc. <?>, <">, and <pi> are tested for and processed separately. The routine is vectored through ($0304) to enable additional keyword tokens to be added to BASIC.

```
., a579  6c 04 03  jmp ($0304)    ;vector ICRNCH - points to
                                  $A57C
., a57c  a6 7a     ldx $7a        ;TXTPTR - position in input
                                  buffer
., a57e  a0 04     ldy #$04
., a580  84 0f     sty $0f        ;GARBFL - flag for quotes
., a582  bd 00 02  lda $0200,x    ;input buffer
., a585  10 07     bpl $a58e
., a587  c9 ff     cmp #$ff       ;<pi>?
., a589  f0 3e     beq $a5c9      ;yes - skip it
., a58b  e8        inx
., a58c  d0 f4     bne $a582      ;do next byte
., a58e  c9 20     cmp #$20       ;<space>?
., a590  f0 37     beq $a5c9      ;skip it
., a592  85 08     sta $08        ;ENDCHR - scan for next quote
., a594  c9 22     cmp #$22       ;<quote>?
., a596  f0 56     beq $a5ee      ;yes - skip to next quote
```

```
., a598  24 0f     bit $0f       ;GARBFL
., a59a  70 2d     bvs $a5c9     ;skip it
., a59c  c9 3f     cmp #$3f      ;<query>?
., a59e  d0 04     bne $a5a4     ;no - continue checks
., a5a0  a9 99     lda #$99      ;token PRINT
., a5a2  d0 25     bne $a5c9     ;store & get next char
., a5a4  c9 30     cmp #$30      ;<0>?
., a5a6  90 04     bcc $a5ac
., a5a8  c9 3c     cmp #$3c      ;ascii <?
., a5aa  90 1d     bcc $a5c9     ;skip it
., a5ac  84 71     sty $71       ;(A) now holds poss keyword
                                 char
., a5ae  a0 00     ldy #$00
., a5b0  84 0b     sty $0b       ;COUNT - keyword # - makes up
                                 token
., a5b2  88        dey
., a5b3  86 7a     stx $7a       ;TXTPTR
., a5b5  ca        dex
., a5b6  c8        iny
., a5b7  e8        inx
., a5b8  bd 00 02 lda $0200,x   ;input buffer
., a5bb  38        sec
., a5bc  f9 9e a0 sbc $a09e,y   ;keyword table
., a5bf  f0 f5     beq $a5b6     ;match - test next character
., a5c1  c9 80     cmp #$80      ;end of keyword
., a5c3  d0 30     bne $a5f5     ;no match - try next keyword
., a5c5  05 0b     ora $0b       ;COUNT - OR with #80 to give
                                 token
., a5c7  a4 71     ldy $71
., a5c9  e8        inx
., a5ca  c8        iny
., a5cb  99 fb 01 sta $01fb,y   ;store processed character
., a5ce  b9 fb 01 lda $01fb,y
., a5d1  f0 36     beq $a609     ;terminator - end
., a5d3  38        sec
., a5d4  e9 3a     sbc #$3a
., a5d6  f0 04     beq $a5dc
., a5d8  c9 49     cmp #$49
., a5da  d0 02     bne $a5de
., a5dc  85 0f     sta $0f       ;GARBFL
., a5de  38        sec
., a5df  e9 55     sbc #$55
., a5e1  d0 9f     bne $a582     ;start again
., a5e3  85 08     sta $08       ;ENDCHR - scan for end quotes
., a5e5  bd 00 02 lda $0200,x   ;input buffer
., a5e8  f0 df     beq $a5c9     ;terminator
., a5ea  c5 08     cmp $08       ;ENDCHR
., a5ec  f0 db     beq $a5c9     ;yes - reenter main routine
., a5ee  c8        iny
```

```
., a5ef  99 fb 01 sta $01fb,y
., a5f2  e8       inx
., a5f3  d0 f0    bne $a5e5    ;next character
., a5f5  a6 7a    ldx $7a
., a5f7  e6 0b    inc $0b      ;COUNT - next keyword
., a5f9  c8       iny
., a5fa  b9 9d a0 lda $a09d,y  ;keyword table
., a5fd  10 fa    bpl $a5f9    ;find end of current keyword
., a5ff  b9 9e a0 lda $a09e,y
., a602  d0 b4    bne $a5b8    ;not end of table - continue
., a604  bd 00 02 lda $0200,x  ;input buffer
., a607  10 be    bpl $a5c7
., a609  99 fd 01 sta $01fd,y  ;move character down
., a60c  c6 7b    dec $7b
., a60e  a9 ff    lda #$ff
., a610  85 7a    sta $7a      ;set TXTPTR to #FF
., a612  60       rts
```

<u>42515 FNDLIN: SEARCH FOR LINE NUMBER</u>

BASIC text is scanned from its start to the end of the BASIC program using ($5F) as a pointer. The line number for each line is compared with that in LINNUM. If they match then carry is set and ($5F) points to the header for that line. If the line number is not found, then the carry flag is cleared.

```
., a613  a5 2b    lda $2b      ;TXTTAB - start of BASIC
., a615  a6 2c    ldx $2c
., a617  a0 01    ldy #$01
., a619  85 5f    sta $5f      ;temp pointer
., a61b  86 60    stx $60
., a61d  b1 5f    lda ($5f),y  ;examine link address
., a61f  f0 1f    beq $a640    ;end of program - address not
                               found
., a621  c8       iny
., a622  c8       iny
., a623  a5 15    lda $15      ;>LINNUM - temp integer
., a625  d1 5f    cmp ($5f),y  ;line # in text
., a627  90 18    bcc $a641    ;LINNUM > line #
., a629  f0 03    beq $a62e    ;equal - test other byte
., a62b  88       dey
., a62c  d0 09    bne $a637    ;next line
., a62e  a5 14    lda $14      ;<LINNUM
., a630  88       dey
., a631  d1 5f    cmp ($5f),y  ;line # in text
., a633  90 0c    bcc $a641    ;LINNUM > line #
., a635  f0 0a    beq $a641    ;positive match - RTS
., a637  88       dey
```

```
., a638  b1 5f     lda ($5f),y  ;get link addr & store in
                                (A/Y)
., a63a  aa        tax
., a63b  88        dey
., a63c  b1 5f     lda ($5f),y
., a63e  b0 d7     bcs $a617    ;examine next line in text
., a640  18        clc
., a641  60        rts
```

42562 SCRTCH: PERFORM NEW

Firstly there is a syntax check to ensure no parameters are passed (ie NEW 20 etc.), otherwise a ?SYNTAX error is generated.. The link address of the first line of BASIC text is set to $0000. VARTAB is set to TXTTAB + 2 and TXTPTR is reset. (A) is set to zero for the syntax check in CLR which is then performed.

```
., a642  d0 fd     bne $a641    ;syntax check - RTS if
                                invalid
., a644  a9 00     lda #$00
., a646  a8        tay
., a647  91 2b     sta ($2b),y  ;set 1st link addr to $0000
., a649  c8        iny
., a64a  91 2b     sta ($2b),y
., a64c  a5 2b     lda $2b      ;TXTTAB - start of BASIC
., a64e  18        clc
., a64f  69 02     adc #$02
., a651  85 2d     sta $2d      ;VARTAB = TXTTAB +2 (start of
                                vars)
., a653  a5 2c     lda $2c
., a655  69 00     adc #$00
., a657  85 2e     sta $2e
., a659  20 8e a6  jsr $a68e    ;reset TXTPTR
., a65c  a9 00     lda #$00     ;prepare for CLR syntax check
```

42590 CLEAR: PERFORM CLR

As with NEW, CLR has a syntax check. All channels and files are closed and variables are erased by resetting FRETOP = MEMSIZ and ARYTAB, STREND = VARTAB. RESTORE is performed, the string descriptor stack pointer is reset and the CONT pointer is zeroed.

```
., a65e  d0 2d     bne $a68d    ;syntax check - RTS if
                                invalid
., a660  20 e7 ff  jsr $ffe7    ;CLALL - close I/O files &
                                channels
., a663  a5 37     lda $37      ;MEMSIZ - highest BASIC
```

```
                                             address
., a665  a4 38     ldy $38
., a667  85 33     sta $33         ;FRETOP - bottom of strings
., a669  84 34     sty $34
., a66b  a5 2d     lda $2d         ;VARTAB - start of variables
., a66d  a4 2e     ldy $2e
., a66f  85 2f     sta $2f         ;ARYTAB - start of arrays
., a671  84 30     sty $30
., a673  85 31     sta $31         ;STREND - end of arrays +1
., a675  84 32     sty $32
., a677  20 1d a8  jsr $a81d       ;do RESTORE
., a67a  a2 19     ldx #$19
., a67c  86 16     stx $16         ;TEMPPT - descriptor stack
                                   ptr
., a67e  68        pla
., a67f  a8        tay
., a680  68        pla
., a681  a2 fa     ldx #$fa
., a683  9a        txs             ;reset stack
., a684  48        pha
., a685  98        tya
., a686  48        pha
., a687  a9 00     lda #$00
., a689  85 3e     sta $3e         ;OLDTXT - pointer for CONT
., a68b  85 10     sta $10         ;SUBFLG
., a68d  60        rts
```

42638 STXPT: RESET TXTPTR

This routine resets TXTPTR to the start of BASIC text.

```
., a68e  18        clc
., a68f  a5 2b     lda $2b         ;TXTTAB - start of BASIC
., a691  69 ff     adc #$ff
., a693  85 7a     sta $7a         ;TXTPTR
., a695  a5 2c     lda $2c
., a697  69 ff     adc #$ff
., a699  85 7b     sta $7b
., a69b  60        rts
```

42652 LIST: PERFORM LIST

Any parameters that follow the LIST command are thoroughly tested for start and finish line numbers. If either is not found, the defaults are:- start =#0000, end =#FFFF. LIST is performed until either the end of the program (link address =$0000) or a line number greater than that in LINNUM is found. The line number is output, then each byte of text is processed. If it is a token (>#7F), then it is expanded and

the keyword output, unless the token is within quotes, when it is output intact. Finally, direct mode is set, and the main BASIC input and identify loop is entered.

```
., a69c  90 06     bcc $a6a4    ;ascii number? (entry is from
                                 CHRGET)
., a69e  f0 04     beq $a6a4    ;no parameters
., a6a0  c9 ab     cmp #$ab     ;token -?
., a6a2  d0 e9     bne $a68d    ;no - RTS
., a6a4  20 6b a9  jsr $a96b    ;read LINNUM from text
., a6a7  20 13 a6  jsr $a613    ;search for LINNUM - header
                                 in ($5F)
., a6aa  20 79 00  jsr $0079    ;CHRGOT - recap parameter
., a6ad  f0 0c     beq $a6bb    ;no parameter
., a6af  c9 ab     cmp #$ab     ;token -?
., a6b1  d0 8e     bne $a641    ;no - do RTS
., a6b3  20 73 00  jsr $0073    ;CHRGET - get next byte of
                                 text
., a6b6  20 6b a9  jsr $a96b    ;read LINNUM from text
., a6b9  d0 86     bne $a641    ;not found - RTS
., a6bb  68        pla
., a6bc  68        pla
., a6bd  a5 14     lda $14      ;LINNUM - end line number
., a6bf  05 15     ora $15      ;($5F) holds start line
                                 address
., a6c1  d0 06     bne $a6c9
., a6c3  a9 ff     lda #$ff
., a6c5  85 14     sta $14      ;LINNUM defaults t0 #FFFF if
                                 no param
., a6c7  85 15     sta $15      ;($5F) defaults to $0801
., a6c9  a0 01     ldy #$01
., a6cb  84 0f     sty $0f      ;GARBFL - quotes flag
., a6cd  b1 5f     lda ($5f),y  ;get link address
., a6cf  f0 43     beq $a714    ;end of program
., a6d1  20 2c a8  jsr $a82c    ;test STOP key
., a6d4  20 d7 aa  jsr $aad7    ;output CR/LF
., a6d7  c8        iny
., a6d8  b1 5f     lda ($5f),y  ;put line number into (X/A)
., a6da  aa        tax
., a6db  c8        iny
., a6dc  b1 5f     lda ($5f),y
., a6de  c5 15     cmp $15      ;compare with end line number
., a6e0  d0 04     bne $a6e6
., a6e2  e4 14     cpx $14
., a6e4  f0 02     beq $a6e8
., a6e6  b0 2c     bcs $a714    ;line # > LINNUM
., a6e8  84 49     sty $49
., a6ea  20 cd bd  jsr $bdcd    ;output number in (X/A)
., a6ed  a9 20     lda #$20     ;ASCII space
```

```
., a6ef  a4 49     ldy $49
., a6f1  29 7f     and #$7f
., a6f3  20 47 ab  jsr $ab47     ;output character in (A)
., a6f6  c9 22     cmp #$22      ;ASCII quotes?
., a6f8  d0 06     bne $a700     ;no
., a6fa  a5 0f     lda $0f
., a6fc  49 ff     eor #$ff      ;set quotes flag
., a6fe  85 0f     sta $0f
., a700  c8        iny
., a701  f0 11     beq $a714
., a703  b1 5f     lda ($5f),y   ;get byte of text
., a705  d0 10     bne $a717     ;not EOL - process character
., a707  a8        tay
., a708  b1 5f     lda ($5f),y   ;get link address for next
                                 line
., a70a  aa        tax
., a70b  c8        iny
., a70c  b1 5f     lda ($5f),y
., a70e  86 5f     stx $5f       ;store link address in ($5F)
., a710  85 60     sta $60
., a712  d0 b5     bne $a6c9     ;not end of program - list
                                 line
., a714  4c 86 e3  jmp $e386     ;restart BASIC
```

42775 QPLOP: HANDLE LIST CHARACTER

This routine tests for a keyword. The character is Printed straight if it is not a token, or if it is <PI> (#FF) or if it is in quotes. The token is expanded into the keyword, and output, then LIST is continued.

```
., a717  6c 06 03  jmp ($0306)   ;vector IQPLOP - points to
                                 $a71a
., a71a  10 d7     bpl $a6f3     ;not token - output character
., a71c  c9 ff     cmp #$ff      ;token - pi?
., a71e  f0 d3     beq $a6f3     ;output character
., a720  24 0f     bit $0f       ;GARBFL - LIST quote flag
., a722  30 cf     bmi $a6f3     ;quotes flag set-output
                                 character
., a724  38        sec
., a725  e9 7f     sbc #$7f
., a727  aa        tax           ;(X) now holds keyword number
., a728  84 49     sty $49
., a72a  a0 ff     ldy #$ff
., a72c  ca        dex
., a72d  f0 08     beq $a737     ;keyword is next in table
., a72f  c8        iny
., a730  b9 9e a0  lda $a09e,y   ;scan table for correct
                                 keyword
```

```
., a733  10 fa     bpl $a72f
., a735  30 f5     bmi $a72c
., a737  c8        iny
., a738  b9 9e a0 lda $a09e,y  ;get character from table
., a73b  30 b2     bmi $a6ef    ;end of keyword - continue
                                LIST
., a73d  20 47 ab jsr $ab47    ;output character in (A)
., a740  d0 f5     bne $a737    ;next character in keyword
```

42818 FOR: PERFORM FOR

This routine sets up a block of 18 bytes of data on the stack. The loop variable is assigned and the stack checked for 18 bytes of available space. The token TO is confirmed, and the limit value of the index is evaluated into fac#1. If the token STEP is found then this value is evaluated. If not, it defaults to 1. The last byte to be pushed onto the stack is the token FOR (#81). The following routine is dropped through to and a BASIC warm start performed. The 18 bytes pushed onto the stack are as follows:

pointer to the following statement. (2)
current line number. (2)
upper limit of index variable. (5)
step value with sign. (7)
pointer to index variable. (2)
token FOR. (1)

```
., a742  a9 80     lda #$80
., a744  85 10     sta $10      ;SUBFLG - user function call
., a746  20 a5 a9 jsr $a9a5    ;do LET - FORPNT is pointer
                                to index
., a749  20 8a a3 jsr $a38a    ;find FOR entry on stack
., a74c  d0 05     bne $a753    ;not found
., a74e  8a        txa
., a74f  69 0f     adc #$0f
., a751  aa        tax
., a752  9a        txs
., a753  68        pla
., a754  68        pla
., a755  a9 09     lda #$09
., a757  20 fb a3 jsr $a3fb    ;check stack for 18 bytes
                                space
., a75a  20 06 a9 jsr $a906    ;search for next statement
., a75d  18        clc
., a75e  98        tya
., a75f  65 7a     adc $7a      ;push TXTPTR - points to next
                                statement
., a761  48        pha
```

```
., a762  a5 7b     lda $7b
., a764  69 00     adc #$00
., a766  48        pha
., a767  a5 3a     lda $3a        ;push CURLIN-current line num
., a769  48        pha
., a76a  a5 39     lda $39
., a76c  48        pha
., a76d  a9 a4     lda #$a4       ;token TO
., a76f  20 ff ae  jsr $aeff      ;confirm character in (A)
., a772  20 8d ad  jsr $ad8d      ;confirm numeric result
., a775  20 8a ad  jsr $ad8a      ;evaluate expression in text
., a778  a5 66     lda $66        ;FACSGN - fac#1 sign
., a77a  09 7f     ora #$7f
., a77c  25 62     and $62
., a77e  85 62     sta $62
., a780  a9 8b     lda #$8b
., a782  a0 a7     ldy #$a7       ;$A78B is return adds for JMP
., a784  85 22     sta $22
., a786  84 23     sty $23
., a788  4c 43 ae  jmp $ae43      ;push fac#1
., a78b  a9 bc     lda #$bc
., a78d  a0 b9     ldy #$b9
., a78f  20 a2 bb  jsr $bba2      ;load fac#1 at (A/Y)=$B9BC=#1
., a792  20 79 00  jsr $0079      ;CHRGOT
., a795  c9 a9     cmp #$a9       ;token STEP
., a797  d0 06     bne $a79f      ;not found - default = #1
., a799  20 73 00  jsr $0073      ;CHRGET
., a79c  20 8a ad  jsr $ad8a      ;evaluate expression in text
., a79f  20 2b bc  jsr $bc2b      ;check sign of fac#1
., a7a2  20 38 ae  jsr $ae38      ;push fac#1 plus sign
., a7a5  a5 4a     lda $4a        ;push FORPNT - pointer to
                                  index var
., a7a7  48        pha
., a7a8  a5 49     lda $49
., a7aa  48        pha
., a7ab  a9 81     lda #$81
., a7ad  48        pha
```

42926 NEWSTT: BASIC WARM START

This routine tests the stop key, and then tests for program/direct mode. In program mode, the CONT pointer is updated and if EOL is found on examining TXTPTR, end of program is checked.

```
., a7ae  20 2c a8  jsr $a82c      ;test stop key
., a7b1  a5 7a     lda $7a        ;TXTPTR
., a7b3  a4 7b     ldy $7b
```

```
., a7b5  c0 02     cpy #$02       ;$0200- ie input buffer -
                                  direct mode
., a7b7  ea        nop
., a7b8  f0 04     beq $a7be      ;direct mode - don't do
                                  update
., a7ba  85 3d     sta $3d        ;OLDTXT - pointer for CONT
., a7bc  84 3e     sty $3e
., a7be  a0 00     ldy #$00
., a7c0  b1 7a     lda ($7a),y    ;get byte of text indicated
                                  by TXTPTR
., a7c2  d0 43     bne $a807      ;if not EOL - do keyword
```

42948 CKEOL: CHECK END OF PROGRAM

This routine assumes that TXTPTR points to EOL. It then tests for a null value link address, signifying end of program. If found then END is performed. If not, then CURLIN is updated with the next line number and TXTPTR is also updated.

```
., a7c4  a0 02     ldy #$02
., a7c6  b1 7a     lda ($7a),y    ;check link address for zero
                                  byte
., a7c8  18        clc
., a7c9  d0 03     bne $a7ce      ;not end of program
., a7cb  4c 4b a8  jmp $a84b      ;do END
., a7ce  c8        iny
., a7cf  b1 7a     lda ($7a),y    ;get next line number
., a7d1  85 39     sta $39        ;into CURLIN - current line
                                  number
., a7d3  c8        iny
., a7d4  b1 7a     lda ($7a),y
., a7d6  85 3a     sta $3a
., a7d8  98        tya
., a7d9  65 7a     adc $7a        ;update TXTPTR
., a7db  85 7a     sta $7a
., a7dd  90 02     bcc $a7e1
., a7df  e6 7b     inc $7b
```

42977 GONE: PREPARE TO EXECUTE STATEMENT

(A) is loaded with the next byte of text by CHRGET, the keyword is performed and a warm start is called.

```
., a7e1  6c 08 03  jmp ($0308)    ;vector IGONE - points to
                                  $A7E4
., a7e4  20 73 00  jsr $0073      ;CHRGET
```

```
.., a7e7 20 ed a7 jsr $a7ed      ;perform BASIC keyword
.., a7ea  4c ae a7 jmp $a7ae     ;BASIC warm start
```

42989 GONE3: PERFORM BASIC KEYWORD

If (A)=0 then EOL. The byte is tested. If is not a token (ie. <#80) then it is assumed to be a variable and LET is performed. If it is a token then it is processed to give an offset to the keyword vector table. The vector is pushed onto the stack so that the keyword routine is 'returned to' when RTS is performed. The token is also tested to ensure it is in the valid range (#80 - #CA) so that BASIC 4 tokens cannot be executed. There is a patch for GO to check that it is followed by TO.

```
.., a7ed  f0 3c     beq $a82b    ;RTS
.., a7ef  e9 80     sbc #$80
.., a7f1  90 11     bcc $a804    ;not a token - assign
                                 variable
.., a7f3  c9 23     cmp #$23
.., a7f5  b0 17     bcs $a80e
.., a7f7  0a        asl
.., a7f8  a8        tay          ;(Y) now gives vector offset
.., a7f9  b9 0d a0 lda $a00d,y  ;get and push keyword vector
.., a7fc  48        pha
.., a7fd  b9 0c a0 lda $a00c,y
.., a800  48        pha
.., a801  4c 73 00 jmp $0073    ;CHRGET and RTS to keyword
.., a804  4c a5 a9 jmp $a9a5    ;do LET
.., a807  c9 3a     cmp #$3a
.., a809  f0 d6     beq $a7e1    ;prepare to execute statement
.., a80b  4c 08 af jmp $af08    ;?SYNTAX error
.., a80e  c9 4b     cmp #$4b
.., a810  d0 f9     bne $a80b    ;token beyond limit of table
.., a812  20 73 00 jsr $0073    ;CHRGET
.., a815  a9 a4     lda #$a4     ;token TO
.., a817  20 ff ae jsr $aeff    ;confirm character in (A)
.., a81a  4c a0 a8 jmp $a8a0    ;do GOTO
```

43037 RESTOR: PERFORM RESTORE

This routine sets the DATA pointer to equal start of BASIC.

```
.., a81d  38        sec
.., a81e  a5 2b     lda $2b      ;TXTTAB - start of BASIC
.., a820  e9 01     sbc #$01
.., a822  a4 2c     ldy $2c
.., a824  b0 01     bcs $a827
.., a826  88        dey
```

```
., a827  85 41     sta $41      ;DATPTR - pointer to DATA
                                statement
., a829  84 42     sty $42      ;DATPTR = TXTTAB +1
., a82b  60        rts
```

43052 STOP: PERFORM STOP, END, BREAK

This routine tests for the <STOP> key being pressed. END and STOP also use this routine. If STOP or <STOP> is pressed, then the message 'BREAK IN' and CURLIN is printed. BASIC is finally restarted. In the course of the routine, OLDTXT, OLDLIN and CURLIN are updated.

```
., a82c  20 e1 ff  jsr $ffe1    ;STOP - test stop key
., a82f  b0 01     bcs $a832    ;stop pressed - RTS
., a831  18        clc
., a832  d0 3c     bne $a870    ;RTS
., a834  a5 7a     lda $7a      ;TXTPTR
., a836  a4 7b     ldy $7b
., a838  a6 3a     ldx $3a      ;>CURLIN - line no. or #FF
                                for direct
., a83a  e8        inx
., a83b  f0 0c     beq $a849    ;direct mode - don't do
                                update
., a83d  85 3d     sta $3d      ;OLDTXT - pointer for CONT
., a83f  84 3e     sty $3e
., a841  a5 39     lda $39      ;CURLIN
., a843  a4 3a     ldy $3a
., a845  85 3b     sta $3b      ;OLDLIN - previous line
                                number
., a847  84 3c     sty $3c
., a849  68        pla
., a84a  68        pla
., a84b  a9 81     lda #$81
., a84d  a0 a3     ldy #$a3     ;$A381 points to message
                                'BREAK
., a84f  90 03     bcc $a854    ;END not STOP
., a851  4c 69 a4  jmp $a469    ;output 'BREAK/ERROR
., a854  4c 86 e3  jmp $e386    ;warm restart
```

43095 CONT: PERFORM CONT

A check is made on syntax to ensure that no parameters are given, then if the high byte of the CONT pointer is zero, a ?CAN'T CONTINUE error is generated. OLDTXT is put into TXTPTR and OLDLIN in CURLIN.

```
., a857  d0 17     bne $a870    ;syntax check - RTS if
                                invalid
```

```
., a859  a2 1a     ldx #$1a     ;pointer - ?CAN'T CONTINUE
., a85b  a4 3e     ldy $3e      ;OLDTXT - pointer to CONT
., a85d  d0 03     bne $a862    ;not zero - can continue
., a85f  4c 37 a4  jmp $a437    ;do error
., a862  a5 3d     lda $3d      ;put OLDTXT in TXTPTR
., a864  85 7a     sta $7a
., a866  84 7b     sty $7b
., a868  a5 3b     lda $3b      ;put OLDLIN in CURLIN
., a86a  a4 3c     ldy $3c
., a86c  85 39     sta $39
., a86e  84 3a     sty $3a
., a870  60        rts
```

43121 RUN: PERFORM RUN

KERNAL messages are disabled and CLR is performed. If parameters are given (eg. RUN 200) then GOTO is performed, otherwise the program is run from the start.

```
., a871  08        php
., a872  a9 00     lda #$00
., a874  20 90 ff  jsr $ff90    ;SETMSG - control KERNAL
                                messages
., a877  28        plp
., a878  d0 03     bne $a87d    ;parameters included
., a87a  4c 59 a6  jmp $a659    ;reset TXTPTR to start of
                                program
., a87d  20 60 a6  jsr $a660    ;do CLR without syntax check
., a880  4c 97 a8  jmp $a897    ;do GOTO
```

43139 GOSUB: PERFORM GOSUB

The stack is tested for 6 free bytes and the following 5 bytes are pushed onto the stack: TXTPTR, CURLIN, token GOSUB. Having done this, GOTO is performed.

```
., a883  a9 03     lda #$03
., a885  20 fb a3  jsr $a3fb    ;check stack for space (6
                                bytes)
., a888  a5 7b     lda $7b      ;push TXTPTR
., a88a  48        pha
., a88b  a5 7a     lda $7a
., a88d  48        pha
., a88e  a5 3a     lda $3a      ;push CURLIN - current line
                                number
., a890  48        pha
., a891  a5 39     lda $39
., a893  48        pha
., a894  a9 8d     lda #$8d     ;push GOSUB token
```

```
., a896  48        pha
., a897  20 79 00 jsr $0079     ;CHRGOT
., a89a  20 a0 a8 jsr $a8a0     ;do GOTO
., a89d  4c ae a7 jmp $a7ae     ;BASIC warm start
```

43168 GOTO: PERFORM GOTO

The line number given in text is searched for in one of two ways: If the new line number is higher than the old line number, BASIC is searched from its current position. If it is lower, then BASIC is searched from the start. This proceedure can save time in searching long programs. The difference may be seen in 10000 GOTO 10001 and 10000 GOTO 9999. Once the line has been found, execution continues from that point.

```
., a8a0  20 6b a9 jsr $a96b     ;get line number into LINNUM
., a8a3  20 09 a9 jsr $a909     ;search for next line
., a8a6  38        sec
., a8a7  a5 39     lda $39      ;CURLIN - current line number
., a8a9  e5 14     sbc $14      ;LINNUM - temp integer value
., a8ab  a5 3a     lda $3a
., a8ad  e5 15     sbc $15
., a8af  b0 0b     bcs $a8bc    ;do search from start
., a8b1  98        tya
., a8b2  38        sec
., a8b3  65 7a     adc $7a      ;TXTPTR - do search from here
., a8b5  a6 7b     ldx $7b
., a8b7  90 07     bcc $a8c0
., a8b9  e8        inx
., a8ba  b0 04     bcs $a8c0
., a8bc  a5 2b     lda $2b      ;TXTTAB - start of BASIC
., a8be  a6 2c     ldx $2c
., a8c0  20 17 a6 jsr $a617     ;search for LINNUM from (A/X)
., a8c3  90 1e     bcc $a8e3    ;line not found - ?UNDEF'D
                                STATEMENT
., a8c5  a5 5f     lda $5f      ;store header posn -1 in
                                TXTPTR
., a8c7  e9 01     sbc #$01
., a8c9  85 7a     sta $7a
., a8cb  a5 60     lda $60
., a8cd  e9 00     sbc #$00
., a8cf  85 7b     sta $7b
., a8d1  60        rts
```

43218 RETURN: PERFORM RETURN

Syntax is checked for no parameters. >FORPNT is set to $FF for the fetch GOSUB/FOR routine. This should return with the

GOSUB token in (A), or a ?RETURN WITHOUT GOSUB error is generated. The return address is recovered and execution continues via DATA

```
., a8d2  d0 fd     bne $a8d1    ;check syntax - RTS if
                                invalid
., a8d4  a9 ff     lda #$ff
., a8d6  85 4a     sta $4a      ;>FORPNT - pointer to index
                                variable
., a8d8  20 8a a3  jsr $a38a    ;find FOR/GOSUB on stack
., a8db  9a        txs
., a8dc  c9 8d     cmp #$8d     ;token GOSUB?
., a8de  f0 0b     beq $a8eb
., a8e0  a2 0c     ldx #$0c     ;flag ?RETURN WITHOUT GOSUB
., a8e2  2c a2 11  bit $11a2    ;mask - flag ?UNDEF'D
                                STATEMENT
., a8e5  4c 37 a4  jmp $a437    ;do error
., a8e8  4c 08 af  jmp $af08    ;?SYNTAX error
., a8eb  68        pla
., a8ec  68        pla
., a8ed  85 39     sta $39      ;pull CURLIN - current line
                                number
., a8ef  68        pla
., a8f0  85 3a     sta $3a
., a8f2  68        pla
., a8f3  85 7a     sta $7a      ;pull TXTPTR
., a8f5  68        pla
., a8f6  85 7b     sta $7b
```

43256 DATA: PERFORM DATA

The start of the next statement is found and stored in TXTPTR. Execution of the program continues at this point, thus all parameters given are just passed over in the same way as REM ($A93B).

```
., a8f8  20 06 a9  jsr $a906    ;search for next statement
., a8fb  98        tya          ;offset is in (Y)
., a8fc  18        clc
., a8fd  65 7a     adc $7a      ;add to TXTPTR
., a8ff  85 7a     sta $7a
., a901  90 02     bcc $a905
., a903  e6 7b     inc $7b
., a905  60        rts
```

43270 DATAN: SEARCH FOR NEXT STATEMENT / LINE

There are two entry points to this routine. $A906 is the entry to find the next statement, and $A909 is the entry to

find the next line. Text is searched until a zero terminator or colon (#3A) is found. On exit, the offset to this point is held in (Y).

```
., a906  a2 3a    ldx #$3a      ;ASCII colon - search for
                                statement/EOL
., a908  2c a2 00 bit $00a2     ;mask - entry point for EOL
                                only
., a90b  86 07    stx $07       ;CHARAC - search character
., a90d  a0 00    ldy #$00
., a90f  84 08    sty $08       ;ENDCHR - search character
., a911  a5 08    lda $08
., a913  a6 07    ldx $07
., a915  85 07    sta $07
., a917  86 08    stx $08
., a919  b1 7a    lda ($7a),y   ;scan text
., a91b  f0 e8    beq $a905     ;terminator - EOL
., a91d  c5 08    cmp $08
., a91f  f0 e4    beq $a905     ;found search character
., a921  c8       iny
., a922  c9 22    cmp #$22      ;ASCII quotes?
., a924  d0 f3    bne $a919     ;continue search
., a926  f0 e9    beq $a911
```

43304 IF: PERFORM IF

The expression in text is evaluated, then the next statement is checked for either GOTO (#89) or THEN (#A7). If it is neither of these, then ?SYNTAX error. The result of IF is in FACEXP. A zero result indicates FALSE and non-zero indicates TRUE. For TRUE, the next statement is executed. For FALSE, the next line is sought.

```
., a928  20 9e ad jsr $ad9e     ;evaluate expression in text
., a92b  20 79 00 jsr $0079     ;CHRGOT
., a92e  c9 89    cmp #$89      ;token GOTO?
., a930  f0 05    beq $a937
., a932  a9 a7    lda #$a7      ;token THEN
., a934  20 ff ae jsr $aeff     ;confirm character in (A)
., a937  a5 61    lda $61       ;FACEXP - fac#1 exponent
., a939  d0 05    bne $a940     ;result TRUE - do next
                                statement
```

43323 REM: PERFORM REM

This routine searches for the next line and then performs DATA. The remainder of the routine is concerned with processing a true result from the IF routine. An ASCII number is tested for and if found, GOTO performed. If not,

the next statement is executed.

```
., a93b  20 09 a9 jsr $a909     ;search for next line
., a93e  f0 bb    beq $a8fb     ;do DATA
., a940  20 79 00 jsr $0079     ;CHRGOT
., a943  b0 03    bcs $a948     ;not numeric
., a945  4c a0 a8 jmp $a8a0     ;do GOTO
., a948  4c ed a7 jmp $a7ed     ;interpret and execute
                                 keyword
```

43339 ONGOTO: PERFORM ON

The argument is evaluated into fac#1 and then GOSUB or GOTO tested for. The variable is decremented to zero, working through the list of line numbers and testing for a comma between each. When the argument reaches zero, the line number reached is executed. If a comma is not found, then the routine 'drops off' onto the next line.

```
., a94b  20 9e b7 jsr $b79e     ;evaluate expression to 1
                                 byte in (X)
., a94e  48       pha
., a94f  c9 8d    cmp #$8d      ;token GOSUB?
., a951  f0 04    beq $a957
., a953  c9 89    cmp #$89      ;token GOTO?
., a955  d0 91    bne $a8e8     ;?SYNTAX error
., a957  c6 65    dec $65       ;FACHO4 - fac#1 mantissa
., a959  d0 04    bne $a95f
., a95b  68       pla
., a95c  4c ef a7 jmp $a7ef     ;interpret and execute
                                 command
., a95f  20 73 00 jsr $0073     ;CHRGET
., a962  20 6b a9 jsr $a96b     ;fetch line into LINNUM
., a965  c9 2c    cmp #$2c      ;ASCII comma?
., a967  f0 ee    beq $a957     ;read next number
., a969  68       pla
., a96a  60       rts
```

43371 LINGET: FETCH LINNUM FROM BASIC

This routine takes an ASCII numeral from text and converts it into a 2 byte integer. LINNUM is set to a default value of $0000. If the character returned from CHRGET is non-numeric then the default is returned. Each digit is multiplied by 10 and added to the next until the first non-numeric character. On exit, (A) holds the next byte of text, and TXTPTR has been updated by CHRGET.

```
., a96b  a2 00    ldx #$00      ;set default location in
```

```
., a96d  86 14    stx $14       ;LINNUM - integer value
., a96f  86 15    stx $15
., a971  b0 f7    bcs $a96a     ;not numeric - RTS
., a973  e9 2f    sbc #$2f      ;gives number value
., a975  85 07    sta $07       ;CHARAC - store for new digit
., a977  a5 15    lda $15       ;LINNUM
., a979  85 22    sta $22       ;INDEX1 - temp
., a97b  c9 19    cmp #$19
., a97d  b0 d4    bcs $a953
., a97f  a5 14    lda $14       ;process digit - * 10
., a981  0a       asl
., a982  26 22    rol $22
., a984  0a       asl
., a985  26 22    rol $22
., a987  65 14    adc $14
., a989  85 14    sta $14
., a98b  a5 22    lda $22
., a98d  65 15    adc $15
., a98f  85 15    sta $15
., a991  06 14    asl $14
., a993  26 15    rol $15
., a995  a5 14    lda $14
., a997  65 07    adc $07       ;add on next digit
., a999  85 14    sta $14
., a99b  90 02    bcc $a99f
., a99d  e6 15    inc $15
., a99f  20 73 00 jsr $0073     ;CHRGET
., a9a2  4c 71 a9 jmp $a971     ;process next digit
```

43429 LET: PERFORM LET

The variable in text is identified and created if it doesn't already exist. The token = (#B2) is confirmed and the following expression evaluated. The result is assigned to the variable according to its type. LET is called automatically whenever a variable assignment isencountered (eg. A=5).

```
., a9a5  20 8b b0 jsr $b08b     ;identify variable in text
., a9a8  85 49    sta $49       ;FORPNT - pointer to variable
., a9aa  84 4a    sty $4a
., a9ac  a9 b2    lda #$b2      ;token =
., a9ae  20 ff ae jsr $aeff     ;confirm character in (A)
., a9b1  a5 0e    lda $0e       ;push INTFLG - data type
., a9b3  48       pha
., a9b4  a5 0d    lda $0d       ;push VALTYP - data type
., a9b6  48       pha
., a9b7  20 9e ad jsr $ad9e     ;evaluate expression into
                                fac#1
```

```
., a9ba  68        pla
., a9bb  2a        rol
., a9bc  20 90 ad jsr $ad90    ;confirm variable type = data
                               type
., a9bf  d0 18     bne $a9d9    ;assign string
., a9c1  68        pla
., a9c2  10 12     bpl $a9d6    ;assign flpt
```

43460 PUTINT: ASSIGN INTEGER

This routine assigns an integer variable. FAC#1 is rounded and converted into a positive integer. The two integer bytes are then stored in RAM.

```
., a9c4  20 1b bc jsr $bc1b    ;round fac#1
., a9c7  20 bf b1 jsr $b1bf    ;convert fac#1 to integer in
                               ($64)
., a9ca  a0 00     ldy #$00
., a9cc  a5 64     lda $64      ;FACHO3 - integer value
., a9ce  91 49     sta ($49),y  ;store in address at FORPNT
., a9d0  c8        iny
., a9d1  a5 65     lda $65      ;FACHO4
., a9d3  91 49     sta ($49),y
., a9d5  60        rts
```

43478 PTFLPT: ASSIGN FLOATING POINT

This routine assigns a flpt variable. It stores the whole of fac#1 in RAM.

```
., a9d6  4c d0 bb jmp $bbd0    ;store fac#1 in high RAM
```

43481 PUTSTR: ASSIGN STRING

This routine assigns a string variable. It sets up the string descriptor and stores the string in high RAM.

```
., a9d9  68        pla
., a9da  a4 4a     ldy $4a      ;>FORPNT - pointer to
                               variable data
., a9dc  c0 bf     cpy #$bf
., a9de  d0 4c     bne $aa2c    ;store string descriptor in
                               high RAM
., a9e0  20 a6 b6 jsr $b6a6    ;do string housekeeping
```

43491 PUTTIM: ASSIGN TI$

This routine assigns a value to TI$. The length of the string is in (A). For TI$ this must be 6 characters or

?ILLEGAL QUANTITY error. FAC#1 is zeroed, then each digit is checked to be numeric, placed in fac#1, multiplied by 10 and added to the next. At the end of this process, fac#1 is converted into an integer and used to set the clock by the KERNAL routine.

```
., a9e3  c9 06     cmp #$06      ;string must be 6 characters
                                  long
., a9e5  d0 3d     bne $aa24     ;or ?ILLEGAL QUANTITY error
., a9e7  a0 00     ldy #$00
., a9e9  84 61     sty $61
., a9eb  84 66     sty $66
., a9ed  84 71     sty $71       ;counter for 6 bytes of
                                  string
., a9ef  20 1d aa  jsr $aa1d     ;get character in string
., a9f2  20 e2 ba  jsr $bae2     ;multiply fac#1 by 10
., a9f5  e6 71     inc $71
., a9f7  a4 71     ldy $71
., a9f9  20 1d aa  jsr $aa1d     ;get character in string
., a9fc  20 0c bc  jsr $bc0c     ;copy fac#1 into fac#2
., a9ff  aa        tax
., aa00  f0 05     beq $aa07
., aa02  e8        inx
., aa03  8a        txa
., aa04  20 ed ba  jsr $baed
., aa07  a4 71     ldy $71
., aa09  c8        iny
., aa0a  c0 06     cpy #$06      ;6th character yet?
., aa0c  d0 df     bne $a9ed     ;no -  do next character
., aa0e  20 e2 ba  jsr $bae2     ;multiply fac#1 by 10
., aa11  20 9b bc  jsr $bc9b     ;convert fac#1 to 4 byte
                                  integer
., aa14  a6 64     ldx $64       ;time mid byte
., aa16  a4 63     ldy $63       ;time low byte
., aa18  a5 65     lda $65       ;time high byte
., aa1a  4c db ff  jmp $ffdb     ;SETTIM - set real time clock
., aa1d  b1 22     lda ($22),y   ;get string character
., aa1f  20 80 00  jsr $0080     ;confirm numeric
., aa22  90 03     bcc $aa27
., aa24  4c 48 b2  jmp $b248     ;?ILLEGAL QUANTITY error
., aa27  e9 2f     sbc #$2f
., aa29  4c 7e bd  jmp $bd7e     ;convert ASCII string to flpt
```

43564 GETSPT: ADD ASCII DIGIT TO FAC#1

This is used by the previous routine to add a digit to fac#1. The string descriptors are checked, setting up the string in high RAM if necessary. The contents of the string are then taken and added to fac#1.

```
.,  aa2c  a0 02     ldy #$02
.,  aa2e  b1 64     lda ($64),y
.,  aa30  c5 34     cmp $34       ;>FRETOP - bottom of strings
.,  aa32  90 17     bcc $aa4b
.,  aa34  d0 07     bne $aa3d
.,  aa36  88        dey
.,  aa37  b1 64     lda ($64),y
.,  aa39  c5 33     cmp $33       ;<FRETOP
.,  aa3b  90 0e     bcc $aa4b
.,  aa3d  a4 65     ldy $65       ;FACHO4 - fac#1 mantissa
.,  aa3f  c4 2e     cpy $2e       ;>VARTAB -start of variables
.,  aa41  90 08     bcc $aa4b
.,  aa43  d0 0d     bne $aa52
.,  aa45  a5 64     lda $64       ;FACHO3
.,  aa47  c5 2d     cmp $2d       ;<VARTAB
.,  aa49  b0 07     bcs $aa52
.,  aa4b  a5 64     lda $64       ;FACHO3
.,  aa4d  a4 65     ldy $65       ;FACHO4
.,  aa4f  4c 68 aa  jmp $aa68     ;descriptor is good - add
                                  digit
.,  aa52  a0 00     ldy #$00
.,  aa54  b1 64     lda ($64),y
.,  aa56  20 75 b4  jsr $b475     ;allocate string pointers
.,  aa59  a5 50     lda $50       ;($50) points to ASCII digit
.,  aa5b  a4 51     ldy $51
.,  aa5d  85 6f     sta $6f       ;pointer for next subroutine
.,  aa5f  84 70     sty $70       ;FACOV - fac#1 rounding byte
.,  aa61  20 7a b6  jsr $b67a     ;store string in high RAM
.,  aa64  a9 61     lda #$61
.,  aa66  a0 00     ldy #$00
.,  aa68  85 50     sta $50       ;($50) points to digit
.,  aa6a  84 51     sty $51
.,  aa6c  20 db b6  jsr $b6db     ;update descriptor stack
                                  pointer
.,  aa6f  a0 00     ldy #$00
.,  aa71  b1 50     lda ($50),y   ;add digit to fac#1
.,  aa73  91 49     sta ($49),y
.,  aa75  c8        iny
.,  aa76  b1 50     lda ($50),y
.,  aa78  91 49     sta ($49),y
.,  aa7a  c8        iny
.,  aa7b  b1 50     lda ($50),y
.,  aa7d  91 49     sta ($49),y
.,  aa7f  60        rts
```

<u>43648 PRINTN: PERFORM PRINT#</u>

This routine performs CMD, then 'UNLISTENs' the serial port and restores default I/O. Note that both PRINT# and CMD

have identical syntax. ie. CMD4,"hello" is valid, however only PRINT# closes the output channel on completion.

```
., aa80  20 86 aa jsr $aa86     ;do CMD
., aa83  4c b5 ab jmp $abb5     ;restore default I/O channels
```

43654 CMD: PERFORM CMD

If the parameter taken from text is not a terminator then it is checked to be a comma. The output device is set up and the main PRINT routine entered.

```
., aa86  20 9e b7 jsr $b79e     ;evaluate text to 1 byte in
                                (X)
., aa89  f0 05    beq $aa90     ;no parameter in text
., aa8b  a9 2c    lda #$2c      ;ASCII comma
., aa8d  20 ff ae jsr $aeff     ;confirm character in (A)
., aa90  08       php
., aa91  86 13    stx $13       ;output channel
., aa93  20 18 e1 jsr $e118     ;set up for output
., aa96  28       plp
., aa97  4c a0 aa jmp $aaa0     ;do PRINT
```

43674 STRDON: PRINT STRING FROM MEMORY

This is a subsection of the main PRINT routine. Its purpose is to output a string pointed to by ($64). The string length must be in (A).

```
., aa9a  20 21 ab jsr $ab21     ;output string pointed to by
                                ($64)
., aa9d  20 79 00 jsr $0079     ;CHRGOT
```

43680 PRINT: PERFORM PRINT

This is the main PRINT routine. If there are no parameters a CR/LF is output. TAB(, SPC(comma and semicolon are checked and dealt with. If none of these is found then text is evaluated and variables output via the KERNAL CHROUT routine.

```
., aaa0  f0 35    beq $aad7     ;no parameter - print CR/LF
., aaa2  f0 43    beq $aae7     ;end of PRINT - RTS
., aaa4  c9 a3    cmp #$a3      ;token TAB(?
., aaa6  f0 50    beq $aaf8
., aaa8  c9 a6    cmp #$a6      ;token SPC(?
., aaaa  18       clc
., aaab  f0 4b    beq $aaf8
., aaad  c9 2c    cmp #$2c      ;ASCII comma?
```

```
., aaaf  f0 37     beq $aae8
., aab1  c9 3b     cmp #$3b       ;ASCII semicolon?
., aab3  f0 5e     beq $ab13      ;get next parameter & reenter
                                  at $AAA2
., aab5  20 9e ad jsr $ad9e      ;evaluate expression in text
```

43704 VAROP: OUTPUT VARIABLE

The variable type is examined. If it is a string then this is output. If it is a number then it is converted into an ASCII string and output. A space is output and any further PRINT parameters dealt with.

```
., aab8  24 0d     bit $0d        ;VALTYP - string or number?
., aaba  30 de     bmi $aa9a      ;print string
., aabc  20 dd bd jsr $bddd      ;convert fac#1 to string
., aabf  20 87 b4 jsr $b487      ;create string descriptor
., aac2  20 21 ab jsr $ab21      ;output string pointed to by
                                  ($64)
., aac5  20 3b ab jsr $ab3b      ;output cursor right or space
., aac8  d0 d3     bne $aa9d      ;re-enter PRINT
., aaca  a9 00     lda #$00
., aacc  9d 00 02 sta $0200,x    ;input buffer
., aacf  a2 ff     ldx #$ff
., aad1  a0 01     ldy #$01
., aad3  a5 13     lda $13        ;output channel
., aad5  d0 10     bne $aae7      ;channel is open - RTS
```

43735 CRDO: OUTPUT CR/LF

A carriage return is output and if the channel >127 ie location $13 >#7F then a line feed is also output.

```
., aad7  a9 0d     lda #$0d       ;carriage return
., aad9  20 47 ab jsr $ab47      ;output character in (A)
., aadc  24 13     bit $13        ;output channel
., aade  10 05     bpl $aae5
., aae0  a9 0a     lda #$0a       ;line feed
., aae2  20 47 ab jsr $ab47      ;output character in (A)
., aae5  49 ff     eor #$ff
., aae7  60        rts
```

43752 COMPRT: HANDLE COMMA, TAB(, SPC(

The routine first handles COMMA, reading the cursor X-Y position then flagging the next 10 column TAB position. The entry point at $AAF8 deals with TAB(and SPC(. Having read the cursor position, a right bracket is checked for. The two must be distinguished here since SPC(works from the

current cursor position and TAB(from the start of the line. The cursor is advanced and then PRINT is reentered.

```
., aae8  38        sec
., aae9  20 f0 ff  jsr $fff0     ;PLOT - read cursor X-Y
                                  position
., aaec  98        tya
., aaed  38        sec
., aaee  e9 0a     sbc #$0a      ;flag next TAB position
., aaf0  b0 fc     bcs $aaee
., aaf2  49 ff     eor #$ff
., aaf4  69 01     adc #$01
., aaf6  d0 16     bne $ab0e
., aaf8  08        php
., aaf9  38        sec
., aafa  20 f0 ff  jsr $fff0     ;PLOT
., aafd  84 09     sty $09
., aaff  20 9b b7  jsr $b79b     ;evaluate expression to 1
                                  byte in (X)
., ab02  c9 29     cmp #$29      ;ASCII )?
., ab04  d0 59     bne $ab5f     ;?SYNTAX error
., ab06  28        plp
., ab07  90 06     bcc $ab0f     ;SPC - cursor right from
                                  current posn.
., ab09  8a        txa
., ab0a  e5 09     sbc $09       ;set column to start of line
., ab0c  90 05     bcc $ab13
., ab0e  aa        tax
., ab0f  e8        inx
., ab10  ca        dex
., ab11  d0 06     bne $ab19     ;do TAB/SPC
., ab13  20 73 00  jsr $0073     ;CHRGET - TAB finished
., ab16  4c a2 aa  jmp $aaa2     ;reenter PRINT
., ab19  20 3b ab  jsr $ab3b     ;output cursor right
., ab1c  d0 f2     bne $ab10     ;next TAB
```

43806 STROUT: OUTPUT STRING

There are several entry points to this routine depending on how the string is pointed to. The string descriptor is set up and each character in the string is output until the terminator is reached.

```
., ab1e  20 87 b4  jsr $b487     ;create descriptor for string
                                  at (A/Y)
., ab21  20 a6 b6  jsr $b6a6     ;do string housekeeping
., ab24  aa        tax           ;length is in (A), INDEX1 is
                                  pointer
., ab25  a0 00     ldy #$00
```

```
., ab27  e8        inx
., ab28  ca        dex
., ab29  f0 bc     beq $aae7      ;finished - RTS
., ab2b  b1 22     lda ($22),y    ;get character from string
., ab2d  20 47 ab  jsr $ab47      ;output character in (A)
., ab30  c8        iny
., ab31  c9 0d     cmp #$0d       ;carriage return?
., ab33  d0 f3     bne $ab28      ;no - output next character
., ab35  20 e5 aa  jsr $aae5
., ab38  4c 28 ab  jmp $ab28
```

43835 OUTSPC: OUTPUT FORMAT CHARACTER

If the output channel is zero (ie the screen) then a cursor right is output, and if not then a space. There is also a provision to output a question mark.

```
., ab3b  a5 13     lda $13        ;output channel
., ab3d  f0 03     beq $ab42      ;screen - do cursor right
., ab3f  a9 20     lda #$20       ;set for space
., ab41  2c a9 1d  bit $1da9      ;mask - set for cursor right
., ab44  2c a9 3f  bit $3fa9      ;mask - set for question mark
., ab47  20 0c e1  jsr $e10c      ;output character in (A)
., ab4a  29 ff     and #$ff
., ab4c  60        rts
```

43853 DOAGIN: HANDLE BAD DATA

INPFLG is tested for mode: #00=INPUT, #40=GET, #98=READ. These are separated out and dealt with. READ takes DATLIN and then joining with GET, stores it in CURLIN. ?SYNTAX error is then performed. INPUT checks for open channels. If none are open then ?REDO FROM START and execution continues at the last statement. If a channel is open then ?FILE DATA error is performed in the normal way.

```
., ab4d  a5 11     lda $11        ;INPFLG - flag for
                                  INPUT/GET/READ
., ab4f  f0 11     beq $ab62      ;do for INPUT (#00)
., ab51  30 04     bmi $ab57      ;do for READ (#98)
., ab53  a0 ff     ldy #$ff       ;GET (#40)
., ab55  d0 04     bne $ab5b
., ab57  a5 3f     lda $3f        ;DATLIN - DATA line number
., ab59  a4 40     ldy $40
., ab5b  85 39     sta $39        ;CURLIN - current line number
., ab5d  84 3a     sty $3a
., ab5f  4c 08 af  jmp $af08      ;?SYNTAX error
., ab62  a5 13     lda $13        ;input channel
., ab64  f0 05     beq $ab6b
```

```
., ab66  a2 18     ldx #$18     ;flag ?FILE DATA
., ab68  4c 37 a4 jmp $a437     ;do error
., ab6b  a9 0c     lda #$0c
., ab6d  a0 ad     ldy #$ad     ;$AD0C = message ?REDO FROM
                                START
., ab6f  20 1e ab jsr $ab1e     ;output string at (A/Y)
., ab72  a5 3d     lda $3d      ;OLDTXT - CONT pointer
., ab74  a4 3e     ldy $3e
., ab76  85 7a     sta $7a      ;TXTPTR - set to previous
                                statement
., ab78  84 7b     sty $7b
., ab7a  60        rts
```

43899 GET: PERFORM GET

GET will only operate in program mode. If '# is found after the token, ie GET#, then the file number is input, comma checked for and the device set up for input. The system input buffer is set up for a single character and the universal 'read' routine entered. Finally, any open channels are restored.

```
., ab7b  20 a6 b3 jsr $b3a6     ;check direct mode
., ab7e  c9 23     cmp #$23     ;ASCII #?
., ab80  d0 10     bne $ab92
., ab82  20 73 00 jsr $0073     ;CHRGET
., ab85  20 9e b7 jsr $b79e     ;evaluate text to 1 byte in
                                (X)
., ab88  a9 2c     lda #$2c     ;ASCII comma
., ab8a  20 ff ae jsr $aeff     ;confirm character in (A)
., ab8d  86 13     stx $13      ;input channel - holds file
                                number
., ab8f  20 1e e1 jsr $e11e     ;set up for input
., ab92  a2 01     ldx #$01
., ab94  a0 02     ldy #$02
., ab96  a9 00     lda #$00
., ab98  8d 01 02 sta $0201     ;input buffer
., ab9b  a9 40     lda #$40     ;flag GET for later
., ab9d  20 0f ac jsr $ac0f     ;do READ
., aba0  a6 13     ldx $13      ;input channel
., aba2  d0 13     bne $abb7    ;close channel
., aba4  60        rts
```

43941 INPUTN: PERFORM INPUT#

The file number is obtained and a comma confirmed, then the device is set up for input and the main INPUT routine performed. Finally the channels used are restored to their defaults.

```
.,  aba5  20 9e b7 jsr $b79e      ;evaluate text to 1 byte in
                                  (X)
.,  aba8  a9 2c    lda #$2c       ;ASCII comma
.,  abaa  20 ff ae jsr $aeff      ;confirm character in (A)
.,  abad  86 13    stx $13        ;input channel - holds file
                                  number
.,  abaf  20 1e e1 jsr $e11e      ;set up for input
.,  abb2  20 ce ab jsr $abce      ;do INPUT
.,  abb5  a5 13    lda $13        ;input channel
.,  abb7  20 cc ff jsr $ffcc      ;CLRCHN - close I/O channels
.,  abba  a2 00    ldx #$00
.,  abbc  86 13    stx $13        ;set channel = 0
.,  abbe  60       rts
```

43967 INPUT: PERFORM INPUT

If quotes are found after the token, then the string within them is output. A semicolon is then sought at the end of the string. An input prompt is printed and the data input to the system input buffer. If the input channel is not zero then the I/O status word is read. The routine exits via DATA.

```
.,  abbf  c9 22    cmp #$22       ;ASCII quotes?
.,  abc1  d0 0b    bne $abce
.,  abc3  20 bd ae jsr $aebd      ;set up string in quotes
.,  abc6  a9 3b    lda #$3b       ;ASCII semicolon
.,  abc8  20 ff ae jsr $aeff      ;confirm character in (A)
.,  abcb  20 21 ab jsr $ab21      ;output string at ($64)
.,  abce  20 a6 b3 jsr $b3a6      ;check direct mode
.,  abd1  a9 2c    lda #$2c       ;ASCII comma
.,  abd3  8d ff 01 sta $01ff
.,  abd6  20 f9 ab jsr $abf9      ;do prompt and input
.,  abd9  a5 13    lda $13        ;input channel
.,  abdb  f0 0d    beq $abea      ;read input buffer
.,  abdd  20 b7 ff jsr $ffb7      ;READST - read I/O status
                                  word
.,  abe0  29 02    and #$02
.,  abe2  f0 06    beq $abea      ;read input buffer
.,  abe4  20 b5 ab jsr $abb5      ;close input channel
.,  abe7  4c f8 a8 jmp $a8f8      ;do DATA
```

44010 BUFFUL: READ INPUT BUFFER

This routine looks for a null input. If return was pressed with the buffer empty then DATA is called to skip to the next statement. On old PETs, END was called, thereby crashing the program.

```
., abea  ad 00 02 lda $0200    ;input buffer
., abed  d0 1e    bne $ac0d    ;do READ
., abef  a5 13    lda $13      ;input channel
., abf1  d0 e3    bne $abd6    ;not zero - input again
., abf3  20 06 a9 jsr $a906    ;scan for next statement
., abf6  4c fb a8 jmp $a8fb    ;do DATA
```

44025 QINLIN: DO INPUT PROMPT

If INPUT is from the screen, then a question mark and cursor right are output. The input buffer is filled from the keyboard until carriage return, when the routine exits.

```
., abf9  a5 13    lda $13      ;input channel
., abfb  d0 06    bne $ac03    ;channel - input data to
                               buffer
., abfd  20 45 ab jsr $ab45    ;output question mark
., ac00  20 3b ab jsr $ab3b    ;output cursor right
., ac03  4c 60 a5 jmp $a560    ;input data to buffer
```

44038 READ: PERFORM READ

This is a universal routine also used by GET and INPUT. INPFLG is set according to the function being performed. INPPTR is set to the location to be read from (ie. DATA statements or the input buffer). The variable to be assigned is identified and set up as needed. Finally the GENERAL PURPOSE READ ROUTINE is performed.

```
., ac06  a6 41    ldx $41      ;DATPTR - pointer to DATA
                               item
., ac08  a4 42    ldy $42
., ac0a  a9 98    lda #$98     ;flag READ
., ac0c  2c a9 00 bit $00a9    ;mask - flag INPUT
., ac0f  85 11    sta $11      ;INPFLG - flag for
                               GET/INPUT/READ
., ac11  86 43    stx $43      ;INPPTR - input vector
., ac13  84 44    sty $44
., ac15  20 8b b0 jsr $b08b    ;identify variable
., ac18  85 49    sta $49      ;FORPNT - pointer to variable
., ac1a  84 4a    sty $4a
., ac1c  a5 7a    lda $7a      ;TXTPTR
., ac1e  a4 7b    ldy $7b
., ac20  85 4b    sta $4b      ;temp area
., ac22  84 4c    sty $4c
., ac24  a6 43    ldx $43      ;INPPTR
., ac26  a4 44    ldy $44
., ac28  86 7a    stx $7a      ;TXTPTR
```

```
., ac2a  84 7b     sty $7b
., ac2c  20 79 00 jsr $0079     ;CHRGOT - get input byte
., ac2f  d0 20     bne $ac51     ;do general purpose read
., ac31  24 11     bit $11       ;INPFLG
., ac33  50 0c     bvc $ac41
```

44085 RDGET: GENERAL PURPOSE READ ROUTINE

This routine is in three major sections - GET, INPUT and READ. The first section, GET, loads one character into the input buffer, then passes it on to the second section, INPUT.

```
., ac35  20 24 e1 jsr $e124     ;get 1 character
., ac38  8d 00 02 sta $0200     ;input buffer
., ac3b  a2 ff     ldx #$ff
., ac3d  a0 01     ldy #$01
., ac3f  d0 0c     bne $ac4d     ;process character
., ac41  30 75     bmi $acb8
```

If there are no channels open then a question mark is output and DO INPUT PROMPT ($ABF9) is performed. Data type is checked and if it is string, then the string is set up and assigned. If it is numeric, then it is converted from ASCII, and LET called to assign the variable. If neither a comma nor a terminator is found at the end of this, then the HANDLE BAD DATA routine is entered.

```
., ac43  a5 13     lda $13       ;input channel
., ac45  d0 03     bne $ac4a
., ac47  20 45 ab jsr $ab45     ;output question mark
., ac4a  20 f9 ab jsr $abf9     ;prompt and input to buffer
., ac4d  86 7a     stx $7a       ;TXTPTR
., ac4f  84 7b     sty $7b
., ac51  20 73 00 jsr $0073     ;CHRGET
., ac54  24 0d     bit $0d       ;VALTYP - string or numeric?
., ac56  10 31     bpl $ac89     ;do numeric
., ac58  24 11     bit $11       ;INPFLG - flag for
                                 INPUT/GET/READ
., ac5a  50 09     bvc $ac65
., ac5c  e8        inx
., ac5d  86 7a     stx $7a       ;TXTPTR
., ac5f  a9 00     lda #$00
., ac61  85 07     sta $07       ;CHARAC - search character
., ac63  f0 0c     beq $ac71
., ac65  85 07     sta $07       ;CHARAC
., ac67  c9 22     cmp #$22      ;ASCII quotes?
., ac69  f0 07     beq $ac72
., ac6b  a9 3a     lda #$3a      ;ASCII colon?
```

```
.,  ac6d  85 07     sta $07       ;CHARAC
.,  ac6f  a9 2c     lda #$2c      ;ASCII comma?
.,  ac71  18        clc
.,  ac72  85 08     sta $08       ;ENDCHR - scan for end quotes
.,  ac74  a5 7a     lda $7a       ;TXTPTR - points to string
.,  ac76  a4 7b     ldy $7b
.,  ac78  69 00     adc #$00
.,  ac7a  90 01     bcc $ac7d
.,  ac7c  c8        iny
.,  ac7d  20 8d b4  jsr $b48d     ;create string descriptor
.,  ac80  20 e2 b7  jsr $b7e2     ;set TXTPTR
.,  ac83  20 da a9  jsr $a9da     ;store string in RAM
.,  ac86  4c 91 ac  jmp $ac91
.,  ac89  20 f3 bc  jsr $bcf3     ;read number into fac#1
.,  ac8c  a5 0e     lda $0e       ;INTFLG - data type
.,  ac8e  20 c2 a9  jsr $a9c2     ;assign variable
.,  ac91  20 79 00  jsr $0079     ;CHRGOT
.,  ac94  f0 07     beq $ac9d     ;terminator
.,  ac96  c9 2c     cmp #$2c      ;ASCII comma?
.,  ac98  f0 03     beq $ac9d
.,  ac9a  4c 4d ab  jmp $ab4d     ;handle bad data
.,  ac9d  a5 7a     lda $7a       ;TXTPTR
.,  ac9f  a4 7b     ldy $7b
.,  aca1  85 43     sta $43       ;INPPTR - input vector
.,  aca3  84 44     sty $44
.,  aca5  a5 4b     lda $4b       ;temp store
.,  aca7  a4 4c     ldy $4c
.,  aca9  85 7a     sta $7a       ;TXTPTR
.,  acab  84 7b     sty $7b
.,  acad  20 79 00  jsr $0079     ;CHRGOT
.,  acb0  f0 2d     beq $acdf     ;terminate routine
.,  acb2  20 fd ae  jsr $aefd     ;confirm comma
.,  acb5  4c 15 ac  jmp $ac15     ;do READ
```

The third section, READ, searches for the next statement, then works through the line searching for a DATA statement. The search continues until either one is found or the end of the program is reached. Once found, DATA is processed by the INPUT section.

```
.,  acb8  20 06 a9  jsr $a906     ;search for next statement
.,  acbb  c8        iny
.,  acbc  aa        tax
.,  acbd  d0 12     bne $acd1     ;not EOL
.,  acbf  a2 0d     ldx #$0d      ;flag ?OUT OF DATA
.,  acc1  c8        iny
.,  acc2  b1 7a     lda ($7a),y   ;read link pointer
.,  acc4  f0 6c     beq $ad32     ;do error
.,  acc6  c8        iny
```

```
., acc7  b1 7a     lda ($7a),y  ;read line number
., acc9  85 3f     sta $3f      ;DATLIN - holds DATA line
                                 number
., accb  c8        iny
., accc  b1 7a     lda ($7a),y
., acce  c8        iny
., accf  85 40     sta $40
., acd1  20 fb a8  jsr $a8fb    ;do DATA
., acd4  20 79 00  jsr $0079    ;CHRGOT
., acd7  aa        tax
., acd8  e0 83     cpx #$83     ;token DATA?
., acda  d0 dc     bne $acb8    ;no - next statement
., acdc  4c 51 ac  jmp $ac51    ;read data from text
```

This last section is concerned with terminating the routine. If READ then the DATA pointer is set to the next line. If there is any extra data in the input buffer then ?EXTRA IGNORED is output, unless an input channel is open, when the message is suppressed.

```
., acdf  a5 43     lda $43      ;INPPTR - input vector
., ace1  a4 44     ldy $44
., ace3  a6 11     ldx $11      ;INPFLG
., ace5  10 03     bpl $acea
., ace7  4c 27 a8  jmp $a827    ;set DATPTR
., acea  a0 00     ldy #$00
., acec  b1 43     lda ($43),y  ;more data?
., acee  f0 0b     beq $acfb
., acf0  a5 13     lda $13      ;input channel
., acf2  d0 07     bne $acfb    ;file open - suppress error
                                 message
., acf4  a9 fc     lda #$fc
., acf6  a0 ac     ldy #$ac     ;$ACFC points to  ?EXTRA
                                 IGNORED
., acf8  4c 1e ab  jmp $ab1e    ;output string at (A/Y)
., acfb  60        rts
```

44284 EXINT: INPUT ERROR MESSAGES

The messages ?EXTRA IGNORED and ?REDO FROM START are stored here. Both messages are followed by a carriage return, and #00 is used as a terminator for both messages.

```
.:acfc 3f 45 58 54 52 41 20 49  ?extra i
.:ad04 47 4e 4f 52 45 44 0d 00  gnored
.:ad0c 3f 52 45 44 4f 20 46 52  ?redo fr
.:ad14 4f 4d 20 53 54 41 52 54  om start
.:ad1c 0d 00
```

44318 NEXT: PERFORM NEXT

If there are no parameters, then FORPNT is set to $0000, otherwise the variable is sought, identified, and put into FORPNT. A FOR entry is looked for on the stack and if it is not found, or if it is the wrong one, then ?NEXT WITHOUT FOR. The step value is added to the loop variable. These are compared and (A) set according to the result, which is then dealt with by the next routine.

```
., ad1e  d0 04     bne $ad24     ;NEXT has parameters
., ad20  a0 00     ldy #$00      ;set default value for FORPNT
., ad22  f0 03     beq $ad27
., ad24  20 8b b0  jsr $b08b     ;identify variable in text
., ad27  85 49     sta $49       ;FORPNT - pointer to loop
                                 variable
., ad29  84 4a     sty $4a
., ad2b  20 8a a3  jsr $a38a     ;find FOR entry on stack
., ad2e  f0 05     beq $ad35
., ad30  a2 0a     ldx #$0a      ;flag - ?NEXT WITHOUT FOR
., ad32  4c 37 a4  jmp $a437     ;do error
., ad35  9a        txs
., ad36  8a        txa
., ad37  18        clc
., ad38  69 04     adc #$04
., ad3a  48        pha
., ad3b  69 06     adc #$06
., ad3d  85 24     sta $24       ;INDEX2
., ad3f  68        pla
., ad40  a0 01     ldy #$01
., ad42  20 a2 bb  jsr $bba2     ;load fac#1 with flpt at
                                 (A/Y)
., ad45  ba        tsx
., ad46  bd 09 01  lda $0109,x
., ad49  85 66     sta $66       ;FACSGN - fac#1 sign
., ad4b  a5 49     lda $49       ;FORPNT
., ad4d  a4 4a     ldy $4a
., ad4f  20 67 b8  jsr $b867     ;add FORPNT to fac#1
., ad52  20 d0 bb  jsr $bbd0     ;put result in loop variable
., ad55  a0 01     ldy #$01
., ad57  20 5d bc  jsr $bc5d     ;compare result with fac#1
., ad5a  ba        tsx
., ad5b  38        sec
., ad5c  fd 09 01  sbc $0109,x
., ad5f  f0 17     beq $ad78
```

44385 DONEXT: CHECK VALID LOOP

A valid loop is assumed on entry, thus NEXT is performed

here. CURLIN and TXTPTR are recovered from the stack and a BASIC warm start performed. If the loop is finished, then the FOR entry is deleted from the stack, and if a comma is found, (ie from NEXT I,J) in the parameters, then NEXT is reentered from the top.

```
., ad61  bd 0f 01 lda $010f,x  ;recover CURLIN from stack
., ad64  85 39    sta $39
., ad66  bd 10 01 lda $0110,x
., ad69  85 3a    sta $3a
., ad6b  bd 12 01 lda $0112,x
., ad6e  85 7a    sta $7a      :recover TXTPTR from stack
., ad70  bd 11 01 lda $0111,x
., ad73  85 7b    sta $7b
., ad75  4c ae a7 jmp $a7ae    ;do BASIC warm start
., ad78  8a       txa
., ad79  69 11    adc #$11     ;delete FOR entry from stack
., ad7b  aa       tax
., ad7c  9a       txs
., ad7d  20 79 00 jsr $0079    ;CHRGOT
., ad80  c9 2c    cmp #$2c     ;ASCII comma?
., ad82  d0 f1    bne $ad75    ;no - do warm start
., ad84  20 73 00 jsr $0073    ;CHRGET
., ad87  20 24 ad jsr $ad24    ;reenter NEXT
```

44426 FRMNUM: CONFIRM RESULT

Firstly, the expression in text is evaluated into fac#1. The condition of the carry flag is used to determine which test is carried out. If carry is set, then string mode is confirmed. If carry is clear, then numeric mode is confirmed. In either case, if the test is failed, ?TYPE MISMATCH error results.

```
., ad8a  20 9e ad jsr $ad9e    ;evaluate expression in text
., ad8d  18       clc          ;flag = confirm numeric mode
., ad8e  24 38    bit $38      ;mask - SEC = confirm string
                               mode
., ad90  24 0d    bit $0d      ;VALTYP - data type (string
                               or numeric)
., ad92  30 03    bmi $ad97
., ad94  b0 03    bcs $ad99
., ad96  60       rts
., ad97  b0 fd    bcs $ad96
., ad99  a2 16    ldx #$16     ;flag ?TYPE MISMATCH
., ad9b  4c 37 a4 jmp $a437    ;do error
```

44446 FRMEVL: EVALUATE EXPRESSION IN TEXT

This is the main function keyword evaluation routine. The

expression in text, including all operators and functions, is evaluated into either a number in fac#1 or a string with pointers in ($64). Since this is a rather long routine, it is divided up into sections, each section being described individually. The first section decrements TXTPTR before entering the start of the first major recursive routine.

```
., ad9e  a6 7a     ldx $7a       ;TXTPTR
., ada0  d0 02     bne $ada4
., ada2  c6 7b     dec $7b       ;decrement TXTPTR
., ada4  c6 7a     dec $7a
., ada6  a2 00     ldx #$00
```

(A) and (X) are pushed onto the stack, which is then tested for two free bytes. The single term in text is then evaluated.

```
., ada8  24 48     bit $48       ;mask - PHA
., adaa  8a        txa
., adab  48        pha
., adac  a9 01     lda #$01
., adae  20 fb a3  jsr $a3fb     ;check stack for two free
                                 bytes
., adb1  20 83 ae  jsr $ae83     ;evaluate single term
., adb4  a9 00     lda #$00
., adb6  85 4d     sta $4d       ;operator code
```

This section tests for <, =, >. If not found, then the following section is performed, otherwise a code is stored in $4D and the section reentered.

```
., adb8  20 79 00  jsr $0079     ;CHRGOT
., adbb  38        sec
., adbc  e9 b1     sbc #$b1      ;test for <=>
., adbe  90 17     bcc $add7     ;not found - code too low
., adc0  c9 03     cmp #$03
., adc2  b0 13     bcs $add7     ;not found - code too high
., adc4  c9 01     cmp #$01
., adc6  2a        rol
., adc7  49 01     eor #$01
., adc9  45 4d     eor $4d       ;operator code
., adcb  c5 4d     cmp $4d
., adcd  90 61     bcc $ae30
., adcf  85 4d     sta $4d       ;operator code
., add1  20 73 00  jsr $0073     ;CHRGET
., add4  4c bb ad  jmp $adbb     ;reenter section
```

This section processes other operators not found by the

previous section. $4D is tested for <, =, > being found, and data type is checked. The operator priority code is retrieved from the operator vectors table. If this has a lower hierarchy than the current result on the stack, then the result is pulled from the stack into fac#2 and combined with fac#1. If the operator has a higher hierarchy value than the current result, then both the operator and fac#1 are pushed onto the stack. This section is recursive by operator precedence until the operator with highest priority is located, when control is passed to the next section.

```
., add7  a6 4d     ldx $4d       ;operator code
., add9  d0 2c     bne $ae07     ;already exists
., addb  b0 7b     bcs $ae58     ;pull fac#2 and end
., addd  69 07     adc #$07
., addf  90 77     bcc $ae58
., ade1  65 0d     adc $0d       ;VALTYP - data type
., ade3  d0 03     bne $ade8
., ade5  4c 3d b6  jmp $b63d     ;concatenate two strings
., ade8  69 ff     adc #$ff
., adea  85 22     sta $22       ;<INDEX1
., adec  0a        asl
., aded  65 22     adc $22       ;produces operator offset in
                                  table
., adef  a8        tay
., adf0  68        pla
., adf1  d9 80 a0  cmp $a080,y   ;operator priority code
., adf4  b0 67     bcs $ae5d     ;pull result from stack
., adf6  20 8d ad  jsr $ad8d     ;confirm numeric
., adf9  48        pha
., adfa  20 20 ae  jsr $ae20     ;push result and do operator
., adfd  68        pla
., adfe  a4 4b     ldy $4b
., ae00  10 17     bpl $ae19
., ae02  aa        tax
., ae03  f0 56     beq $ae5b     ;get FACEXP and end
., ae05  d0 5f     bne $ae66     ;pull fac#2
., ae07  46 0d     lsr $0d       ;VALTYP
., ae09  8a        txa
., ae0a  2a        rol
., ae0b  a6 7a     ldx $7a       ;TXTPTR
., ae0d  d0 02     bne $ae11
., ae0f  c6 7b     dec $7b       ;decrement TXTPTR
., ae11  c6 7a     dec $7a
., ae13  a0 1b     ldy #$1b
., ae15  85 4d     sta $4d       ;set operator code
., ae17  d0 d7     bne $adf0
., ae19  d9 80 a0  cmp $a080,y   ;operator priority
., ae1c  b0 48     bcs $ae66     ;pull fac#2
```

```
., ae1e  90 d9    bcc $adf9    ;reenter section
```

The operator address is taken from the table, pushed onto the stack and placed in INDEX1. FAC#1 is rounded and pushed onto the stack. The operator is then executed. It should be noted that the argument of the operator has already been evaluated into fac#1 by the major routine here before the operator itself is called.

```
., ae20  b9 82 a0 lda $a082,y  ;push operator vector
., ae23  48       pha
., ae24  b9 81 a0 lda $a081,y
., ae27  48       pha
., ae28  20 33 ae jsr $ae33    ;push fac#1 and do operator
., ae2b  a5 4d    lda $4d
., ae2d  4c a9 ad jmp $ada9    ;re-enter main routine at
                               start
., ae30  4c 08 af jmp $af08    ;?SYNTAX error
., ae33  a5 66    lda $66      ;FACSGN - fac#1 sign
., ae35  be 80 a0 ldx $a080,y
., ae38  a8       tay
., ae39  68       pla
., ae3a  85 22    sta $22      ;INDEX1 - operator vector
., ae3c  e6 22    inc $22
., ae3e  68       pla
., ae3f  85 23    sta $23
., ae41  98       tya
., ae42  48       pha
., ae43  20 1b bc jsr $bc1b    ;round fac#1
., ae46  a5 65    lda $65      ;push fac#1
., ae48  48       pha
., ae49  a5 64    lda $64
., ae4b  48       pha
., ae4c  a5 63    lda $63
., ae4e  48       pha
., ae4f  a5 62    lda $62
., ae51  48       pha
., ae52  a5 61    lda $61
., ae54  48       pha
., ae55  6c 22 00 jmp ($0022)  ;do operator
```

This final section confirms the result, then pulls fac#2 from the stack. Comparison is made with fac#1. On exit, (A) holds FACEXP.

```
., ae58  a0 ff    ldy #$ff
., ae5a  68       pla
., ae5b  f0 23    beq $ae80    ;get FACEXP and end
., ae5d  c9 64    cmp #$64
```

```
.,  ae5f   f0 03     beq $ae64
.,  ae61   20 8d ad  jsr $ad8d     ;conirm numeric mode
.,  ae64   84 4b     sty $4b
.,  ae66   68        pla
.,  ae67   4a        lsr
.,  ae68   85 12     sta $12       ;TANSGN - comparison result
.,  ae6a   68        pla
.,  ae6b   85 69     sta $69       ;pull ARGEXP - fac#2 exponent
.,  ae6d   68        pla
.,  ae6e   85 6a     sta $6a       ;pull fac#2 mantissa
.,  ae70   68        pla
.,  ae71   85 6b     sta $6b
.,  ae73   68        pla
.,  ae74   85 6c     sta $6c
.,  ae76   68        pla
.,  ae77   85 6d     sta $6d
.,  ae79   68        pla
.,  ae7a   85 6e     sta $6e       ;pull ARGSGN - fac#2 sign
.,  ae7c   45 66     eor $66       ;FACSGN - fac#1 sign
.,  ae7e   85 6f     sta $6f       ;ARISGN - sign comparison
                                   result
.,  ae80   a5 61     lda $61       ;FACEXP - fac#1 exponent
.,  ae82   60        rts
```

44675 EVAL: EVALUATE SINGLE TERM

This routine takes a single term from an expression in text and evaluates it to a flpt number in fac#1. Data type is set to numeric then CHRGET performed. If this points to an ASCII numeral, then this is converted to a number in fac#1. If it is a letter, then a variable is searched for. If it the token PI, then this value is set into fac#1. If it is none of these, then the expression handling is continued by the next routine.

```
.,  ae83   6c 0a 03 jmp ($030a)   ;vector IEVAL - points to
                                   $AE86
.,  ae86   a9 00     lda #$00
.,  ae88   85 0d     sta $0d       ;set VALTYP to numeric mode
.,  ae8a   20 73 00  jsr $0073     ;CHRGET
.,  ae8d   b0 03     bcs $ae92     ;not numeric character
.,  ae8f   4c f3 bc  jmp $bcf3     ;read number into fac#1
.,  ae92   20 13 b1  jsr $b113     ;is character a letter?
.,  ae95   90 03     bcc $ae9a     ;no
.,  ae97   4c 28 af  jmp $af28     ;find named variable
.,  ae9a   c9 ff     cmp #$ff      ;is it pi?
.,  ae9c   d0 0f     bne $aead     ;no - continue expression
.,  ae9e   a9 a8     lda #$a8
.,  aea0   a0 ae     ldy #$ae      ;$AEA8 = pi in flpt
```

```
.,  aea2   20 a2 bb jsr $bba2    ;load fac#1 with flpt at
                                 (A/Y)
.,  aea5   4c 73 00 jmp $0073    ;CHRGET
```

PIVAL: CONSTANT - PI

The value 3.1415965 in 5 byte flpt format.

```
.:aea8 82 49 0f da a1
```

44717 QDOT: CONTINUE EXPRESSION

This part of the evaluation routine handles non-variable terms. It tests for and handles the following - ASCII decimals, -, +, NOT, FN, SGN. Also, if quotes are found, the string within them is set up.

```
.,  aead   c9 2e    cmp #$2e     ;ASCII decimal?
.,  aeaf   f0 de    beq $ae8f    ;read number into fac#1
.,  aeb1   c9 ab    cmp #$ab     ;token -?
.,  aeb3   f0 58    beq $af0d    ;set up monadic minus
.,  aeb5   c9 aa    cmp #$aa     ;token +?
.,  aeb7   f0 d1    beq $ae8a    ;read number into fac#1
.,  aeb9   c9 22    cmp #$22     ;ASCII quotes?
.,  aebb   d0 0f    bne $aecc    ;NOT?
.,  aece   d0 13    bne $aee3
.,  aed0   a0 18    ldy #$18     ;vector offset for NOT
.,  aed2   d0 3b    bne $af0f    ;set up NOT function
.,  aed4   20 bf b1 jsr $b1bf    ;convert fac#1 to integer in
                                 ($64)
.,  aed7   a5 65    lda $65
.,  aed9   49 ff    eor #$ff     ;negate it
.,  aedb   a8       tay
.,  aedc   a5 64    lda $64
.,  aede   49 ff    eor #$ff
.,  aee0   4c 91 b3 jmp $b391    ;convert integer to flpt
.,  aee3   c9 a5    cmp #$a5     ;token FN?
.,  aee5   d0 03    bne $aeea
.,  aee7   4c f4 b3 jmp $b3f4    ;do FN
.,  aeea   c9 b4    cmp #$b4     ;token SGN?
.,  aeec   90 03    bcc $aef1    ;do expression in brackets
.,  aeee   4c a7 af jmp $afa7    ;identify and evaluate
                                 function
```

44785 PARCHK: EXPRESSION IN BRACKETS

A left bracket is confirmed, then the expression within evaluated by calling the FRMEVL ($AD9E) routine. Control finally drops through to the next routine to confirm a right

bracket.

```
., aef1  20 fa ae jsr $aefa    ;confirm left bracket
., aef4  20 9e ad jsr $ad9e    ;evaluate expression in text
```

44791 CHKCLS: CONFIRM CHARACTER

There are several entry points to confirm preset characters (left and right brackets and comma) and also for entry with the character to be confirmed in (A). If the character is not the same as that pointed to by TXTPTR, then ?SYNTAX error

```
., aef7  a9 29    lda #$29     ;ASCII )
., aef9  2c a9 28 bit $28a9    ;mask - ASCII (
., aefc  2c a9 2c bit $2ca9    ;mask - ASCII comma
., aeff  a0 00    ldy #$00
., af01  d1 7a    cmp ($7a),y  ;compare (A) with byte of
                               text
., af03  d0 03    bne $af08    ;different - ?SYNTAX error
., af05  4c 73 00 jmp $0073    ;CHRGET
```

44808 SYNERR: OUTPUT ?SYNTAX ERROR

This routine prints the message ?SYNTAX ERROR, and then restarts BASIC in direct mode.

```
., af08  a2 0b    ldx #$0b     ;flag ?SYNTAX
., af0a  4c 37 a4 jmp $a437    ;do error
```

44813 DOMIN: SET UP NOT FUNCTION

This routine sets up monadic minus or NOT for later evaluation. The vector offset for minus (#15) or that for NOT (#18) is placed in (Y), then the operator is executed.

```
., af0d  a0 15    ldy #$15     ;vector offset for minus
., af0f  68       pla
., af10  68       pla
., af11  4c fa ad jmp $adfa    ;push fac#1 and do operator
```

44820 RSVVAR: IDENTIFY RESERVED VARIABLE

If the variable pointed to by ($64) is either TI or ST, then the carry flag is set, otherwise it is cleared.

```
., af14  38       sec
., af15  a5 64    lda $64      ;pointer to variable
., af17  e9 00    sbc #$00
```

```
., af19  a5 65     lda $65
., af1b  e9 a0     sbc #$a0
., af1d  90 08     bcc $af27
., af1f  a9 a2     lda #$a2
., af21  e5 64     sbc $64
., af23  a9 e3     lda #$e3
., af25  e5 65     sbc $65
., af27  60        rts
```

44840 ISVAR: SEARCH FOR VARIABLE

The variable is identified from text. If it is reserved and a string (ie TI$), this is processed. If it is numeric, it drops to the following routine for further processing.

```
., af28  20 8b b0 jsr $b08b    ;identify variable in text
., af2b  85 64     sta $64      ;holds pointer to variable
., af2d  84 65     sty $65
., af2f  a6 45     ldx $45      ;VARNAM - variable name
., af31  a4 46     ldy $46
., af33  a5 0d     lda $0d      ;VALTYP - data type
., af35  f0 26     beq $af5d    ;number - put into fac#1
., af37  a9 00     lda #$00
., af39  85 70     sta $70      ;FACOV - fac#1 rounding
., af3b  20 14 af jsr $af14    ;identify reserved variable
., af3e  90 1c     bcc $af5c    ;not reserved - RTS
., af40  e0 54     cpx #$54     ;ASCII T?
., af42  d0 18     bne $af5c
., af44  c0 c9     cpy #$c9     ;ASCII shift I - ie TI$
., af46  d0 14     bne $af5c
```

44872 TISASC: CONVERT TI TO ASCII STRING

Firstly, the real-time clock is read into $63 - $65. This is then converted into an ASCII string, and the string set up in memory.

```
., af48  20 84 af jsr $af84    ;read real time clock
., af4b  84 5e     sty $5e
., af4d  88        dey
., af4e  84 71     sty $71
., af50  a0 06     ldy #$06
., af52  84 5d     sty $5d
., af54  a0 24     ldy #$24
., af56  20 68 be jsr $be68    ;convert TI into ASCII string
., af59  4c 6f b4 jmp $b46f    ;set up string
., af5c  60        rts
```

The remainder of the routine handles numeric variables. If

it is flpt, then it is tested to see if it is reserved. If TI, then the real-time clock is read. If ST, then the I/O status word is read. The variable is placed into fac#1.

```
., af5d  24 0e     bit $0e       ;INTFLG - data type
., af5f  10 0d     bpl $af6e     ;flpt
., af61  a0 00     ldy #$00
., af63  b1 64     lda ($64),y   ;set up integer in (A/Y)
., af65  aa        tax
., af66  c8        iny
., af67  b1 64     lda ($64),y
., af69  a8        tay
., af6a  8a        txa
., af6b  4c 91 b3  jmp $b391     ;convert to flpt in fac#1
., af6e  20 14 af  jsr $af14     ;identify reserved variable
., af71  90 2d     bcc $afa0     ;not reserved
., af73  e0 54     cpx #$54      ;ASCII T
., af75  d0 1b     bne $af92
., af77  c0 49     cpy #$49      ;ASCII I
., af79  d0 25     bne $afa0
., af7b  20 84 af  jsr $af84     ;read TI
., af7e  98        tya
., af7f  a2 a0     ldx #$a0
., af81  4c 4f bc  jmp $bc4f     ;do SGN
., af84  20 de ff  jsr $ffde     ;RDTIM - read real-time clock
., af87  86 64     stx $64       ;store time in fac#1
., af89  84 63     sty $63
., af8b  85 65     sta $65
., af8d  a0 00     ldy #$00
., af8f  84 62     sty $62       ;sign byte of fac#1
., af91  60        rts
., af92  e0 53     cpx #$53      ;ASCII S?
., af94  d0 0a     bne $afa0
., af96  c0 54     cpy #$54      ;ASCII T?
., af98  d0 06     bne $afa0
., af9a  20 b7 ff  jsr $ffb7     ;READST - read I/O status
                                 word
., af9d  4c 3c bc  jmp $bc3c     ;do SGN
., afa0  a5 64     lda $64
., afa2  a4 65     ldy $65
., afa4  4c a2 bb  jmp $bba2     ;load fac#1 with flpt at
                                 (A/Y)
```

44967 ISFUN: IDENTIFY FUNCTION TYPE

(A) is doubled, pushed onto the stack and placed in (X). After performing CHRGET, the result of comparing (X) with #8F determines whether a string or numeric function is evaluated.

```
.,  afa7  0a        asl
.,  afa8  48        pha
.,  afa9  aa        tax
.,  afaa  20 73 00  jsr $0073     ;CHRGET
.,  afad  e0 8f     cpx #$8f      ;string or numeric?
.,  afaf  90 20     bcc $afd1     ;do numeric
```

44977 STRFUN: EVALUATE STRING FUNCTION

Left bracket is confirmed and the expression within evaluated. The next character is confirmed to be a comma and the variable confirmed as a string. A 1 byte parameter is input and the function performed.

```
.,  afb1  20 fa ae  jsr $aefa     ;confirm left bracket
.,  afb4  20 9e ad  jsr $ad9e     ;evaluate expression in text
.,  afb7  20 fd ae  jsr $aefd     ;confirm comma
.,  afba  20 8f ad  jsr $ad8f     ;confirm string result
.,  afbd  68        pla
.,  afbe  aa        tax
.,  afbf  a5 65     lda $65       ;push ($64)
.,  afc1  48        pha
.,  afc2  a5 64     lda $64
.,  afc4  48        pha
.,  afc5  8a        txa
.,  afc6  48        pha
.,  afc7  20 9e b7  jsr $b79e     ;evaluate text to 1 byte in
                                  (X)
.,  afca  68        pla
.,  afcb  a8        tay
.,  afcc  8a        txa
.,  afcd  48        pha
.,  afce  4c d6 af  jmp $afd6     ;perform function
```

45009 NUMFUN: EVALUATE NUMERIC FUNCTION

The expression within brackets is evaluated. The address of the function is set up in ($55) and the function performed. Finally the result is confirmed as numeric.

```
.,  afd1  20 f1 ae  jsr $aef1     ;do expression in brackets
.,  afd4  68        pla
.,  afd5  a8        tay
.,  afd6  b9 ea 9f  lda $9fea,y   ;get function address
.,  afd9  85 55     sta $55
.,  afdb  b9 eb 9f  lda $9feb,y
.,  afde  85 56     sta $56
.,  afe0  20 54 00  jsr $0054     ;perform function
.,  afe3  4c 8d ad  jmp $ad8d     ;confirm numeric result
```

45030 OROP: PERFORM OR, AND

This is essentially the same routine for both keywords, the flag $0B being used to distinguish them. The two arguments are in fac#1 and fac#2. First, fac#1 is EORed with $0B and the result put in ($07). Next, fac#2 is copied into fac#1 and EORed with $0B. It is then ANDed with the first result and finally EORed again with $0B. The entry point for AND is $AFE9, and is given the label ANDOP.

```
., afe6  a0 ff    ldy #$ff
., afe8  2c a0 00 bit $00a0    ;mask - AND entry point
., afeb  84 0b    sty $0b      ;COUNT - #FF = OR, #00 = AND
., afed  20 bf b1 jsr $b1bf    ;convert fac#1 to integer in
                               ($64)
., aff0  a5 64    lda $64
., aff2  45 0b    eor $0b      ;COUNT
., aff4  85 07    sta $07      ;($07) holds intermediate
                               result
., aff6  a5 65    lda $65
., aff8  45 0b    eor $0b      ;COUNT
., affa  85 08    sta $08
., affc  20 fc bb jsr $bbfc    ;copy fac#2 to fac#1
., afff  20 bf b1 jsr $b1bf    ;convert fac#1 to integer in
                               ($64)
., b002  a5 65    lda $65
., b004  45 0b    eor $0b      ;COUNT
., b006  25 08    and $08
., b008  45 0b    eor $0b      ;COUNT
., b00a  a8       tay
., b00b  a5 64    lda $64      ;repeat for next byte
., b00d  45 0b    eor $0b
., b00f  25 07    and $07
., b011  45 0b    eor $0b      ;result in (A/Y)
., b013  4c 91 b3 jmp $b391    ;convert (A/Y) to flpt in
                               fac#1
```

45078 DOREL: RERFORM <, =, >

The routine confirms both items to be of the same data type, then does string or numeric comparison accordingly.

```
., b016  20 90 ad jsr $ad90    ;confirm result
., b019  b0 13    bcs $b02e    ;do string comparison
```

45083 NUMREL: NUMERIC COMPARISON

FAC#2 is modified to include the sign bit in its mantissa.

(X/Y) are set to point to fac#2 which is compared with fac#1. Finally (A) is set according to the result.

```
., b01b  a5 6e     lda $6e      ;ARGSGN - fac#2 sign
., b01d  09 7f     ora #$7f
., b01f  25 6a     and $6a      ;ARGHO1 - fac#2 mantissa
., b021  85 6a     sta $6a      ;put sign into fac#2 mantissa
., b023  a9 69     lda #$69
., b025  a0 00     ldy #$00     ;(A/Y) points to fac#2
., b027  20 5b bc  jsr $bc5b    ;compare fac#1 with fac#2
., b02a  aa        tax
., b02b  4c 61 b0  jmp $b061    ;set (A) according to result
```

<u>45102 STRREL: STRING COMPARISON</u>

The two strings for comparison are set up. String 1 has $61 = length and ($62) = pointer. String 2 has (A) = length and ($6C) = pointer. The two strings are compared throughout their length until an inequality is found. (X) is set according to the result of the comparison - string 1 > string 2 and (X)=1, string 1 < string 2 and (X)=#FF, equal and (X)=0, Finally (A) is set according to this result.

```
., b02e  a9 00     lda #$00
., b030  85 0d     sta $0d      ;VALTYP - set numeric data
., b032  c6 4d     dec $4d
., b034  20 a6 b6  jsr $b6a6    ;do string 1 housekeeping set
                                 pointers
., b037  85 61     sta $61      ;string 1: length
., b039  86 62     stx $62      ;string 1: pointer
., b03b  84 63     sty $63
., b03d  a5 6c     lda $6c      ;ARGHO3
., b03f  a4 6d     ldy $6d
., b041  20 aa b6  jsr $b6aa    ;do string 2 housekeeping set
                                 pointers
., b044  86 6c     stx $6c      ;string 2: pointer. length is
                                 in (A)
., b046  84 6d     sty $6d
., b048  aa        tax
., b049  38        sec
., b04a  e5 61     sbc $61      ;subtract string lengths
., b04c  f0 08     beq $b056
., b04e  a9 01     lda #$01
., b050  90 04     bcc $b056
., b052  a6 61     ldx $61
., b054  a9 ff     lda #$ff
., b056  85 66     sta $66      ;counter
., b058  a0 ff     ldy #$ff
., b05a  e8        inx
```

```
., b05b  c8        iny
., b05c  ca        dex
., b05d  d0 07     bne $b066    ;not end of string - compare
., b05f  a6 66     ldx $66      ;counter
., b061  30 0f     bmi $b072
., b063  18        clc
., b064  90 0c     bcc $b072
., b066  b1 6c     lda ($6c),y  ;get string 2
., b068  d1 62     cmp ($62),y  ;compare with string 1
., b06a  f0 ef     beq $b05b
., b06c  a2 ff     ldx #$ff
., b06e  b0 02     bcs $b072
., b070  a2 01     ldx #$01
., b072  e8        inx
., b073  8a        txa
., b074  2a        rol
., b075  25 12     and $12      ;TANSGN - comparison result
., b077  f0 02     beq $b07b
., b079  a9 ff     lda #$ff
., b07b  4c 3c bc  jmp $bc3c    ;do SGN
```

45182 DIM: PERFORM DIM

The variable is identified and set up. If the next byte of text is a comma, then DIM is reentered, otherwise it must be a terminator.

```
., b07e  20 fd ae  jsr $aefd    ;confirm comma
., b081  aa        tax          ;normal entry point here
., b082  20 90 b0  jsr $b090    ;identify variable
., b085  20 79 00  jsr $0079    ;CHRGOT
., b088  d0 f4     bne $b07e    ;reenter DIM
., b08a  60        rts
```

45195 PTRGET: IDENTIFY VARIABLE

The first character of the variable name is checked to be alphabetic. The second character of the name is checked to be either alphabetic or numeric. All further characters are bypassed until the first invalid character. If this is a dollar sign, then string mode is set; if it is a percentage sign, then integer mode is set. The name in VARNAM is then set according to type (see table below). If a left bracket is found, then array parameters are set up.

byte	INTEGER VARIABLE	FLOATING POINT VARIABLE
0	name character 1 +128	name character 1
1	name character 2 +128	name character 2
2	integer high byte	exponent +128
3	integer low byte	sign bit & mantissa 1
4	0	mantissa 2
5	0	mantissa 3
6	0	mantissa 4

byte	STRING VARIABLE	FUNCTION DEFINITION
0	name character 1	name character 1 +128
1	name character 2 +128	name character 2
2	length of string	DEFFN= address low byte
3	start address low byte	DEFFN= address high byte
4	start address high byte	pointer to var. exponent lo
5	0	pointer to var. exponent hi
6	0	0

```
., b08b  a2 00     ldx #$00
., b08d  20 79 00  jsr $0079    ;CHRGOT
., b090  86 0c     stx $0c      ;DIMFLG - default array
                                dimension
., b092  85 45     sta $45      ;<VARNAM - variable name
., b094  20 79 00  jsr $0079    ;CHRGOT
., b097  20 13 b1  jsr $b113    ;is (A) alphabetic?
., b09a  b0 03     bcs $b09f    ;yes
., b09c  4c 08 af  jmp $af08    ;?SYNTAX error
., b09f  a2 00     ldx #$00
., b0a1  86 0d     stx $0d      ;VALTYP - set numeric data
., b0a3  86 0e     stx $0e      ;INTFLG - set flpt
., b0a5  20 73 00  jsr $0073    ;CHRGET
., b0a8  90 05     bcc $b0af    ;(A) is numeric
., b0aa  20 13 b1  jsr $b113    ;check for alphabetic
                                character
., b0ad  90 0b     bcc $b0ba    ;no - invalid character - end
                                of name
., b0af  aa        tax
., b0b0  20 73 00  jsr $0073    ;CHRGET
., b0b3  90 fb     bcc $b0b0    ;numeric or
., b0b5  20 13 b1  jsr $b113    ;alphabetic -
., b0b8  b0 f6     bcs $b0b0    ;loop till end of variable
```

```
                                    name
., b0ba  c9 24     cmp #$24         ;ASCII $?
., b0bc  d0 06     bne $b0c4
., b0be  a9 ff     lda #$ff
., b0c0  85 0d     sta $0d          ;set VALTYP = string
., b0c2  d0 10     bne $b0d4        ;adjust VARNAM
., b0c4  c9 25     cmp #$25         ;ASCII %?
., b0c6  d0 13     bne $b0db
., b0c8  a5 10     lda $10          ;SUBFLG - subscript ref
., b0ca  d0 d0     bne $b09c        ;?SYNTAX error
., b0cc  a9 80     lda #$80
., b0ce  85 0e     sta $0e          ;set INTFLG to integer
., b0d0  05 45     ora $45          ;set bit 7 of <VARNAM
., b0d2  85 45     sta $45
., b0d4  8a        txa
., b0d5  09 80     ora #$80         ;set bit 7 of >VARNAM
., b0d7  aa        tax
., b0d8  20 73 00  jsr $0073        ;CHRGET
., b0db  86 46     stx $46          ;store >VARNAM
., b0dd  38        sec
., b0de  05 10     ora $10          ;SUBFLG
., b0e0  e9 28     sbc #$28         ;check for left bracket
., b0e2  d0 03     bne $b0e7        ;locate ordinary variable
., b0e4  4c d1 b1  jmp $b1d1        ;get array parameters
```

45286 ORDVAR: LOCATE ORDINARY VARIABLE

This is a loop to search for a variable between VARTAB and ARYTAB. ($5F) holds a temporary pointer. Each variable found is compared with that in VARNAM. If they match, then the variable is set up. If not, then the next variable is checked. All variables are 7 bytes long so the next variable is located simply by adding 7 to the pointer to the last one. If the variable is not found then it is created.

```
., b0e7  a0 00     ldy #$00
., b0e9  84 10     sty $10          ;SUBFLG - subscript ref
., b0eb  a5 2d     lda $2d          ;VARTAB - start of variables
., b0ed  a6 2e     ldx $2e
., b0ef  86 60     stx $60
., b0f1  85 5f     sta $5f
., b0f3  e4 30     cpx $30          ;>ARYTAB - start of arrays
., b0f5  d0 04     bne $b0fb
., b0f7  c5 2f     cmp $2f          ;<ARYTAB
., b0f9  f0 22     beq $b11d        ;create new variable
., b0fb  a5 45     lda $45          ;<VARNAM - variable name
., b0fd  d1 5f     cmp ($5f),y      ;compare it with name in
                                    table
., b0ff  d0 08     bne $b109        ;not same - next
```

```
., b101  a5 46     lda $46       ;>VARNAM
., b103  c8        iny
., b104  d1 5f     cmp ($5f),y   ;compare it with name in
                                 table
., b106  f0 7d     beq $b185     ;set up variable
., b108  88        dey
., b109  18        clc
., b10a  a5 5f     lda $5f
., b10c  69 07     adc #$07      ;set pointer to next variable
., b10e  90 e1     bcc $b0f1
., b110  e8        inx
., b111  d0 dc     bne $b0ef
```

45331 ISLETC: DOES (A) HOLD AN ALPHABETIC CHARACTER?

If (A) >= #40 or (A) <= #A5 then it is alphabetic and C=1.

```
., b113  c9 41     cmp #$41      ;ASCII A?
., b115  90 05     bcc $b11c     ;less - end
., b117  e9 5b     sbc #$5b      ;ASCII [ - next character to
                                 Z
., b119  38        sec
., b11a  e9 a5     sbc #$a5
., b11c  60        rts
```

45341 NOTFNS: CREATE NEW VARIABLE

If the top byte on the stack is #2A then the variable is not created since the routine was called from EVALUATE EXPRESSION. A null result is returned by using the constant at $B1F3. If #2A is not found, then the following routine is called to create the ordinary variable.

```
., b11d  68        pla
., b11e  48        pha
., b11f  c9 2a     cmp #$2a      ;called from evaluate
                                 expression?
., b121  d0 05     bne $b128     ;no - create variable
., b123  a9 13     lda #$13
., b125  a0 bf     ldy #$bf      ;$BF13 = constant 0 in flpt
., b127  60        rts
```

45352 NOTEVL: CREATE VARIABLE

Reserved variables are tested for. TI$ produces a null string, but both TI and ST produce ?SYNTAX error. If there are arrays in existance, they are moved up by 7 bytes to allow space for the new variable. If there are a large number of arrays, this can take a second or two. ARYTAB is

updated and VARNAM written into RAM followed by 5 zeros. Thus the variable is created with an initial null value. On exit, ($5F) points to the name and VARPNT to its value (or the string pointers).

```
., b128  a5 45     lda $45       ;VARNAM - variable name
., b12a  a4 46     ldy $46
., b12c  c9 54     cmp #$54      ;ASCII T?
., b12e  d0 0b     bne $b13b
., b130  c0 c9     cpy #$c9      ;ASCII shift I?
., b132  f0 ef     beq $b123     ;TI$ - return null string
., b134  c0 49     cpy #$49      ;ASCII I?
., b136  d0 03     bne $b13b
., b138  4c 08 af  jmp $af08     ;?SYNTAX error
., b13b  c9 53     cmp #$53      ;ASCII S?
., b13d  d0 04     bne $b143
., b13f  c0 54     cpy #$54      ;ASCII T?
., b141  f0 f5     beq $b138     ;ST gives ?SYNTAX error
., b143  a5 2f     lda $2f       ;ARYTAB - start of arrays
., b145  a4 30     ldy $30
., b147  85 5f     sta $5f       ;pointer for memory move
., b149  84 60     sty $60
., b14b  a5 31     lda $31       ;STREND - end of arrays +1
., b14d  a4 32     ldy $32
., b14f  85 5a     sta $5a       ;pointer for memory move
., b151  84 5b     sty $5b
., b153  18        clc
., b154  69 07     adc #$07      ;move up by 7 bytes
., b156  90 01     bcc $b159
., b158  c8        iny
., b159  85 58     sta $58       ;pointer for memory move
., b15b  84 59     sty $59
., b15d  20 b8 a3  jsr $a3b8     ;move memory
., b160  a5 58     lda $58
., b162  a4 59     ldy $59
., b164  c8        iny
., b165  85 2f     sta $2f       ;update ARYTAB
., b167  84 30     sty $30
., b169  a0 00     ldy #$00      ;index through variable
., b16b  a5 45     lda $45       ;<VARNAM
., b16d  91 5f     sta ($5f),y   ;write variable name
., b16f  c8        iny
., b170  a5 46     lda $46       ;>VARNAM
., b172  91 5f     sta ($5f),y
., b174  a9 00     lda #$00
., b176  c8        iny
., b177  91 5f     sta ($5f),y   ;write null value to variable
., b179  c8        iny
., b17a  91 5f     sta ($5f),y
```

```
., b17c  c8       iny
., b17d  91 5f    sta ($5f),y
., b17f  c8       iny
., b180  91 5f    sta ($5f),y
., b182  c8       iny
., b183  91 5f    sta ($5f),y
., b185  a5 5f    lda $5f
., b187  18       clc
., b188  69 02    adc #$02
., b18a  a4 60    ldy $60
., b18c  90 01    bcc $b18f
., b18e  c8       iny
., b18f  85 47    sta $47         ;VARPNT - pointer to variable
., b191  84 48    sty $48
., b193  60       rts
```

45460 ARYGET: ALLOCATE ARRAY POINTER SPACE

The number of subscripts for the array is held in COUNT. This is doubled, #5 added then the result added to ($5F). This provides space for the array header and not for the data itself. The header format is shown below:

```
byte:
 0  1st character in name (type coded)
 1  2nd character in name (type coded)
 2  pointer to next array, low byte
 3  pointer to next array, high byte
 4  number of dimensions in array
 5  number of elements in last subscript
low byte
 6  number of elements in last subscript
high byte
 7  number in penultimate subscript low
byte
```

```
., b194  a5 0b    lda $0b
., b196  0a       asl
., b197  69 05    adc #$05
., b199  65 5f    adc $5f
., b19b  a4 60    ldy $60
., b19d  90 01    bcc $b1a0
., b19f  c8       iny
., b1a0  85 58    sta $58
., b1a2  84 59    sty $59
., b1a4  60       rts             ;continuing to first
                                   subscript
```

45477 N32768: CONSTANT 32768 IN FLPT

```
.:b1a5 90 80 00 00 00
```

45482 FACINX: FAC#1 TO INTEGER IN (A/Y)

```
., b1aa  20 bf b1 jsr $b1bf    ;convert fac#1 to integer in
                               ($64)
., b1ad  a5 64    lda $64      ;put integer into (A/Y)
., b1af  a4 65    ldy $65
., b1b1  60       rts
```

45490 INTIDX: EVALUATE TEXT FOR INTEGER

The next byte of text is fetched and evaluated. It is confirmed as numeric, and if the result is positive and less than 32768 then the following routine is performed.

```
., b1b2  20 73 00 jsr $0073    ;CHRGET
., b1b5  20 9e ad jsr $ad9e    ;evaluate expression in text
., b1b8  20 8d ad jsr $ad8d    ;confirm numeric result
., b1bb  a5 66    lda $66      ;FACSGN - fac#1 sign
., b1bd  30 0d    bmi $b1cc    ;?ILLEGAL QTY error
```

45503 AYINT: FAC#1 TO SIGNED INTEGER

FACEXP is compared with #90. If not less, then it is compared with the flpt number 32768. If not equal, then ?ILLEGAL QUANTITY. Finally, fac#1 is converted to an integer in the range -32768 to +32767.

```
., b1bf  a5 61    lda $61      ;FACEXP - fac#1 exponent
., b1c1  c9 90    cmp #$90
., b1c3  90 09    bcc $b1ce
., b1c5  a9 a5    lda #$a5
., b1c7  a0 b1    ldy #$b1     ;$B1A5 = constant 32768 in
                               flpt
., b1c9  20 5b bc jsr $bc5b    ;compare fac#1 with flpt at
                               (A/Y)
., b1cc  d0 7a    bne $b248    ;?ILLEGAL QUANTITY error
., b1ce  4c 9b bc jmp $bc9b    ;convert fac#1 to integer
```

45521 ISARY: GET ARRAY PARAMETERS

The array parameters are read from text and the details of the array are set up on the stack. Text is evaluated to an integer, giving the number of elements in the first subscript (including 0th). This is placed on the stack. If

there is a comma immediately following in text then the procedure is repeated for the next subscript etc. Finally, the number of subscripts is put in COUNT, a right bracket checked and the next routine performed.

```
., b1d1  a5 0c     lda $0c       ;DIMFLG - default array DIM
., b1d3  05 0e     ora $0e       ;INTFLG - data type
., b1d5  48        pha
., b1d6  a5 0d     lda $0d       ;push VALTYP
., b1d8  48        pha
., b1d9  a0 00     ldy #$00      ;(Y) holds number of
                                 subscripts
., b1db  98        tya
., b1dc  48        pha
., b1dd  a5 46     lda $46       ;push VARNAM - variable name
., b1df  48        pha
., b1e0  a5 45     lda $45
., b1e2  48        pha
., b1e3  20 b2 b1  jsr $b1b2     ;evaluate text - get #
                                 elements
., b1e6  68        pla
., b1e7  85 45     sta $45       ;pull VARNAM
., b1e9  68        pla
., b1ea  85 46     sta $46
., b1ec  68        pla
., b1ed  a8        tay
., b1ee  ba        tsx
., b1ef  bd 02 01  lda $0102,x   ;insert ($64) under top 2
                                 stack bytes
., b1f2  48        pha
., b1f3  bd 01 01  lda $0101,x
., b1f6  48        pha
., b1f7  a5 64     lda $64
., b1f9  9d 02 01  sta $0102,x
., b1fc  a5 65     lda $65
., b1fe  9d 01 01  sta $0101,x
., b201  c8        iny
., b202  20 79 00  jsr $0079     ;CHRGOT
., b205  c9 2c     cmp #$2c      ;ASCII comma?
., b207  f0 d2     beq $b1db     ;get elements for next
                                 subscript
., b209  84 0b     sty $0b
., b20b  20 f7 ae  jsr $aef7     ;confirm right bracket
., b20e  68        pla
., b20f  85 0d     sta $0d       ;pull VALTYP
., b211  68        pla
., b212  85 0e     sta $0e       ;pull INTFLG
., b214  29 7f     and #$7f
., b216  85 0c     sta $0c       ;DIMFLG
```

45592 FNDARY: FIND ARRAY

This is a loop to check for the presence of the subscripted variable between ARYTAB and STREND. There are only two exits to the loop: (1) if the array is not found, then it must be created, and (2) if the array is found, when the element within it is located.

```
., b218  a6 2f     ldx $2f       ;ARYTAB - start of arrays
., b21a  a5 30     lda $30
., b21c  86 5f     stx $5f       ;pointer - current position
                                 in table
., b21e  85 60     sta $60
., b220  c5 32     cmp $32       ;>STREND - end of arrays +1
., b222  d0 04     bne $b228
., b224  e4 31     cpx $31       ;<STREND
., b226  f0 39     beq $b261     ;array not found - create it
., b228  a0 00     ldy #$00
., b22a  b1 5f     lda ($5f),y   ;get array name
., b22c  c8        iny
., b22d  c5 45     cmp $45       ;VARNAM - variable name
., b22f  d0 06     bne $b237
., b231  a5 46     lda $46
., b233  d1 5f     cmp ($5f),y   ;compare names
., b235  f0 16     beq $b24d     ;array found - find element
., b237  c8        iny
., b238  b1 5f     lda ($5f),y   ;increment pointer to next
                                 position
., b23a  18        clc
., b23b  65 5f     adc $5f
., b23d  aa        tax
., b23e  c8        iny
., b23f  b1 5f     lda ($5f),y
., b241  65 60     adc $60
., b243  90 d7     bcc $b21c     ;loop back to continue search
```

45637 BSERR: ?BAD SUBSCRIPT/?ILLEGAL QUANTITY

The two entry points are $B245 for ?BAD SUBSCRIPT and $B248 for ?ILLEGAL QUANTITY. The remainder of the routine is concerned with an array being found by the previous routine. If DIMFLG is set, then ?REDIM'D ARRAY error, otherwise pointer space is allocated and the number of subscripts is checked. Finally the element is located within the array.

```
., b245  a2 12     ldx #$12      ;flag ?BAD SUBSCRIPT
., b247  2c a2 0e bit $0ea2      ;mask - flag ?ILLEGAL
                                 QUANTITY
```

```
., b24a  4c 37 a4 jmp $a437     ;do error
., b24d  a2 13    ldx #$13      ;flag ?REDIM'D ARRAY
., b24f  a5 0c    lda $0c       ;DIMFLG - array DIM flag
., b251  d0 f7    bne $b24a     ;set - do error
., b253  20 94 b1 jsr $b194     ;allocate array pointer space
., b256  a5 0b    lda $0b       ;COUNT - number of subscripts
., b258  a0 04    ldy #$04
., b25a  d1 5f    cmp ($5f),y   ;check subscripts in array
., b25c  d0 e7    bne $b245     ;wrong - ?BAD SUBSCRIPT error
., b25e  4c ea b2 jmp $b2ea     ;find element in array
```

45665 NOTFDD: CREATE ARRAY

This routine is called to create the array if it was not found. Array header space is allocated and memory checked. The variable name is written into high RAM. A loop works through each of the subscripts, calculating its size and storing the element from the stack. Once this is done, memory is again checked and the pointer to the end of arrays is updated.

```
., b261  20 94 b1 jsr $b194     ;allocate array pointer space
., b264  20 08 a4 jsr $a408     ;is (A/Y) lower than FRETOP?
., b267  a0 00    ldy #$00
., b269  84 72    sty $72
., b26b  a2 05    ldx #$05
., b26d  a5 45    lda $45       ;VARNAM - variable name
., b26f  91 5f    sta ($5f),y   ;write <VARNAM in memory
., b271  10 01    bpl $b274
., b273  ca       dex
., b274  c8       iny
., b275  a5 46    lda $46
., b277  91 5f    sta ($5f),y   ;write >VARNAM in memory
., b279  10 02    bpl $b27d
., b27b  ca       dex
., b27c  ca       dex
., b27d  86 71    stx $71
., b27f  a5 0b    lda $0b       ;COUNT - number of subscripts
., b281  c8       iny
., b282  c8       iny
., b283  c8       iny
., b284  91 5f    sta ($5f),y   ;store count in array
., b286  a2 0b    ldx #$0b
., b288  a9 00    lda #$00
., b28a  24 0c    bit $0c       ;DIMFLG - array dimension
                                flag
., b28c  50 08    bvc $b296
., b28e  68       pla
., b28f  18       clc
```

```
., b290  69 01     adc #$01
., b292  aa        tax
., b293  68        pla
., b294  69 00     adc #$00
., b296  c8        iny
., b297  91 5f     sta ($5f),y  ;store DIMFLG
., b299  c8        iny
., b29a  8a        txa
., b29b  91 5f     sta ($5f),y
., b29d  20 4c b3  jsr $b34c    ;calculate # bytes in
                                subscript
., b2a0  86 71     stx $71
., b2a2  85 72     sta $72
., b2a4  a4 22     ldy $22      ;<INDEX1
., b2a6  c6 0b     dec $0b      ;COUNT - next subscript
., b2a8  d0 dc     bne $b286
., b2aa  65 59     adc $59
., b2ac  b0 5d     bcs $b30b    ;?OUT OF MEMORY error
., b2ae  85 59     sta $59
., b2b0  a8        tay
., b2b1  8a        txa
., b2b2  65 58     adc $58
., b2b4  90 03     bcc $b2b9
., b2b6  c8        iny
., b2b7  f0 52     beq $b30b    ;?OUT OF MEMORY error
., b2b9  20 08 a4  jsr $a408    ;is (A/Y) lower than FRETOP?
., b2bc  85 31     sta $31      ;STREND - end of arrays +1
., b2be  84 32     sty $32
., b2c0  a9 00     lda #$00
., b2c2  e6 72     inc $72
., b2c4  a4 71     ldy $71
., b2c6  f0 05     beq $b2cd
., b2c8  88        dey
., b2c9  91 58     sta ($58),y
., b2cb  d0 fb     bne $b2c8
., b2cd  c6 59     dec $59
., b2cf  c6 72     dec $72
., b2d1  d0 f5     bne $b2c8
., b2d3  e6 59     inc $59
., b2d5  38        sec
., b2d6  a5 31     lda $31      ;<STREND
., b2d8  e5 5f     sbc $5f
., b2da  a0 02     ldy #$02
., b2dc  91 5f     sta ($5f),y
., b2de  a5 32     lda $32      ;>STREND
., b2e0  c8        iny
., b2e1  e5 60     sbc $60
., b2e3  91 5f     sta ($5f),y
., b2e5  a5 0c     lda $0c      ;DIMFLG
```

```
., b2e7  d0 62     bne $b34b     ; - RTS
., b2e9  c8        iny
., b2ea  b1 5f     lda ($5f),y
., b2ec  85 0b     sta $0b       ;COUNT - number of subscripts
., b2ee  a9 00     lda #$00
., b2f0  85 71     sta $71
., b2f2  85 72     sta $72
., b2f4  c8        iny
., b2f5  68        pla
., b2f6  aa        tax
., b2f7  85 64     sta $64
., b2f9  68        pla
., b2fa  85 65     sta $65
., b2fc  d1 5f     cmp ($5f),y
., b2fe  90 0e     bcc $b30e     ;locate element in array
., b300  d0 06     bne $b308     ;?BAD SUBSCRIPT error
., b302  c8        iny
., b303  8a        txa
., b304  d1 5f     cmp ($5f),y
., b306  90 07     bcc $b30f     ;locate element in array
., b308  4c 45 b2  jmp $b245     ;?BAD SUBSCRIPT error
., b30b  4c 35 a4  jmp $a435     ;?OUT OF MEMORY error
```

45838 INLPN2: LOCATE ELEMENT IN ARRAY

This routine works through the array until the required element is found. VARPNT is set to point to its location.

```
., b30e  c8        iny           ;(Y) points to subscript
., b30f  a5 72     lda $72       ;pointer to array
., b311  05 71     ora $71
., b313  18        clc
., b314  f0 0a     beq $b320
., b316  20 4c b3  jsr $b34c     ;calculate bytes in subscript
., b319  8a        txa
., b31a  65 64     adc $64       ;add result to ($64)
., b31c  aa        tax
., b31d  98        tya
., b31e  a4 22     ldy $22       ;INDEX1
., b320  65 65     adc $65
., b322  86 71     stx $71       ;<pointer
., b324  c6 0b     dec $0b       ;COUNT - number of subscripts
., b326  d0 ca     bne $b2f2
., b328  85 72     sta $72       ;>pointer
., b32a  a2 05     ldx #$05
., b32c  a5 45     lda $45       ;VARNAM - variable name
., b32e  10 01     bpl $b331
., b330  ca        dex
., b331  a5 46     lda $46
```

```
., b333  10 02    bpl $b337
., b335  ca       dex
., b336  ca       dex
., b337  86 28    stx $28
., b339  a9 00    lda #$00
., b33b  20 55 b3 jsr $b355
., b33e  8a       txa
., b33f  65 58    adc $58
., b341  85 47    sta $47       ;VARPNT - points to element
., b343  98       tya
., b344  65 59    adc $59
., b346  85 48    sta $48
., b348  a8       tay
., b349  a5 47    lda $47
., b34b  60       rts
```

45900 UMULT: NUMBER OF BYTES IN SUBSCRIPT

This routine calculates the size of an array subscript. The subscript (Y) is stored in INDEX1 and the number of elements in the subscript in ($28). The number of bytes is calculated by a loop which loops 16 times. The result is in (X/Y).

```
., b34c  84 22    sty $22       ;INDEX1
., b34e  b1 5f    lda ($5f),y   ;get number of elements in
                                subscript
., b350  85 28    sta $28
., b352  88       dey
., b353  b1 5f    lda ($5f),y
., b355  85 29    sta $29
., b357  a9 10    lda #$10
., b359  85 5d    sta $5d       ;loop counter
., b35b  a2 00    ldx #$00
., b35d  a0 00    ldy #$00
., b35f  8a       txa
., b360  0a       asl
., b361  aa       tax
., b362  98       tya
., b363  2a       rol
., b364  a8       tay
., b365  b0 a4    bcs $b30b     ;?OUT OF MEMORY error
., b367  06 71    asl $71
., b369  26 72    rol $72
., b36b  90 0b    bcc $b378
., b36d  18       clc
., b36e  8a       txa
., b36f  65 28    adc $28       ;<no of elements
., b371  aa       tax
```

```
., b372  98        tya
., b373  65 29     adc $29        ;>no of elements
., b375  a8        tay
., b376  b0 93     bcs $b30b      ;?OUT OF MEMORY error
., b378  c6 5d     dec $5d        ;loop counter
., b37a  d0 e3     bne $b35f      ;continue loop
., b37c  60        rts
```

45949 FRE: PERFORM FRE

This routine causes unwanted strings to be erased, and the amount of available RAM to be calculated. If string mode is set then string housekeeping is performed. Garbage collect is performed and STREND is subtracted from FRETOP to give the number of bytes free. The result is transferred from (A/Y) to fac#1 by the next routine. The dummy argument in FRE(n) is ignored. This value was placed in fac#1 before the routine was called. Note that any value above 32767 will be treated as negative by the next routine, since the most significant bit of the integer value is used to indicate the sign of the number.

```
., b37d  a5 0d     lda $0d        ;VALTYP - data type
., b37f  f0 03     beq $b384      ;numeric
., b381  20 a6 b6  jsr $b6a6      ;do string housekeeping
., b384  20 26 b5  jsr $b526      ;do garbage collect
., b387  38        sec
., b388  a5 33     lda $33        ;FRETOP - bottom of strings
., b38a  e5 31     sbc $31        ;STREND - end of arrays +1
., b38c  a8        tay
., b38d  a5 34     lda $34
., b38f  e5 32     sbc $32
```

45969 GIVAYF: CONVERT INTEGER IN (A/Y) TO FLPT

(A/Y) holds a signed integer in the range -32768 to +32767. This is placed in fac#1 and converted to flpt.

```
., b391  a2 00     ldx #$00
., b393  86 0d     stx $0d        ;VALTYP - set numeric mode
., b395  85 62     sta $62        ;FACHO - fac#1
., b397  84 63     sty $63
., b399  a2 90     ldx #$90
., b39b  4c 44 bc  jmp $bc44      ;convert fac#1 to flpt
```

45982 POS: PERFORM POS

The dummy argument in POS(n) has been placed in fac#1 and is ignored. The KERNAL routine, PLOT is called to read the

cursor X-Y position. The Y position is discarded and the previous routine performed.

```
., b39e  38       sec
., b39f  20 f0 ff jsr $fff0    ;PLOT - read cursor position
., b3a2  a9 00    lda #$00     ;ignore cursor row
., b3a4  f0 eb    beq $b391    ;convert to flpt
```

45990 ERRDIR: CONFIRM PROGRAM MODE

This routine is called from keywords which do not operate in direct mode. The high byte of CURLIN is checked. If it holds #FF, then the command was entered in direct mode and ?ILLEGAL DIRECT error is called.

```
., b3a6  a6 3a    ldx $3a      ;>CURLIN - current line
                               number
., b3a8  e8       inx
., b3a9  d0 a0    bne $b34b    ;program mode - RTS
., b3ab  a2 15    ldx #$15     ;flag ?ILLEGAL DIRECT
., b3ad  2c a2 1b bit $1ba2    ;mask - flag ?UNDEF'D
                               FUNCTION
., b3b0  4c 37 a4 jmp $a437    ;do error
```

46003 DEF: PERFORM DEF

The syntax of FN is checked, program mode confirmed, a left bracket checked for and SUBFLG set. The variable is identified, confirmed as numeric and right bracket confirmed. Once the token = is confirmed, it is pushed onto the stack along with VARPNT and TXTPTR.

```
., b3b3  20 e1 b3 jsr $b3e1    ;check syntax of FN
., b3b6  20 a6 b3 jsr $b3a6    ;confirm program mode
., b3b9  20 fa ae jsr $aefa    ;confirm left bracket
., b3bc  a9 80    lda #$80
., b3be  85 10    sta $10      ;set SUBFLG - user function
                               call
., b3c0  20 8b b0 jsr $b08b    ;identify variable
., b3c3  20 8d ad jsr $ad8d    ;confirm numeric result
., b3c6  20 f7 ae jsr $aef7    ;confirm right bracket
., b3c9  a9 b2    lda #$b2     ;token =
., b3cb  20 ff ae jsr $aeff    ;confirm character in (A)
., b3ce  48       pha
., b3cf  a5 48    lda $48      ;push VARPNT - variable data
                               pointer
., b3d1  48       pha
., b3d2  a5 47    lda $47
., b3d4  48       pha
```

```
., b3d5  a5 7b     lda $7b       ;push TXTPTR
., b3d7  48        pha
., b3d8  a5 7a     lda $7a
., b3da  48        pha
., b3db  20 f8 a8  jsr $a8f8     ;do DATA
., b3de  4c 4f b4  jmp $b44f     ;pull FN data
```

46049 GETFNM: CHECK SYNTAX OF FN

This routine checks that DEF is immediately followed by the keyword FN, and that the dependant variable is numeric.

```
., b3e1  a9 a5     lda #$a5      ;token - FN
., b3e3  20 ff ae  jsr $aeff     ;confirm character in (A)
., b3e6  09 80     ora #$80
., b3e8  85 10     sta $10       ;SUBFLG - user function call
., b3ea  20 92 b0  jsr $b092     ;identify dependant variable
., b3ed  85 4e     sta $4e       ;DEFPNT - pointer to FN
                                 descriptor
., b3ef  84 4f     sty $4f
., b3f1  4c 8d ad  jmp $ad8d     ;confirm numeric result
```

46068 FNDOER: PERFORM FN

This routine evaluates the FN expression and assigns the result to the function variable. The dependant variable X, as in FN(X), is not changed by the routine.

```
., b3f4  20 e1 b3  jsr $b3e1     ;check syntax of FN
., b3f7  a5 4f     lda $4f       ;DEFPNT - pointer to FN
                                 descriptor
., b3f9  48        pha
., b3fa  a5 4e     lda $4e
., b3fc  48        pha
., b3fd  20 f1 ae  jsr $aef1     ;evaluate expression in
                                 brackets
., b400  20 8d ad  jsr $ad8d     ;confirm numeric result
., b403  68        pla
., b404  85 4e     sta $4e       ;DEFPNT
., b406  68        pla
., b407  85 4f     sta $4f
., b409  a0 02     ldy #$02
., b40b  b1 4e     lda ($4e),y
., b40d  85 47     sta $47       ;<VARPNT - pointer to current
                                 variable
., b40f  aa        tax
., b410  c8        iny
., b411  b1 4e     lda ($4e),y
., b413  f0 99     beq $b3ae     ;?UNDEF'D FUNCTION error
```

```
., b415  85 48     sta $48        ;>VARPNT
., b417  c8        iny
., b418  b1 47     lda ($47),y
., b41a  48        pha            ;push dependant variable
., b41b  88        dey
., b41c  10 fa     bpl $b418
., b41e  a4 48     ldy $48        ;>VARPNT
., b420  20 d4 bb  jsr $bbd4      ;store fac#1 in memory
., b423  a5 7b     lda $7b        ;push TXTPTR
., b425  48        pha
., b426  a5 7a     lda $7a
., b428  48        pha
., b429  b1 4e     lda ($4e),y    ;point TXTPTR to FN
                                  expression
., b42b  85 7a     sta $7a
., b42d  c8        iny
., b42e  b1 4e     lda ($4e),y
., b430  85 7b     sta $7b
., b432  a5 48     lda $48
., b434  48        pha
., b435  a5 47     lda $47
., b437  48        pha
., b438  20 8a ad  jsr $ad8a      ;confirm result
., b43b  68        pla
., b43c  85 4e     sta $4e
., b43e  68        pla
., b43f  85 4f     sta $4f
., b441  20 79 00  jsr $0079      ;CHRGOT - get current byte of
                                  text
., b444  f0 03     beq $b449      ;terminator
., b446  4c 08 af  jmp $af08      ;?SYNTAX error
., b449  68        pla
., b44a  85 7a     sta $7a        ;pull original TXTPTR
., b44c  68        pla
., b44d  85 7b     sta $7b
., b44f  a0 00     ldy #$00
., b451  68        pla
., b452  91 4e     sta ($4e),y    ;restore dependant variable
., b454  68        pla
., b455  c8        iny
., b456  91 4e     sta ($4e),y
., b458  68        pla
., b459  c8        iny
., b45a  91 4e     sta ($4e),y
., b45c  68        pla
., b45d  c8        iny
., b45e  91 4e     sta ($4e),y
., b460  68        pla
., b461  c8        iny
```

```
., b462  91 4e     sta ($4e),y
., b464  60        rts
```

46181 STRD: PERFORM STR$

Firstly, a check is made to ensure that the operand is numeric, then it is converted into an ASCII string, and the string set up in memory.

```
., b465  20 8d ad jsr $ad8d    ;confirm numeric operand
., b468  a0 00    ldy #$00
., b46a  20 df bd jsr $bddf    ;convert fac#1 to ASCII
., b46d  68       pla
., b46e  68       pla
., b46f  a9 ff    lda #$ff
., b471  a0 00    ldy #$00
., b473  f0 12    beq $b487    ;set up string at $0100
., b475  a6 64    ldx $64      ;FACHO - fac#1 mantissa
., b477  a4 65    ldy $65
., b479  86 50    stx $50      ;DSCPNT - temp pointer to
                               descriptor
., b47b  84 51    sty $51
., b47d  20 f4 b4 jsr $b4f4    ;allocate space for string
., b480  86 62    stx $62      ;FACHO
., b482  84 63    sty $63
., b484  85 61    sta $61
., b486  60       rts
```

46215 STRLIT: SET UP STRING

On entry, (A/Y) holds start of string -1. $07 and $08 hold the string terminator. This is usually quotes (#22) but can be other characters -colon, comma, etc., depending on the entry point and calling routine; (this routine is also called from INPUT, READ etc). The string is scanned from the start until a valid terminator is found. The length is stored and if the string resides in the input buffer, pointers are allocated and it is stored in high RAM. Finally, the descriptor is saved.

```
., b487  a2 22    ldx #$22     ;ASCII quotes
., b489  86 07    stx $07      ;string terminator
., b48b  86 08    stx $08
., b48d  85 6f    sta $6f      ;start of string -1
., b48f  84 70    sty $70
., b491  85 62    sta $62      ;start of string -1
., b493  84 63    sty $63
., b495  a0 ff    ldy #$ff
., b497  c8       iny
```

```
., b498  b1 6f     lda ($6f),y   ;examine string character
., b49a  f0 0c     beq $b4a8     ;zero terminator
., b49c  c5 07     cmp $07       ;valid terminator?
., b49e  f0 04     beq $b4a4
., b4a0  c5 08     cmp $08       ;valid terminator?
., b4a2  d0 f3     bne $b497     ;no - read next character in
                                 string
., b4a4  c9 22     cmp #$22      ;ASCII quotes
., b4a6  f0 01     beq $b4a9
., b4a8  18        clc
., b4a9  84 61     sty $61       ;(Y) holds string length
., b4ab  98        tya
., b4ac  65 6f     adc $6f
., b4ae  85 71     sta $71
., b4b0  a6 70     ldx $70       ;($71) points to end of
                                 string
., b4b2  90 01     bcc $b4b5
., b4b4  e8        inx
., b4b5  86 72     stx $72
., b4b7  a5 70     lda $70       ;>start of string
., b4b9  f0 04     beq $b4bf
., b4bb  c9 02     cmp #$02      ;input buffer?
., b4bd  d0 0b     bne $b4ca
., b4bf  98        tya
., b4c0  20 75 b4  jsr $b475     ;allocate string pointers
., b4c3  a6 6f     ldx $6f
., b4c5  a4 70     ldy $70
., b4c7  20 88 b6  jsr $b688     ;store string in high RAM
., b4ca  a6 16     ldx $16       ;TEMPPT - descriptor stack
                                 pointer
., b4cc  e0 22     cpx #$22      ;descriptor stack full?
., b4ce  d0 05     bne $b4d5     ;save string descriptor
., b4d0  a2 19     ldx #$19      ;flag ?FORMULA TOO COMPLEX
., b4d2  4c 37 a4  jmp $a437     ;do error
```

46293 PUTNW1: SAVE STRING DESCRIPTOR

The string descriptor is saved on the descriptor stack ($19 - $21). VALTYP is set to string and the descriptor stack pointer is updated.

```
., b4d5  a5 61     lda $61
., b4d7  95 00     sta $00,x     ;(X) points to descriptor
                                 stack
., b4d9  a5 62     lda $62
., b4db  95 01     sta $01,x
., b4dd  a5 63     lda $63
., b4df  95 02     sta $02,x
., b4e1  a0 00     ldy #$00
```

```
., b4e3  86 64     stx $64
., b4e5  84 65     sty $65
., b4e7  84 70     sty $70
., b4e9  88        dey
., b4ea  84 0d     sty $0d        ;set VALTYP to string
., b4ec  86 17     stx $17        ;LASTPT - last temp string
                                  address
., b4ee  e8        inx
., b4ef  e8        inx
., b4f0  e8        inx
., b4f1  86 16     stx $16        ;TEMPPT - descriptor stack
                                  pointer
., b4f3  60        rts
```

46324 GETSPA: ALLOCATE SPACE FOR STRING

The length of the string is pushed onto the stack and FRETOP is decremented by this amount using a 2's complement method. The result is compared with STREND. If they overlap, then garbage is collected and the routine is reentered. If they still overlap then ?OUT OF MEMORY error.

```
., b4f4  46 0f     lsr $0f        ;GARBFL - garbage collect
                                  flag
., b4f6  48        pha            ;push string length
., b4f7  49 ff     eor #$ff
., b4f9  38        sec
., b4fa  65 33     adc $33        ;FRETOP - bottom of strings
., b4fc  a4 34     ldy $34
., b4fe  b0 01     bcs $b501
., b500  88        dey
., b501  c4 32     cpy $32        ;<STREND - end of arrays +1
., b503  90 11     bcc $b516      ;overlap - garbage collect
., b505  d0 04     bne $b50b
., b507  c5 31     cmp $31        ;>STREND
., b509  90 0b     bcc $b516      ;overlap - garbage collect
., b50b  85 33     sta $33
., b50d  84 34     sty $34
., b50f  85 35     sta $35        ;FRESPC - utility string
                                  pointer
., b511  84 36     sty $36
., b513  aa        tax
., b514  68        pla
., b515  60        rts
., b516  a2 10     ldx #$10       ;flag ?OUT OF MEMORY
., b518  a5 0f     lda $0f        ;GARBFL
., b51a  30 b6     bmi $b4d2      ;flag set - do error
., b51c  20 26 b5  jsr $b526      ;do garbage collect
., b51f  a9 80     lda #$80
```

```
., b521  85 0f     sta $0f      ;set GARBFL
., b523  68        pla
., b524  d0 d0     bne $b4f6    ;re-allocate string space
```

46374 GARBAG: GARBAGE COLLECTION

This is a routine to tidy up unwanted strings held in the high end of RAM. Unwanted strings are built up every time a dynamic string is re-allocated, since the old value is not removed. In BASIC 2, garbage collection is notoriously slow when compared to later versions.

```
., b526  a6 37     ldx $37      ;MEMSIZ- highest address in
                                BASIC text
., b528  a5 38     lda $38
., b52a  86 33     stx $33      ;FRETOP - bottom of strings
., b52c  85 34     sta $34
., b52e  a0 00     ldy #$00
., b530  84 4f     sty $4f      ;pointer to unwanted string
., b532  84 4e     sty $4e
., b534  a5 31     lda $31      ;STREND - end of arrays +1
., b536  a6 32     ldx $32
., b538  85 5f     sta $5f
., b53a  86 60     stx $60
., b53c  a9 19     lda #$19
., b53e  a2 00     ldx #$00     ;pointer to TEMPST -
                                descriptor stack
., b540  85 22     sta $22      ;INDEX1 points to TEMPST
., b542  86 23     stx $23
., b544  c5 16     cmp $16      ;TEMPPT - descriptor stack
                                pointer
., b546  f0 05     beq $b54d
., b548  20 c7 b5  jsr $b5c7    ;get descriptor for
                                collection
., b54b  f0 f7     beq $b544
., b54d  a9 07     lda #$07
., b54f  85 53     sta $53      ;length of variable data = 7
                                bytes
., b551  a5 2d     lda $2d      ;VARTAB - start of variables
., b553  a6 2e     ldx $2e
., b555  85 22     sta $22      ;INDEX1 - current var under
                                test
., b557  86 23     stx $23
., b559  e4 30     cpx $30      ;ARYTAB - end of arrays
., b55b  d0 04     bne $b561
., b55d  c5 2f     cmp $2f
., b55f  f0 05     beq $b566
., b561  20 bd b5  jsr $b5bd    ;find string descriptor for
                                collection
```

```
., b564  f0 f3    beq $b559
., b566  85 58    sta $58       ;points to next variable in
                                table
., b568  86 59    stx $59
., b56a  a9 03    lda #$03
., b56c  85 53    sta $53       ;length of variable data = 3
                                bytes
., b56e  a5 58    lda $58       ;next variable in table
., b570  a6 59    ldx $59
., b572  e4 32    cpx $32       ;STREND
., b574  d0 07    bne $b57d
., b576  c5 31    cmp $31
., b578  d0 03    bne $b57d
., b57a  4c 06 b6 jmp $b606     ;collect string
., b57d  85 22    sta $22       ;INDEX1
., b57f  86 23    stx $23
., b581  a0 00    ldy #$00
., b583  b1 22    lda ($22),y
., b585  aa       tax
., b586  c8       iny
., b587  b1 22    lda ($22),y
., b589  08       php
., b58a  c8       iny
., b58b  b1 22    lda ($22),y
., b58d  65 58    adc $58
., b58f  85 58    sta $58
., b591  c8       iny
., b592  b1 22    lda ($22),y
., b594  65 59    adc $59
., b596  85 59    sta $59
., b598  28       plp
., b599  10 d3    bpl $b56e
., b59b  8a       txa
., b59c  30 d0    bmi $b56e
., b59e  c8       iny
., b59f  b1 22    lda ($22),y
., b5a1  a0 00    ldy #$00
., b5a3  0a       asl
., b5a4  69 05    adc #$05
., b5a6  65 22    adc $22
., b5a8  85 22    sta $22
., b5aa  90 02    bcc $b5ae
., b5ac  e6 23    inc $23
., b5ae  a6 23    ldx $23
., b5b0  e4 59    cpx $59
., b5b2  d0 04    bne $b5b8
., b5b4  c5 58    cmp $58
., b5b6  f0 ba    beq $b572
., b5b8  20 c7 b5 jsr $b5c7
```

```
., b5bb  f0 f3    beq $b5b0
```

46525 DVARS: SEARCH FOR NEXT STRING

The variable pointed to by INDEX1 is scanned to ensure that it is (1) a string variable and (2) not a null string. The descriptor is placed in (A/X) and compared with FRETOP. Assuming this to be ok, it is stored in ($5F) and the pointer to the variable in ($4E). INDEX1 is updated to point to the next variable by adding the variable length held in $53.

```
., b5bd  b1 22    lda ($22),y  ;1st character in variable
                               name
., b5bf  30 35    bmi $b5f6    ;integer - wrong type
., b5c1  c8       iny
., b5c2  b1 22    lda ($22),y  ;2nd character in variable
                               name
., b5c4  10 30    bpl $b5f6    ;flpt/function - wrong type
., b5c6  c8       iny
., b5c7  b1 22    lda ($22),y  ;get string length
., b5c9  f0 2b    beq $b5f6    ;null string
., b5cb  c8       iny
., b5cc  b1 22    lda ($22),y  ;string address low
., b5ce  aa       tax
., b5cf  c8       iny
., b5d0  b1 22    lda ($22),y  ;string address high
., b5d2  c5 34    cmp $34      ;>FRETOP - bottom of strings
., b5d4  90 06    bcc $b5dc
., b5d6  d0 1e    bne $b5f6
., b5d8  e4 33    cpx $33      ;<FRETOP
., b5da  b0 1a    bcs $b5f6
., b5dc  c5 60    cmp $60
., b5de  90 16    bcc $b5f6
., b5e0  d0 04    bne $b5e6
., b5e2  e4 5f    cpx $5f
., b5e4  90 10    bcc $b5f6
., b5e6  86 5f    stx $5f
., b5e8  85 60    sta $60
., b5ea  a5 22    lda $22      ;INDEX1 - pointer to current
                               variable
., b5ec  a6 23    ldx $23
., b5ee  85 4e    sta $4e      ;pointer to unwanted variable
., b5f0  86 4f    stx $4f
., b5f2  a5 53    lda $53      ;length of variable data
., b5f4  85 55    sta $55
., b5f6  a5 53    lda $53
., b5f8  18       clc
., b5f9  65 22    adc $22      ;INDEX1 points to next
```

```
                                    variable
., b5fb  85 22    sta $22
., b5fd  90 02    bcc $b601
., b5ff  e6 23    inc $23
., b601  a6 23    ldx $23
., b603  a0 00    ldy #$00
., b605  60       rts
```

46598 GRBPAS: COLLECT A STRING

The length of the string is obtained and added to its pointer. The pointer to the end of the string is then stored in ($5A). A space is opened and the string is moved up over that which was not required. Finally the pointers are updated and the main garbage collect routine reentered.

```
., b606  a5 4f    lda $4f       ;pointer to unwanted string
., b608  05 4e    ora $4e
., b60a  f0 f5    beq $b601     ;nothing to collect - end
., b60c  a5 55    lda $55
., b60e  29 04    and #$04
., b610  4a       lsr
., b611  a8       tay
., b612  85 55    sta $55
., b614  b1 4e    lda ($4e),y   ;get length of string
., b616  65 5f    adc $5f       ;top of string = ($5F)
., b618  85 5a    sta $5a       ;bottom of string = ($5A)
., b61a  a5 60    lda $60
., b61c  69 00    adc #$00
., b61e  85 5b    sta $5b
., b620  a5 33    lda $33       ;FRETOP - bottom of strings
., b622  a6 34    ldx $34
., b624  85 58    sta $58
., b626  86 59    stx $59
., b628  20 bf a3 jsr $a3bf     ;move memory
., b62b  a4 55    ldy $55
., b62d  c8       iny
., b62e  a5 58    lda $58
., b630  91 4e    sta ($4e),y   ;update pointers
., b632  aa       tax
., b633  e6 59    inc $59
., b635  a5 59    lda $59
., b637  c8       iny
., b638  91 4e    sta ($4e),y
., b63a  4c 2a b5 jmp $b52a
```

46653 CAT: CONCATENATE TWO STRINGS

This routine adds one string onto the end of a second

string. ($64) must point to the first string. This is pushed onto the stack and the second string evaluated. The pointer to the first string is then recovered into ($6F), and the lengths of the strings are added. The total length must not exceed 255 characters or ?STRING TOO LONG. Pointers are allocated to the new string and the old strings are stored side by side in high RAM. Finally expression evaluation is continued.

```
., b63d  a5 65     lda $65      ;($64) points to string 1
., b63f  48        pha
., b640  a5 64     lda $64
., b642  48        pha
., b643  20 83 ae  jsr $ae83    ;do single term in expression
., b646  20 8f ad  jsr $ad8f    ;confirm string result
., b649  68        pla
., b64a  85 6f     sta $6f      ;pull string 1 into ($6F)
., b64c  68        pla
., b64d  85 70     sta $70
., b64f  a0 00     ldy #$00
., b651  b1 6f     lda ($6f),y  ;get length of string 1
., b653  18        clc
., b654  71 64     adc ($64),y  ;add it to that of string 2
., b656  90 05     bcc $b65d    ;<255 characters long - ok
., b658  a2 17     ldx #$17     ;flag ?STRING TOO LONG
., b65a  4c 37 a4  jmp $a437    ;do error
., b65d  20 75 b4  jsr $b475    ;allocate string space
., b660  20 7a b6  jsr $b67a    ;store string 1 in high RAM
., b663  a5 50     lda $50
., b665  a4 51     ldy $51
., b667  20 aa b6  jsr $b6aa    ;do string housekeeping
., b66a  20 8c b6  jsr $b68c    ;store string 2 beside string
                                 1
., b66d  a5 6f     lda $6f
., b66f  a4 70     ldy $70
., b671  20 aa b6  jsr $b6aa    ;do string housekeeping
., b674  20 ca b4  jsr $b4ca    ;save new string descriptor
., b677  4c b8 ad  jmp $adb8    ;continue evaluation of text
```

46714 MOVINS: STORE STRING IN HIGH RAM

This routine stores a string at the bottom of the dynamic string area. On entry, ($6F) points to the byte following the variable name or start of string. Its length and pointer are placed in (A), (X) and (Y) respectively. The pointer is then placed in INDEX1. This method provides for several entry points to the routine. The string is finally stored in RAM with the aid of ($35).

```
., b67a  a0 00     ldy #$00
., b67c  b1 6f     lda ($6f),y  ;push string length
., b67e  48        pha
., b67f  c8        iny
., b680  b1 6f     lda ($6f),y  ;pointer to string in (X/Y)
., b682  aa        tax
., b683  c8        iny
., b684  b1 6f     lda ($6f),y
., b686  a8        tay
., b687  68        pla          ;pull string length
., b688  86 22     stx $22      ;INDEX1 - pointer to string
., b68a  84 23     sty $23
., b68c  a8        tay
., b68d  f0 0a     beq $b699    ;null string - nothing to
                                store!
., b68f  48        pha
., b690  88        dey
., b691  b1 22     lda ($22),y
., b693  91 35     sta ($35),y  ;store string in RAM
., b695  98        tya
., b696  d0 f8     bne $b690
., b698  68        pla
., b699  18        clc
., b69a  65 35     adc $35      ;FRESPC - utility string
                                pointer
., b69c  85 35     sta $35      ;points to next store for
                                strings
., b69e  90 02     bcc $b6a2
., b6a0  e6 36     inc $36
., b6a2  60        rts
```

46755 FRESTR: PERFORM STRING HOUSEKEEPING

String mode is confirmed and the string descriptor placed in INDEX1. The following routine is used to clean the descriptor stack, then (A) is loaded with string length, and (X/Y) with its start location. If the string was the last to be defined then FRETOP is moved up to overwrite it. On exit, INDEX1 points to the string.

```
., b6a3  20 8f ad  jsr $ad8f    ;confirm string mode
., b6a6  a5 64     lda $64      ;pointer to string descriptor
., b6a8  a4 65     ldy $65
., b6aa  85 22     sta $22      ;INDEX1
., b6ac  84 23     sty $23
., b6ae  20 db b6  jsr $b6db    ;clean descriptor stack
., b6b1  08        php
., b6b2  a0 00     ldy #$00
., b6b4  b1 22     lda ($22),y  ;push string length
```

```
., b6b6  48        pha
., b6b7  c8        iny
., b6b8  b1 22     lda ($22),y  ;get string pointer
., b6ba  aa        tax
., b6bb  c8        iny
., b6bc  b1 22     lda ($22),y
., b6be  a8        tay
., b6bf  68        pla          ;pull length
., b6c0  28        plp
., b6c1  d0 13     bne $b6d6
., b6c3  c4 34     cpy $34      ;FRETOP - bottom of strings
., b6c5  d0 0f     bne $b6d6
., b6c7  e4 33     cpx $33
., b6c9  d0 0b     bne $b6d6
., b6cb  48        pha
., b6cc  18        clc
., b6cd  65 33     adc $33      ;add length to FRETOP
., b6cf  85 33     sta $33
., b6d1  90 02     bcc $b6d5
., b6d3  e6 34     inc $34
., b6d5  68        pla
., b6d6  86 22     stx $22
., b6d8  84 23     sty $23
., b6da  60        rts
```

46811 FREFAC: CLEAN DESCRIPTOR STACK

On entry, (A/Y) holds a string vector. If this matches that held in LASTPT then the descriptor stack is cleaned.

```
., b6db  c4 18     cpy $18      ;LASTPT - last temp string
                                address
., b6dd  d0 0c     bne $b6eb
., b6df  c5 17     cmp $17
., b6e1  d0 08     bne $b6eb
., b6e3  85 16     sta $16      ;TEMPPT - pointer to temp
                                string stack
., b6e5  e9 03     sbc #$03
., b6e7  85 17     sta $17      ;<LASTPT
., b6e9  a0 00     ldy #$00
., b6eb  60        rts
```

46828 CHRD: PERFORM CHR$

X$=CHR$(Y). The one-byte parameter following the token is input and pointers allocated for a string of length 1. The string character is stored and the descriptor saved.

```
., b6ec  20 a1 b7 jsr $b7a1     ;evaluate text to 1 byte in
```

```
                                   (X)
., b6ef  8a         txa
., b6f0  48         pha
., b6f1  a9 01      lda #$01
., b6f3  20 7d b4 jsr $b47d      ;allocate space for 1 byte
                                 string
., b6f6  68         pla
., b6f7  a0 00      ldy #$00
., b6f9  91 62      sta ($62),y  ;store string character in
                                 RAM
., b6fb  68         pla
., b6fc  68         pla
., b6fd  4c ca b4 jmp $b4ca      ;save string descriptor
```

46848 LEFTD: PERFORM LEFT$

X$=LEFT$(Y$,Z). The string parameters for the new substring are pulled from the stack. ($50) holds the vector to the string. Pointers are allocated and housekeeping performed. The new string is finally stored in high RAM. If Z is specified greater than the length of Y$, then X$ is made equal to Y$.

```
., b700  20 61 b7 jsr $b761      ;pull string parameters
., b703  d1 50      cmp ($50),y  ;string descriptor
., b705  98         tya
., b706  90 04      bcc $b70c
., b708  b1 50      lda ($50),y
., b70a  aa         tax
., b70b  98         tya
., b70c  48         pha
., b70d  8a         txa
., b70e  48         pha
., b70f  20 7d b4 jsr $b47d      ;allocate string space
., b712  a5 50      lda $50
., b714  a4 51      ldy $51
., b716  20 aa b6 jsr $b6aa      ;do string housekeeping
., b719  68         pla
., b71a  a8         tay
., b71b  68         pla
., b71c  18         clc
., b71d  65 22      adc $22      ;add string length to INDEX1
., b71f  85 22      sta $22
., b721  90 02      bcc $b725
., b723  e6 23      inc $23
., b725  98         tya
., b726  20 8c b6 jsr $b68c      ;do string housekeeping
., b729  4c ca b4 jmp $b4ca      ;save string descriptor
```

46892 RIGHTD: PERFORM RIGHT$

X$=RIGHT$(Y$,Z). The string parameters for the new substring are pulled. The length of the parent string is reduced by the right parameter, ensuring that it does not reach beyond the end of the parent string. Carry is used to select the shorter length in this case. Finally LEFT$ is performed.

```
., b72c  20 61 b7 jsr $b761    ;pull string parameters
., b72f  18       clc
., b730  f1 50    sbc ($50),y  ;subtract length from parent
                               string
., b732  49 ff    eor #$ff
., b734  4c 06 b7 jmp $b706    ;do LEFT$
```

46903 MIDD: PERFORM MID$

W$=MID$(X$,Y,Z). $65 holds the right limit. This defaults to #FF. CHRGOT checks for a right bracket, indicating only two parameters, in which case the default limit is set. The string parameters are pulled and if the length of the resultant substring is zero ?ILLEGAL QUANTITY error is called. LEFT$ is performed.

```
., b737  a9 ff    lda #$ff
., b739  85 65    sta $65      ;set default value of right
                               limit
., b73b  20 79 00 jsr $0079    ;CHRGOT
., b73e  c9 29    cmp #$29     ;ASCII )?
., b740  f0 06    beq $b748
., b742  20 fd ae jsr $aefd    ;confirm comma
., b745  20 9e b7 jsr $b79e    ;evaluate text to 1 byte in
                               (X)
., b748  20 61 b7 jsr $b761    ;pull string parameters
., b74b  f0 4b    beq $b798    ;?ILLEGAL QUANTITY ERROR
., b74d  ca       dex
., b74e  8a       txa
., b74f  48       pha
., b750  18       clc
., b751  a2 00    ldx #$00
., b753  f1 50    sbc ($50),y
., b755  b0 b6    bcs $b70d    ;do LEFT$
., b757  49 ff    eor #$ff
., b759  c5 65    cmp $65      ;right limit
., b75b  90 b1    bcc $b70e    ;do LEFT$
., b75d  a5 65    lda $65      ;right limit
., b75f  b0 ad    bcs $b70e    ;do LEFT$
```

46945 PREAM: PULL STRING PARAMETERS

This routine is called by all of the string functions. A right bracket is confirmed, then the string pointer is pulled into ($50) and the string length is pulled into both (X) and (A). All additional parameters pulled in the process are replaced on the stack.

```
., b761  20 f7 ae jsr $aef7     ;confirm right bracket
., b764  68       pla
., b765  a8       tay           ;pull (Y)
., b766  68       pla
., b767  85 55    sta $55       ;pull $55
., b769  68       pla
., b76a  68       pla
., b76b  68       pla
., b76c  aa       tax           ;pull string length
., b76d  68       pla
., b76e  85 50    sta $50       ;pull string descriptor
., b770  68       pla
., b771  85 51    sta $51
., b773  a5 55    lda $55       ;push $55
., b775  48       pha
., b776  98       tya           ;push (Y)
., b777  48       pha
., b778  a0 00    ldy #$00
., b77a  8a       txa           ;string length
., b77b  60       rts
```

46972 LEN: PERFORM LEN

This routine returns the length of a string. String mode is exited and the length of the string (in (Y)) is then stored in fac#1.

```
., b77c  20 82 b7 jsr $b782     exit string mode
., b77f  4c a2 b3 jmp $b3a2     convert (Y) to flpt in fac#1
```

46978 LEN1: EXIT STRING MODE

The string is checked for and pointed to by the string housekeeping routine then numeric mode is set. On exit, INDEX1 points to the string and (Y) holds its length.

```
., b782  20 a3 b6 jsr $b6a3     ;do string housekeeping
., b785  a2 00    ldx #$00
., b787  86 0d    stx $0d       ;VALTYP - set numeric mode
., b789  a8       tay
., b78a  60       rts
```

46987 ASC: PERFORM ASC

String mode is exited, If the length of the string is zero then ?ILLEGAL QUANTITY. Finally, the first character in the string is converted to flpt.

```
., b78b  20 82 b7 jsr $b782     ;exit string mode
., b78e  f0 08    beq $b798     ;?ILLEGAL QUANTITY error
., b790  a000     ldy #$00
., b792  b1 22    lda ($22),y   ;get 1st character in string
., b794  a8       tay
., b795  4c a2 b3 jmp $b248     ;convert (Y) to flpt in fac#1
., b798  4c 48 b2 jmp $b248     ;?ILLEGAL QUANTITY ERROR
```

47003 GTBYTC: EVALUATE TEXT TO 1 BYTE IN (X)

This routine evaluates an ASCII number in the range 0-255 into a single byte value in (X). Text is evaluated and a numeric result confirmed. This is converted into a two-byte parameter of which the high byte must be zero. Finally the result is placed in (X) and the current byte of text in (A).

```
., b79b  20 73 00 jsr $0073     ;CHRGET
., b79e  20 8a ad jsr $ad8a     ;evaluate expression &
                                 confirm numeric
., b7a1  20 b8 b1 jsr $b1b8     ;convert to integer in ($64)
., b7a4  a6 64    ldx $64       ;integer hi byte must be
                                 zero, or...
., b7a6  d0 f0    bne $b798     ;?ILLEGAL QUANTITY error
., b7a8  a6 65    ldx $65       ;put parameter in (X)
., b7aa  4c 79 00 jmp $0079     ;CHRGOT
```

47021 VAL: PERFORM VAL

String mode is exited. If the string pointed to is null, then fac#1 is set to zero. Otherwise the string is converted to flpt by the next routine.

```
., b7ad  20 82 b7 jsr $b782     ;get string and exit string
                                 mode
., b7b0  d0 03    bne $b7b5     ;convert to flpt
., b7b2  4c f7 b8 jmp $b8f7     ;set fac#1 to zero
```

47029 STRVAL: CONVERT ASCII STRING TO FLPT

TXTPTR is placed into a temporary store for the duration of this routine. INDEX1 must point to the string, and (A) hold its length. The end of the string is found and its terminator replaced by #00 as the valid terminator for the

routine to convert text to a flpt number. TXTPTR is restored and the routine exits.

```
., b7b5  a6 7a     ldx $7a        ;TXTPTR
., b7b7  a4 7b     ldy $7b
., b7b9  86 71     stx $71        ;FBUFPT - temp store for
                                  TXTPTR
., b7bb  84 72     sty $72
., b7bd  a6 22     ldx $22        ;INDEX1 - points to string
., b7bf  86 7a     stx $7a        ;store in TXTPTR
., b7c1  18        clc
., b7c2  65 22     adc $22        ;add length of string to
                                  TXTPTR
., b7c4  85 24     sta $24        ;end of string pointer
., b7c6  a6 23     ldx $23        ;put >INDEX1 in TXTPTR
., b7c8  86 7b     stx $7b
., b7ca  90 01     bcc $b7cd
., b7cc  e8        inx
., b7cd  86 25     stx $25
., b7cf  a0 00     ldy #$00
., b7d1  b1 24     lda ($24),y    ;replace string terminator
                                  with #00
., b7d3  48        pha            ;and push original
., b7d4  98        tya
., b7d5  91 24     sta ($24),y
., b7d7  20 79 00  jsr $0079      ;CHRGOT
., b7da  20 f3 bc  jsr $bcf3      ;get number from text into
                                  fac#1
., b7dd  68        pla
., b7de  a0 00     ldy #$00
., b7e0  91 24     sta ($24),y    ;replace original terminator
., b7e2  a6 71     ldx $71        ;restore TXTPTR from FBUFPT
., b7e4  a4 72     ldy $72
., b7e6  86 7a     stx $7a
., b7e8  84 7b     sty $7b
., b7ea  60        rts
```

47083 GETNUM: GET PARAMETERS FOR POKE/WAIT

This routine gets the parameters for, say, POKE 5432,1 and WAIT 1234,5,6. Firstly the expression in text is evaluated and confirmed numeric. FAC#1 is converted to a 2-byte integer in LINNUM and the 1-byte parameter after the comma is evaluated into (X). For its other parameter, WAIT must reenter the routine.

```
., b7eb  20 8a ad  jsr $ad8a      ;evaluate expression &
                                  confirm numeric
., b7ee  20 f7 b7  jsr $b7f7      ;convert fac#1 to integer in
```

```
                                   LINNUM
., b7f1  20 fd ae jsr $aefd       ;confirm comma
., b7f4  4c 9e b7 jmp $b79e       ;evaluate text to 1 byte in
                                   (X)
```

47095 GETADR: CONVERT FAC#1 TO INTEGER IN LINNUM

The sign and exponent of fac#1 are checked to ensure the number is positive and below 65536. FAC#1 is converted into a 4-byte integer of which the two low bytes are stored in LINNUM.

```
., b7f7  a5 66     lda $66        ;FACSGN - fac#1 sign byte
., b7f9  30 9d     bmi $b798      ;?ILLEGAL QUANTITY error
., b7fb  a5 61     lda $61        ;FACEXP - fac#1 exponent
., b7fd  c9 91     cmp #$91
., b7ff  b0 97     bcs $b798      ;?ILLEGAL QUANTITY error
., b801  20 9b bc  jsr $bc9b      ;convert fac#1 to 4-byte
                                   integer
., b804  a5 64     lda $64        ;put low two bytes in LINNUM
., b806  a4 65     ldy $65
., b808  84 14     sty $14        ;LINNUM - integer value
., b80a  85 15     sta $15
., b80c  60        rts
```

47117 PEEK: PERFORM PEEK

LINNUM is pushed onto the stack, then the flpt address in fac#1 is converted to an integer in LINNUM. The address pointed to is peeked and the result put in (Y). This is converted to flpt and LINNUM restored to its original value.

```
., b80d  a5 15     lda $15        ;push LINNUM - integer value
., b80f  48        pha
., b810  a5 14     lda $14
., b812  48        pha
., b813  20 f7 b7  jsr $b7f7      ;convert fac#1 to integer in
                                   LINNUM
., b816  a0 00     ldy #$00
., b818  b1 14     lda ($14),y    ;peek contents of address
                                   into (A)
., b81a  a8        tay
., b81b  68        pla
., b81c  85 14     sta $14        ;restore LINNUM from stack
., b81e  68        pla
., b81f  85 15     sta $15
., b821  4c a2 b3  jmp $b3a2      ;convert integer in (Y) to
                                   flpt
```

47140 POKE: PERFORM POKE

The parameters are obtained, the address to be poked is in LINNUM and the value in (X). This value is then poked into the correct address.

```
., b824  20 eb b7 jsr $b7eb    ;get parameters for POKE
., b827  8a       txa
., b828  a0 00    ldy #$00
., b82a  91 14    sta ($14),y  ;poke value into address at
                                LINNUM
., b82c  60       rts
```

47149 WAIT: PERFORM WAIT

The first two parameters are obtained with the address in LINNUM and value 1 in <FORPNT. If there is a third parameter, this is obtained and the result put in >FORPNT. If there is no third parameter, it defaults to zero. A loop is performed, in which the contents of the address in LINNUM are exclusive-ORed with value 2 and ANDed with value 1. When the result is non-zero, the loop ends.

```
., b82d  20 eb b7 jsr $b7eb    ;get first 2 parameters for
                                WAIT
., b830  86 49    stx $49      ;<FORPNT - value 1
., b832  a2 00    ldx #$00
., b834  20 79 00 jsr $0079    ;GHRGOT
., b837  f0 03    beq $b83c    ;terminator - no third
                                parameter
., b839  20 f1 b7 jsr $b7f1    ;get third parameter for WAIT
., b83c  86 4a    stx $4a      ;>FORPNT - value 2
., b83e  a0 00    ldy #$00
., b840  b1 14    lda ($14),y  ;LINNUM - holds address
., b842  45 4a    eor $4a      ;value 2
., b844  25 49    and $49      ;value 1
., b846  f0 f8    beq $b840
., b848  60       rts
```

47177 FADDH: ADD 0.5 TO FAC#1

The address of the flpt constant 0.5 is placed into (A/Y) and this is then added to fac#1. This routine is commonly used in rounding.

```
., b849  a9 11    lda #$11
., b84b  a0 bf    ldy #$bf     ;$BF11 holds constant 0.5 in
                                flpt
., b84d  4c 67 b8 jmp $b867    ;add (A/Y) to fac#1
```

47184 FSUB: PERFORM SUBTRACTION

fac#1 = fac#2 - fac#1. Firstly fac#2 is loaded from the address at (A/Y) - this must be a 5-byte flpt number. The sign byte of fac#1 is reversed and compared with that of fac#2. Finally the addition routine is called to add the two facs together. Thus the result is fac#1 = fac#2 + (-fac#1).

```
., b850  20 8c ba jsr $ba8c
., b853  a5 66    lda $66
., b855  49 ff    eor #$ff
., b857  85 66    sta $66
., b859  45 6e    eor $6e
., b85b  85 6f    sta $6f
., b85d  a5 61    lda $61
., b85f  4c 6a b8 jmp $b86a
```

47202 FADD5: NORMALISE ADDITION

This is part of the main addition routine below.

```
., b862  20 99 b9 jsr $b999     ;multiply by zero byte
., b865  90 3c    bcc $b8a3     ;check sign and add
```

47207 FADD: PERFORM ADDITION

fac#1 = fac#1 + fac#2. On entry, (A/Y) points to a flpt number which is placed in fac#2. A test is made for fac#1 = zero, in which case fac#2 is simply copied into fac#1. When the two accumulators are added together, they must have equal exponents. If the exponents are not equal, one number must be modified until they are.

```
., b867  20 8c ba jsr $ba8c     ;load fac#2 from memory
., b86a  d0 03    bne $b86f
., b86c  4c fc bb jmp $bbfc     ;copy fac#2 to fac#1
., b86f  a6 70    ldx $70       ;FACOV - fac#1 rounding
., b871  86 56    stx $56
., b873  a2 69    ldx #$69
., b875  a5 69    lda $69       ;ARGEXP - fac#2 exponent
., b877  a8       tay
., b878  f0 ce    beq $b848     ;fac#2 is zero - end
., b87a  38       sec
., b87b  e5 61    sbc $61       ;FACEXP - fac#1 exponent
., b87d  f0 24    beq $b8a3     ;exponents are equal - add
                                numbers
., b87f  90 12    bcc $b893
```

```
., b881  84 61    sty $61       ;FACEXP
., b883  a4 6e    ldy $6e       ;ARGSGN - fac#2 sign
., b885  84 66    sty $66       ;FACSGN - fac#1 sign
., b887  49 ff    eor #$ff
., b889  69 00    adc #$00
., b88b  a0 00    ldy #$00
., b88d  84 56    sty $56
., b88f  a2 61    ldx #$61
., b891  d0 04    bne $b897
., b893  a0 00    ldy #$00
., b895  84 70    sty $70       ;FACOV
., b897  c9 f9    cmp #$f9
., b899  30 c7    bmi $b862     ;normalise addition
., b89b  a8       tay
., b89c  a5 70    lda $70       ;FACOV
., b89e  56 01    lsr $01,x
., b8a0  20 b0 b9 jsr $b9b0
., b8a3  24 6f    bit $6f       ;ARISGN - sign comparison
                                result
., b8a5  10 57    bpl $b8fe     ;add numbers
., b8a7  a0 61    ldy #$61      ;sign -ve - subtract numbers
., b8a9  e0 69    cpx #$69
., b8ab  f0 02    beq $b8af
., b8ad  a0 69    ldy #$69
., b8af  38       sec
., b8b0  49 ff    eor #$ff
., b8b2  65 56    adc $56
., b8b4  85 70    sta $70       ;FACOV
., b8b6  b9 04 00 lda $0004,y   ;points to fac#(2/1)
., b8b9  f5 04    sbc $04,x     ;points to fac#(1/2)
., b8bb  85 65    sta $65       ;store in fac#1 - FACHO4
., b8bd  b9 03 00 lda $0003,y
., b8c0  f5 03    sbc $03,x
., b8c2  85 64    sta $64       ;FACHO3
., b8c4  b9 02 00 lda $0002,y
., b8c7  f5 02    sbc $02,x
., b8c9  85 63    sta $63       ;FACHO2
., b8cb  b9 01 00 lda $0001,y
., b8ce  f5 01    sbc $01,x
., b8d0  85 62    sta $62       ;FACHO1
., b8d2  b0 03    bcs $b8d7
., b8d4  20 47 b9 jsr $b947
., b8d7  a0 00    ldy #$00
., b8d9  98       tya
., b8da  18       clc
., b8db  a6 62    ldx $62       ;FACHO1 - low byte
., b8dd  d0 4a    bne $b929     ;rotate fac right
., b8df  a6 63    ldx $63       ;move fac#1 left 1 byte
., b8e1  86 62    stx $62
```

```
., b8e3  a6 64     ldx $64
., b8e5  86 63     stx $63
., b8e7  a6 65     ldx $65
., b8e9  86 64     stx $64
., b8eb  a6 70     ldx $70        ;put FACOV into FACHO4
., b8ed  86 65     stx $65
., b8ef  84 70     sty $70        ;and (Y) into FACOV
., b8f1  69 08     adc #$08
., b8f3  c9 20     cmp #$20
., b8f5  d0 e4     bne $b8db
., b8f7  a9 00     lda #$00
., b8f9  85 61     sta $61        ;FACEXP - zero fac#1 exponent
., b8fb  85 66     sta $66        ;FACEXP - zero fac#2 exponent
., b8fd  60        rts
., b8fe  65 56     adc $56
., b900  85 70     sta $70        ;FACOV
., b902  a5 65     lda $65        ;add fac#2 to fac#1
., b904  65 6d     adc $6d
., b906  85 65     sta $65
., b908  a5 64     lda $64
., b90a  65 6c     adc $6c
., b90c  85 64     sta $64
., b90e  a5 63     lda $63
., b910  65 6b     adc $6b
., b912  85 63     sta $63
., b914  a5 62     lda $62
., b916  65 6a     adc $6a
., b918  85 62     sta $62
., b91a  4c 36 b9  jmp $b936      ;round fac#1 and end
., b91d  69 01     adc #$01
., b91f  06 70     asl $70        ;FACOV - fac#1 rounding byte
., b921  26 65     rol $65        ;rotate fac#1 left
., b923  26 64     rol $64
., b925  26 63     rol $63
., b927  26 62     rol $62
., b929  10 f2     bpl $b91d
., b92b  38        sec
., b92c  e5 61     sbc $61        ;FACEXP - fac#1 exponent
., b92e  b0 c7     bcs $b8f7
., b930  49 ff     eor #$ff
., b932  69 01     adc #$01
., b934  85 61     sta $61        ;FACEXP
., b936  90 0e     bcc $b946
., b938  e6 61     inc $61        ;FACEXP
., b93a  f0 42     beq $b97e
., b93c  66 62     ror $62        ;rotate fac#1 right
., b93e  66 63     ror $63
., b940  66 64     ror $64
., b942  66 65     ror $65
```

```
., b944  66 70     ror $70
., b946  60        rts
```

47431 NEGFAC: 2'S COMPLEMENT FAC#1

This routine takes the 2's complement of fac#1 and places it in fac#1. All the bits in the accumulator are reversed and 1 is added. Since the accumulator is spread over several bytes, if a carry occurs in a less significant byte, all the others must be updated accordingly.

```
., b947  a5 66     lda $66       ;FACSGN - fac#1 sign byte
., b949  49 ff     eor #$ff
., b94b  85 66     sta $66
., b94d  a5 62     lda $62       ;complement fac#1
., b94f  49 ff     eor #$ff      ;ie make all 0's 1 and 1's 0
., b951  85 62     sta $62
., b953  a5 63     lda $63
., b955  49 ff     eor #$ff
., b957  85 63     sta $63
., b959  a5 64     lda $64
., b95b  49 ff     eor #$ff
., b95d  85 64     sta $64
., b95f  a5 65     lda $65
., b961  49 ff     eor #$ff
., b963  85 65     sta $65
., b965  a5 70     lda $70       ;FACOV - fac#1 rounding byte
., b967  49 ff     eor #$ff
., b969  85 70     sta $70
., b96b  e6 70     inc $70       ;add 1 to make it 2's
                                 complement
., b96d  d0 0e     bne $b97d
., b96f  e6 65     inc $65       ;adjust the rest of fac#1 as
                                 needed
., b971  d0 0a     bne $b97d
., b973  e6 64     inc $64
., b975  d0 06     bne $b97d
., b977  e6 63     inc $63
., b979  d0 02     bne $b97d
., b97b  e6 62     inc $62
., b97d  60        rts
```

47486 OVERR: OUTPUT ?OVERFLOW ERROR

```
., b97e  a2 0f     ldx #$0f      ;flag ?OVERFLOW
., b980  4c 37 a4  jmp $a437     ;do error
```

47491 MULSHF: MULTIPLY BY ZERO BYTE

This routine performs multiplication of zeros contained within the flpt multiplicand. It is called as part of the main multiplication routine.

```
., b983  a2 25     ldx #$25
., b985  b4 04     ldy $04,x     ;RESHO - temp product fac
., b987  84 70     sty $70       ;FACOV - fac#1 rounding byte
., b989  b4 03     ldy $03,x
., b98b  94 04     sty $04,x
., b98d  b4 02     ldy $02,x
., b98f  94 03     sty $03,x
., b991  b4 01     ldy $01,x
., b993  94 02     sty $02,x
., b995  a4 68     ldy $68       ;BITS - fac#1 overflow digit
., b997  94 01     sty $01,x
., b999  69 08     adc #$08
., b99b  30 e8     bmi $b985
., b99d  f0 e6     beq $b985
., b99f  e9 08     sbc #$08
., b9a1  a8        tay           ;(Y) holds count
., b9a2  a5 70     lda $70       ;FACOV
., b9a4  b0 14     bcs $b9ba
., b9a6  16 01     asl $01,x     ;place bit to multiply in
                                 carry
., b9a8  90 02     bcc $b9ac
., b9aa  f6 01     inc $01,x     ;add 1
., b9ac  76 01     ror $01,x     ;adjust product
., b9ae  76 01     ror $01,x
., b9b0  76 02     ror $02,x
., b9b2  76 03     ror $03,x
., b9b4  76 04     ror $04,x
., b9b6  6a        ror
., b9b7  c8        iny           ;increment counter
., b9b8  d0 ec     bne $b9a6
., b9ba  18        clc
., b9bb  60        rts
```

47548 FONE: TABLE OF FLPT CONSTANTS

The table includes the following numbers, starting at the following addresses:

```
$B9Bc = 1
$B9C1 = #3 - counter for LOG series
$B9C2 = 0.434255942 - LOG constant 1
$B9C7 = 0.57658454  - LOG constant 2
$B9CC = 0.961800759 - LOG constant 3
$B9D1 = 2.885390073 - LOG constant 4
$B9D6 = 0.707106781 - SQR(0.5)
```

```
                  $B9DB = 1.41421356  - SQR(2)
                  $B9E0 = -0.5
                  $B9E5 = 0.693147181 - LOG(2)

.:b9bc 81 00 00 00 00 03 7f 5e
.:b9c4 56 cb 79 80 13 9b 0b 64
.:b9cc 80 76 38 93 16 82 38 aa
.:b9d4 3b 20 80 35 04 f3 34 81
.:b9dc 35 04 f3 34 80 80 00 00
.:b9e4 00 80 31 72 17 f8
```

47594 LOG: PERFORM LOG

This routine takes the natural log (base e) of the number in fac#1. The value is calculated by the major series evaluation routine using the values in the above table. The result is stored in fac#1.

```
., b9ea  20 2b bc jsr $bc2b     ;check fac#1 sign
., b9ed  f0 02    beq $b9f1     ;fac#1 is zero
., b9ef  10 03    bpl $b9f4     ;fac#1 is positive
., b9f1  4c 48 b2 jmp $b248     ;?ILLEGAL QUANTITY error
., b9f4  a5 61    lda $61       ;FACEXP -fac#1 exponent
., b9f6  e9 7f    sbc #$7f
., b9f8  48       pha
., b9f9  a9 80    lda #$80
., b9fb  85 61    sta $61       ;FACEXP
., b9fd  a9 d6    lda #$d6
., b9ff  a0 b9    ldy #$b9      ;$B9D6 = SQR(0.5) - flpt
                                constant
., ba01  20 67 b8 jsr $b867     ;add (A/Y) to fac#1
., ba04  a9 db    lda #$db
., ba06  a0 b9    ldy #$b9      ;$B9DB = SQR(2) - flpt
                                constant
., ba08  20 0f bb jsr $bb0f     ;divide (A/Y) by fac#1
., ba0b  a9 bc    lda #$bc
., ba0d  a0 b9    ldy #$b9      ;$B9BC = 1 - flpt constant
., ba0f  20 50 b8 jsr $b850     ;subtract fac#1 from (A/Y)
., ba12  a9 c1    lda #$c1
., ba14  a0 b9    ldy #$b9      ;$B9C1 = #03 - LOG series
                                counter
., ba16  20 43 e0 jsr $e043     ;evaluate series for function
., ba19  a9 e0    lda #$e0
., ba1b  a0 b9    ldy #$b9      ;$B9E0 = -0.5 - flpt constant
., ba1d  20 67 b8 jsr $b867     :add (A/Y) to fac#1
., ba20  68       pla
., ba21  20 7e bd jsr $bd7e
., ba24  a9 e5    lda #$e5
., ba26  a0 b9    ldy #$b9      $B9E5 = LOG(2) - flpt
                                constant
```

47656 FMULT: PERFORM MULTIPLY

fac#1 = fac#2 * fac#1. On entry, (A/Y) must hold the address of the flpt number to be put in fac#2. FACEXP is checked, and if zero, then fac#1 is zero and the routine ends. The accumulators are tested for a potential under/overflow and the two accumulators are multiplied together byte by byte using the following routine. The intermediate result is kept in RESHO, a temporary accumulator held at $26 - $29.

```
., ba28  20 8c ba jsr $ba8c    ;load fac#2 from flpt at
                               (A/Y)
., ba2b  d0 03    bne $ba30    ;(A) holds FACEXP - fac#1
                               exponent
., ba2d  4c 8b ba jmp $ba8b    ;fac#1 is zero - end
., ba30  20 b7 ba jsr $bab7    ;test both accumulators
., ba33  a9 00    lda #$00
., ba35  85 26    sta $26      ;set RESHO - intermediate
                               product =0
., ba37  85 27    sta $27
., ba39  85 28    sta $28
., ba3b  85 29    sta $29
., ba3d  a5 70    lda $70      ;FACOV - fac#1 rounding byte
., ba3f  20 59 ba jsr $ba59    ;multiply by a byte
., ba42  a5 65    lda $65      ;FACHO4
., ba44  20 59 ba jsr $ba59    ;multiply by a byte
., ba47  a5 64    lda $64      ;FACHO3
., ba49  20 59 ba jsr $ba59    ;multiply by a byte
., ba4c  a5 63    lda $63      ;FACHO2
., ba4e  20 59 ba jsr $ba59    ;multiply by a byte
., ba51  a5 62    lda $62      ;FACHO1
., ba53  20 5e ba jsr $ba5e    ;multiply by a byte
., ba56  4c 8f bb jmp $bb8f    ;put RESHO into fac#1
```

47705 MULTPLY: MULTIPLY BY A BYTE

A zero byte is tested for and, if found, processed by its own routine. The carry flag is used to determine multiply; if set then fac#2 is added to the temporary accumulator, RESHO. The product is rotated left in readiness for the next bit, and (A) is shifted left, putting the next bit in C and introducing a zero. The routine ends when all 8 bits have been processed.

```
., ba59  d0 03    bne $ba5e
., ba5b  4c 83 b9 jmp $b983    ;multiply by a zero byte
```

```
., ba5e  4a       lsr
., ba5f  09 80    ora #$80
., ba61  a8       tay            ;holds byte for multiply
., ba62  90 19    bcc $ba7d      ;carry clear - don't add
., ba64  18       clc
., ba65  a5 29    lda $29        ;add fac#2 to RESHO - temp
                                 fac
., ba67  65 6d    adc $6d
., ba69  85 29    sta $29
., ba6b  a5 28    lda $28
., ba6d  65 6c    adc $6c
., ba6f  85 28    sta $28
., ba71  a5 27    lda $27
., ba73  65 6b    adc $6b
., ba75  85 27    sta $27
., ba77  a5 26    lda $26
., ba79  65 6a    adc $6a
., ba7b  85 26    sta $26
., ba7d  66 26    ror $26        ;rotate RESHO
., ba7f  66 27    ror $27
., ba81  66 28    ror $28
., ba83  66 29    ror $29
., ba85  66 70    ror $70
., ba87  98       tya            ;byte for multiply
., ba88  4a       lsr            ;put next bit in sequence in
                                 carry
., ba89  d0 d6    bne $ba61      ;loop through byte sequence
., ba8b  60       rts
```

47756 CONUPK: LOAD FAC#2 FROM MEMORY

On entry, (A/Y) must hold the address of a 5-byte flpt number. This address is transferred to INDEX1 and the number placed in fac#2. The sign bit is unpacked from the mantissa at this point and stored separately in ARGSGN. On exit, the exponent of fac#1 is put in (A).

```
., ba8c  85 22    sta $22        ;INDEX1 - points to start of
                                 flpt
., ba8e  84 23    sty $23
., ba90  a0 04    ldy #$04       ;store mantissa in ARGHO -
                                 fac#2
., ba92  b1 22    lda ($22),y
., ba94  85 6d    sta $6d        ;ARGHO4
., ba96  88       dey
., ba97  b1 22    lda ($22),y
., ba99  85 6c    sta $6c        ;ARGHO3
., ba9b  88       dey
., ba9c  b1 22    lda ($22),y
```

```
., ba9e  85 6b     sta $6b        ;ARGHO2
., baa0  88        dey
., baa1  b1 22     lda ($22),y    ;get lsb/sign for unpacking
., baa3  85 6e     sta $6e        ;ARGSGN - fac#2 sign
., baa5  45 66     eor $66        ;FACSGN - fac#1 sign
., baa7  85 6f     sta $6f        ;ARISGN - sign comparison
                                  result
., baa9  a5 6e     lda $6e        ;ARGSGN
., baab  09 80     ora #$80
., baad  85 6a     sta $6a        ;ARGHO1
., baaf  88        dey
., bab0  b1 22     lda ($22),y
., bab2  85 69     sta $69        ;ARGEXP - fac#2 exponent
., bab4  a5 61     lda $61        ;FACEXP - fac#1 exponent
., bab6  60        rts
```

47799 MULDIV: TEST BOTH ACCUMULATORS

This routine checks for possible overflow/underflow of both floating point accumulators. If an overflow or underflow occurs, then it is passed to the next routine for processing.

```
., bab7  a5 69     lda $69        ;ARGEXP - fac#2 exponent
., bab9  f0 1f     beq $bada      ;fac#2 is zero - cause
                                  underflow
., babb  18        clc
., babc  65 61     adc $61        ;FACEXP - fac#1 exponent
., babe  90 04     bcc $bac4
., bac0  30 1d     bmi $badf      ;?OVERFLOW error
., bac2  18        clc
., bac3  2c 10 14  bit $1410      ;mask - BPL $BADA - do
                                  underflow
., bac6  69 80     adc #$80
., bac8  85 61     sta $61        ;FACEXP
., baca  d0 03     bne $bacf
., bacc  4c fb b8  jmp $b8fb      ;add fac#2 to fac#1
., bacf  a5 6f     lda $6f        ;ARISGN - sign comparison
                                  result
., bad1  85 66     sta $66        ;FACSGN - fac#1 sign
., bad3  60        rts
```

47828 MLDVEX: OVERFLOW/UNDERFLOW

FACSGN is tested and the error processed according to the result. The return address is pulled from the stack and discarded. If an underflow has occurred, both accumulators are set to zero. If an overflow has occurred, then ?OVERFLOW error is called.

```
., bad4  a5 66     lda $66       ;FACSGN - fac#1 sign
., bad6  49 ff     eor #$ff
., bad8  30 05     bmi $badf
., bada  68        pla           ;discard return address
., badb  68        pla
., badc  4c f7 b8  jmp $b8f7     ;zero both fac's
., badf  4c 7e b9  jmp $b97e     ;?OVERFLOW error
```

47842 MUL10: MULTIPLY FAC#1 BY 10

fac#1 = fac#1 * 10. The flpt constant 10 in ROM is not used, but instead the following procedure is carried out - Firstly copy fac#1 into fac#2, then multiply fac#1 by 4. Add fac#2 to this and double the final result. As an example, 2 * 4 = 8, + 2 = 10, * 2 = 20.

```
., bae2  20 0c bc  jsr $bc0c     ;copy fac#1 to fac#2
., bae5  aa        tax
., bae6  f0 10     beq $baf8
., bae8  18        clc
., bae9  69 02     adc #$02
., baeb  b0 f2     bcs $badf     ;?OVERFLOW error
., baed  a2 00     ldx #$00
., baef  86 6f     stx $6f       ;ARISGN - sign comparison
                                 result
., baf1  20 77 b8  jsr $b877     ;perform addition
., baf4  e6 61     inc $61       ;FACEXP - fac#1 exponent
., baf6  f0 e7     beq $badf     ;?OVERFLOW error
., baf8  60        rts
```

47865 TENC: CONSTANT 10 IN FLPT

```
.:baf9 84 20 00 00 00
```

47870 DIV10: DIVIDE FAC#1 BY 10

fac#1 = fac#1 / 10. The routine copies fac#1 into fac#2, sets up pointers to the flpt constant 10 in ROM and enters the following divide routine.

```
., bafe  20 0c bc  jsr $bc0c     ;copy fac#1 to fac#2
., bb01  a9 f9     lda #$f9
., bb03  a0 ba     ldy #$ba      ;$BAF9 = flpt constant 10
., bb05  a2 00     ldx #$00      ;sign comparison result
```

47879 FDIV: DIVIDE FAC#2 BY FLPT AT (A/Y)

fac#1 = fac#2 / flpt at (A/Y). On entry, (A/Y) must point

to a 5-byte flpt number and (X) hold the sign comparison result byte. This number is then loaded into fac#2 and the main divide routine entered.

```
., bb07  86 6f     stx $6f       ;ARISGN - sign comparison
                                 result
., bb09  20 a2 bb  jsr $bba2     ;load fac#1 with flpt at
                                 (A/Y)
., bb0c  4c 12 bb  jmp $bb12
```

47887 FDIVT: PERFORM DIVIDE

fac#1 = fac#2 / fac#1. On entry, (A) must hold FACEXP. The sign comparison result should also be set up on entry. This can be done by ORing FACSGN with ARGSGN. The result should be placed in ARISGN.

```
., bb0f  20 8c ba  jsr $ba8c     ;load fac#2 with flpt at
                                 (A/Y)
., bb12  f0 76     beq $bb8a     ;?DIVISION BY ZERO error
., bb14  20 1b bc  jsr $bc1b     ;round fac#1
., bb17  a9 00     lda #$00
., bb19  38        sec
., bb1a  e5 61     sbc $61       ;FACEXP - fac#1 exponent
., bb1c  85 61     sta $61
., bb1e  20 b7 ba  jsr $bab7     ;test both accumulators
., bb21  e6 61     inc $61       ;FACEXP
., bb23  f0 ba     beq $badf     ;?OVERFLOW error
., bb25  a2 fc     ldx #$fc
., bb27  a9 01     lda #$01
., bb29  a4 6a     ldy $6a       ;ARGHO1 - fac#2 mantissa
., bb2b  c4 62     cpy $62       ;FACHO1 - fac#1 mantissa
., bb2d  d0 10     bne $bb3f
., bb2f  a4 6b     ldy $6b       ;ARGHO2
., bb31  c4 63     cpy $63       ;FACHO2
., bb33  d0 0a     bne $bb3f
., bb35  a4 6c     ldy $6c       ;ARGHO3
., bb37  c4 64     cpy $64       ;FACHO3
., bb39  d0 04     bne $bb3f
., bb3b  a4 6d     ldy $6d       ;ARGHO4
., bb3d  c4 65     cpy $65       ;FACHO4
., bb3f  08        php
., bb40  2a        rol
., bb41  90 09     bcc $bb4c
., bb43  e8        inx
., bb44  95 29     sta $29,x
., bb46  f0 32     beq $bb7a
., bb48  10 34     bpl $bb7e
., bb4a  a9 01     lda #$01
```

```
., bb4c  28        plp
., bb4d  b0 0e     bcs $bb5d
., bb4f  06 6d     asl $6d      ;ARGHO4
., bb51  26 6c     rol $6c      ;ARGHO3
., bb53  26 6b     rol $6b      ;ARGHO2
., bb55  26 6a     rol $6a      ;ARGHO1
., bb57  b0 e6     bcs $bb3f
., bb59  30 ce     bmi $bb29
., bb5b  10 e2     bpl $bb3f
., bb5d  a8        tay
., bb5e  a5 6d     lda $6d      ;ARGHO4 = ARGHO4 - FACHO4
., bb60  e5 65     sbc $65
., bb62  85 6d     sta $6d
., bb64  a5 6c     lda $6c      ;ARGHO3 = ARGHO3 - FACHO3
., bb66  e5 64     sbc $64
., bb68  85 6c     sta $6c
., bb6a  a5 6b     lda $6b      ;ARGHO2 = ARGHO2 - FACHO2
., bb6c  e5 63     sbc $63
., bb6e  85 6b     sta $6b
., bb70  a5 6a     lda $6a      ;ARGHO1 = ARGHO1 - FACHO1
., bb72  e5 62     sbc $62
., bb74  85 6a     sta $6a
., bb76  98        tya
., bb77  4c 4f bb  jmp $bb4f
., bb7a  a9 40     lda #$40
., bb7c  d0 ce     bne $bb4c
., bb7e  0a        asl
., bb7f  0a        asl
., bb80  0a        asl
., bb81  0a        asl
., bb82  0a        asl
., bb83  0a        asl
., bb84  85 70     sta $70      ;FACOV - fac#1 rounding byte
., bb86  28        plp
., bb87  4c 8f bb  jmp $bb8f
., bb8a  a2 14     ldx #$14     ;flag ?DIVISION BY ZERO
., bb8c  4c 37 a4  jmp $a437    ;do error
., bb8f  a5 26     lda $26
., bb91  85 62     sta $62
., bb93  a5 27     lda $27
., bb95  85 63     sta $63
., bb97  a5 28     lda $28
., bb99  85 64     sta $64
., bb9b  a5 29     lda $29
., bb9d  85 65     sta $65
., bb9f  4c d7 b8  jmp $b8d7    ;(add?)
```

48034 MOVFM: LOAD FAC#1 FROM MEMORY

On entry, (A/Y) must point to a 5-byte flpt number. This

address is stored in INDEX1 and from there the number is placed in fac#1. The sign byte is unpacked from bit 7 of FACHO1 and placed in FACSGN.

```
., bba2  85 22     sta $22       ;INDEX1 - points to 5 byte
                                 flpt
., bba4  84 23     sty $23
., bba6  a0 04     ldy #$04
., bba8  b1 22     lda ($22),y   ;get byte from memory
., bbaa  85 65     sta $65       ;FACHO4 - fac#1 mantissa
., bbac  88        dey
., bbad  b1 22     lda ($22),y
., bbaf  85 64     sta $64       ;FACHO3
., bbb1  88        dey
., bbb2  b1 22     lda ($22),y
., bbb4  85 63     sta $63       ;FACHO2
., bbb6  88        dey
., bbb7  b1 22     lda ($22),y   ;get lsb/sign for unpacking
., bbb9  85 66     sta $66       ;FACSGN - fac#1 sign
., bbbb  09 80     ora #$80
., bbbd  85 62     sta $62       ;FACHO1
., bbbf  88        dey
., bbc0  b1 22     lda ($22),y
., bbc2  85 61     sta $61       ;FACEXP - fac#1 exponent
., bbc4  84 70     sty $70       ;FACOV - fac#1 rounding
., bbc6  60        rts
```

48071 MOV2F: STORE FAC#1 IN MEMORY

This routine stores fac#1 in the 5 bytes following the address held in (X/Y). In the process, the sign byte is loaded into bit 7 of FACHO1 and the accumulator is rounded. This ensures that the 6-byte accumulator fits into the 5 byte space allocated. There are four entry points to this routine. The first two point to zero-page locations for calculating TAN and series expansions. They are $005C - $0060 and $0057 - $005B. The third uses FORPNT to point to the current variable address, and the fourth is where (X) and (Y) must be manually set.

```
., bbc7  a2 5c     ldx #$5c      ;entry point for series
                                 evaluation
., bbc9  2c a2 57  bit $57a2     ;mask - LDX #$57 entry point
                                 for TAN
., bbcc  a0 00     ldy #$00      ;hi byte - points to zero
                                 page
., bbce  f0 04     beq $bbd4     ;store fac#1
., bbd0  a6 49     ldx $49       ;FORPNT entry - point to
```

```
                                    variable data
., bbd2  a4 4a     ldy $4a
., bbd4  20 1b bc  jsr $bc1b        ;round fac#1
., bbd7  86 22     stx $22          ;INDEX1 - location for
                                    storage
., bbd9  84 23     sty $23
., bbdb  a0 04     ldy #$04
., bbdd  a5 65     lda $65
., bbdf  91 22     sta ($22),y      ;store FACHO4 - fac#1
                                    mantissa
., bbe1  88        dey
., bbe2  a5 64     lda $64
., bbe4  91 22     sta ($22),y      ;store FACHO3
., bbe6  88        dey
., bbe7  a5 63     lda $63
., bbe9  91 22     sta ($22),y      ;store FACHO2
., bbeb  88        dey
., bbec  a5 66     lda $66          ;FACSGN - fac#1 sign
., bbee  09 7f     ora #$7f
., bbf0  25 62     and $62          ;pack sign onto FACHO1
., bbf2  91 22     sta ($22),y      ;store FACHO1 & sign
., bbf4  88        dey
., bbf5  a5 61     lda $61
., bbf7  91 22     sta ($22),y      ;store FACEXP - fac#1
                                    exponent
., bbf9  84 70     sty $70          ;FACOV - fac#1 rounding
., bbfb  60        rts
```

48124 MOVFA: COPY FAC#2 INTO FAC#1

This routine copies the value of fac#2 into fac#1 and sets FACOV = 0. This causes a little information to be lost by rounding.

```
., bbfc  a5 6e     lda $6e          ;ARGSGN - fac#2 sign
., bbfe  85 66     sta $66          ;FACSGN - fac#1 sign
., bc00  a2 05     ldx #$05
., bc02  b5 68     lda $68,x        ;copy exponent and mantissa
., bc04  95 60     sta $60,x        ;from fac#2 to fac#1
., bc06  ca        dex
., bc07  d0 f9     bne $bc02
., bc09  86 70     stx $70          ;FACOV - fac#1 rounding
., bc0b  60        rts
```

48140 MOVAF: COPY FAC#1 INTO FAC#2

FAC#1 is rounded by the following routine and then copied into fac#2. FACOV is set to zero, thereby causing some data to be lost.

```
., bc0c  20 1b bc jsr $bc1b    ;round fac#1
., bc0f  a2 06    ldx #$06
., bc11  b5 60    lda $60,x     ;copy fac#1 to fac#2
., bc13  95 68    sta $68,x
., bc15  ca       dex
., bc16  d0 f9    bne $bc11
., bc18  86 70    stx $70       ;FACOV - fac#1 rounding
., bc1a  60       rts
```

48155 ROUND: ROUND FAC#1

This routine exits if fac#1 is zero (ie. FACEXP = 0) or FACOV < #7F. A single bit is added to fac#1. If this is rippled throughout the accumulator then the exponent is incremented and the mantissa is rotated right. If this occurs then the rounding bit is lost.

```
., bc1b  a5 61    lda $61       ;FACEXP - fac#1 exponent
., bc1d  f0 fb    beq $bc1a     ;fac#1 is zero - RTS
., bc1f  06 70    asl $70       ;FACOV - fac#1 rounding byte
., bc21  90 f7    bcc $bc1a     ;<#7F - RTS
., bc23  20 6f b9 jsr $b96f     ;add a single bit to fac#1
., bc26  d0 f2    bne $bc1a
., bc28  4c 38 b9 jmp $b938     ;increment FACEXP and rotate
                                FACHO
```

48171 SIGN: CHECK SIGN OF FAC#1

FACEXP and FACSGN are examined to determine the sign of fac#1. The result is placed in (A) as follows:

```
#00 = fac#1 zero
#01 = fac#1 positive
#FF = fac#1 negative
```

```
., bc2b  a5 61    lda $61
., bc2d  f0 09    beq $bc38
., bc2f  a5 66    lda $66
., bc31  2a       rol
., bc32  a9 ff    lda #$ff
., bc34  b0 02    bcs $bc38
., bc36  a9 01    lda #$01
., bc38  60       rts
```

48185 SGN: PERFORM SGN

The previous routine is called to compute the sign of fac#1. This is held in (A), thus there is an entry point to store

the contents of (A) in fac#1. The exponent is set to #88 to set the size of the accumulator for normalisation later. At this point, if an entry is made with X=#90 and ($62) holding an integer value, this integer can be converted to flpt. Carry is used to indicate the sign - set to indicate positive, and clear to indicate negative. The number is set into flpt and normalised via the addition routines.

```
., bc39  20 2b bc jsr $bc2b    ;check sign of fac#1
., bc3c  85 62    sta $62      ;FACHO1 - entry to put (A) in
                               flpt
., bc3e  a9 00    lda #$00
., bc40  85 63    sta $63      ;FACHO2 - fac#1 mantissa
., bc42  a2 88    ldx #$88     ;exponent size for
                               normalisation
., bc44  a5 62    lda $62      ;FACHO1
., bc46  49 ff    eor #$ff
., bc48  2a       rol
., bc49  a9 00    lda #$00
., bc4b  85 65    sta $65      ;FACHO4
., bc4d  85 64    sta $64      ;FACHO3
., bc4f  86 61    stx $61      ;FACEXP - fac#1 exponent
., bc51  85 70    sta $70      ;FACOV - fac#1 rounding byte
., bc53  85 66    sta $66      ;FACSGN - fac#1 sign
., bc55  4c d2 b8 jmp $b8d2    ;put number in flpt and
                               normalise
```

48216 ABS: PERFORM ABS

A zero is put into bit 7 of FACSGN, thus making fac#1 a positive value.

```
., bc58  46 66    lsr $66      ;FACSGN - fac#1 sign
., bc5a  60       rts
```

48219 FCOMP: COMPARE FAC#1 WITH MEMORY

On entry, (A/Y) must point to a 5 byte flpt number. This number is then compared byte by byte with that in fac#1. The result is placed in (A) on exit and holds one of the following values:

```
#00 = both values are equal
#01 = fac#1 > memory flpt
#FF = fac#1 < memory flpt
```

```
., bc5b  85 24    sta $24      ;INDEX2 - points to flpt
                               number
., bc5d  84 25    sty $25
```

```
.,  bc5f   a0 00    ldy #$00
.,  bc61   b1 24    lda ($24),y
.,  bc63   c8       iny
.,  bc64   aa       tax
.,  bc65   f0 c4    beq $bc2b      ;check sign of fac#1
.,  bc67   b1 24    lda ($24),y
.,  bc69   45 66    eor $66        ;FACSGN - fac#1 sign
.,  bc6b   30 c2    bmi $bc2f      ;signs are different - set
                                   (A)
.,  bc6d   e4 61    cpx $61        ;FACEXP - fac#1 exponent
.,  bc6f   d0 21    bne $bc92
.,  bc71   b1 24    lda ($24),y    ;compare mantissas
.,  bc73   09 80    ora #$80
.,  bc75   c5 62    cmp $62
.,  bc77   d0 19    bne $bc92
.,  bc79   c8       iny
.,  bc7a   b1 24    lda ($24),y
.,  bc7c   c5 63    cmp $63
.,  bc7e   d0 12    bne $bc92
.,  bc80   c8       iny
.,  bc81   b1 24    lda ($24),y
.,  bc83   c5 64    cmp $64
.,  bc85   d0 0b    bne $bc92
.,  bc87   c8       iny
.,  bc88   a9 7f    lda #$7f
.,  bc8a   c5 70    cmp $70
.,  bc8c   b1 24    lda ($24),y
.,  bc8e   e5 65    sbc $65
.,  bc90   f0 28    beq $bcba
.,  bc92   a5 66    lda $66        ;FACSGN
.,  bc94   90 02    bcc $bc98
.,  bc96   49 ff    eor #$ff
.,  bc98   4c 31 bc jmp $bc31      ;set (A) according to sign
```

48283 QINT: CONVERT FAC#1 TO INTEGER

The number in fac#1 is converted into a 4 byte signed integer within the mantissa of fac#1. ie. $62 - $65. This is in the hi - lo format. ie. $62 is the hi byte and $65 the lo byte.

```
.,  bc9b   a5 61    lda $61        ;FACEXP - fac#1 exponent
.,  bc9d   f0 4a    beq $bce9      ;zero - write zero value in
                                   fac#1
.,  bc9f   38       sec
.,  bca0   e9 a0    sbc #$a0
.,  bca2   24 66    bit $66        ;FACSGN - fac#1 sign
.,  bca4   10 09    bpl $bcaf
.,  bca6   aa       tax
```

```
., bca7  a9 ff     lda #$ff
., bca9  85 68     sta $68      ;BITS - fac#1 overflow digit
., bcab  20 4d b9  jsr $b94d    ;2's complement fac#1
., bcae  8a        txa
., bcaf  a2 61     ldx #$61     ;pointer to fac#1
., bcb1  c9 f9     cmp #$f9
., bcb3  10 06     bpl $bcbb
., bcb5  20 99 b9  jsr $b999    ;rotate accumulator
., bcb8  84 68     sty $68      ;BITS
., bcba  60        rts
., bcbb  a8        tay
., bcbc  a5 66     lda $66      ;FACSGN
., bcbe  29 80     and #$80
., bcc0  46 62     lsr $62      ;FACHO1-put sgn in mantissa
., bcc2  05 62     ora $62
., bcc4  85 62     sta $62
., bcc6  20 b0 b9  jsr $b9b0    ;rotate FACHO2 - FACHO4 right
., bcc9  84 68     sty $68      ;BITS
., bccb  60        rts
```

48332 INT: PERFORM INT

This routine converts fac#1 into the nearest integer value (rounding down). However instead of leaving the result in integer form, it is converted back into flpt and normalised by part of the addition routine. The final part is used by the previous routine to set fac#1 to zero when FACEXP was found to be zero.

```
., bccc  a5 61     lda $61      ;FACEXP - fac#1 exponent
., bcce  c9 a0     cmp #$a0     ;already integer value?
., bcd0  b0 20     bcs $bcf2    ;yes - RTS
., bcd2  20 9b bc  jsr $bc9b    ;convert fac#1 to integer
., bcd5  84 70     sty $70      ;FACOV - fac#1 rounding
., bcd7  a5 66     lda $66      ;FACSGN - fac#1 sign
., bcd9  84 66     sty $66
., bcdb  49 80     eor #$80
., bcdd  2a        rol
., bcde  a9 a0     lda #$a0
., bce0  85 61     sta $61      ;FACEXP
., bce2  a5 65     lda $65      ;FACHO4 - fac#1 mantissa
., bce4  85 07     sta $07      ;CHARAC - search character
., bce6  4c d2 b8  jmp $b8d2    ;set to flpt and normalise
., bce9  85 62     sta $62      ;FACHO1
., bceb  85 63     sta $63      ;FACHO2
., bced  85 64     sta $64      ;FACHO3
., bcef  85 65     sta $65      ;FACHO4
., bcf1  a8        tay
., bcf2  60        rts
```

48371 FIN: CONVERT ASCII STRING TO A NUMBER IN FAC#1

This routine evaluates a number which is in ASCII string form, and places it in fac#1. The characters -, +, ., E etc., are scanned for and dealt with. As each digit is encountered, the accumulator is multiplied by 10 and the digit, now placed in (A) is added to it. Thus the entry point $BD7E can be used to add the contents of (A) to fac#1. Note: for the main routine, TXTPTR must point to the start of the ASCII string to be scanned.

```
., bcf3  a0 00     ldy #$00
., bcf5  a2 0a     ldx #$0a      ;counter
., bcf7  94 5d     sty $5d,x     ;zero fac#1 + temps
., bcf9  ca        dex
., bcfa  10 fb     bpl $bcf7
., bcfc  90 0f     bcc $bd0d
., bcfe  c9 2d     cmp #$2d      ;ASCII minus?
., bd00  d0 04     bne $bd06
., bd02  86 67     stx $67       ;SCNFLG
., bd04  f0 04     beq $bd0a
., bd06  c9 2b     cmp #$2b      ;ASCII +?
., bd08  d0 05     bne $bd0f
., bd0a  20 73 00  jsr $0073     ;CHRGET
., bd0d  90 5b     bcc $bd6a
., bd0f  c9 2e     cmp #$2e      ;ASCII decimal?
., bd11  f0 2e     beq $bd41
., bd13  c9 45     cmp #$45      ;ASCII E?
., bd15  d0 30     bne $bd47
., bd17  20 73 00  jsr $0073     ;CHRGET
., bd1a  90 17     bcc $bd33
., bd1c  c9 ab     cmp #$ab      ;token NOT?
., bd1e  f0 0e     beq $bd2e
., bd20  c9 2d     cmp #$2d      ;ASCII minus?
., bd22  f0 0a     beq $bd2e
., bd24  c9 aa     cmp #$aa      ;token +?
., bd26  f0 08     beq $bd30
., bd28  c9 2b     cmp #$2b      ;ASCII +?
., bd2a  f0 04     beq $bd30
., bd2c  d0 07     bne $bd35
., bd2e  66 60     ror $60
., bd30  20 73 00  jsr $0073     ;CHRGET
., bd33  90 5c     bcc $bd91
., bd35  24 60     bit $60
., bd37  10 0e     bpl $bd47
., bd39  a9 00     lda #$00
., bd3b  38        sec
., bd3c  e5 5e     sbc $5e
```

```
., bd3e  4c 49 bd jmp $bd49
., bd41  66 5f    ror $5f
., bd43  24 5f    bit $5f
., bd45  50 c3    bvc $bd0a
., bd47  a5 5e    lda $5e
., bd49  38       sec
., bd4a  e5 5d    sbc $5d
., bd4c  85 5e    sta $5e
., bd4e  f0 12    beq $bd62
., bd50  10 09    bpl $bd5b
., bd52  20 fe ba jsr $bafe      ;divide fac#1 by 10
., bd55  e6 5e    inc $5e
., bd57  d0 f9    bne $bd52
., bd59  f0 07    beq $bd62
., bd5b  20 e2 ba jsr $bae2      ;multiply fac#1 by 10
., bd5e  c6 5e    dec $5e
., bd60  d0 f9    bne $bd5b
., bd62  a5 67    lda $67
., bd64  30 01    bmi $bd67
., bd66  60       rts
., bd67  4c b4 bf jmp $bfb4      ;negate fac#1
., bd6a  48       pha            ;-- add digit to mantissa
., bd6b  24 5f    bit $5f
., bd6d  10 02    bpl $bd71
., bd6f  e6 5d    inc $5d
., bd71  20 e2 ba jsr $bae2      ;multiply fac#1 by 10
., bd74  68       pla
., bd75  38       sec
., bd76  e9 30    sbc #$30
., bd78  20 7e bd jsr $bd7e      ;add contents of A to fac#1
., bd7b  4c 0a bd jmp $bd0a
., bd7e  48       pha
., bd7f  20 0c bc jsr $bc0c      ;copy fac#1 to fac#2
., bd82  68       pla
., bd83  20 3c bc jsr $bc3c      ;store A in fac#1
., bd86  a5 6e    lda $6e        ;ARGSGN - fac#2 sign
., bd88  45 66    eor $66        ;FACSGN - fac#1 sign
., bd8a  85 6f    sta $6f        ;ARISGN - sign comparison
                                 result
., bd8c  a6 61    ldx $61        ;FACEXP - fac#1 exponent
., bd8e  4c 6a b8 jmp $b86a      ;add fac#2 to fac#1
., bd91  a5 5e    lda $5e
., bd93  c9 0a    cmp #$0a
., bd95  90 09    bcc $bda0
., bd97  a9 64    lda #$64
., bd99  24 60    bit $60
., bd9b  30 11    bmi $bdae
., bd9d  4c 7e b9 jmp $b97e      ;?OVERFLOW error
., bda0  0a       asl
```

```
., bda1  0a        asl
., bda2  18        clc
., bda3  65 5e     adc $5e
., bda5  0a        asl
., bda6  18        clc
., bda7  a0 00     ldy #$00
., bda9  71 7a     adc ($7a),y
., bdab  38        sec
., bdac  e9 30     sbc #$30
., bdae  85 5e     sta $5e
., bdb0  4c 30 bd jmp $bd30
```

48563 N0999: STRING CONVERSION CONSTANTS

There are three 5-byte constants in this table: 99999999.9, 999999999 and 1000000000.

```
.:bdb3 9b 3e bc 1f fd 9e 6e 6b
.:bdbb 27 fd 9e 6e 6b 28 00
```

48578 INPRT: OUTPUT 'IN' AND LINE NUMBER

Pointers are set up to the message 'IN', which is then output. CURLIN is placed in ($62) and converted from an integer to flpt and then to a string. This string is finally output. There are several possible entry points to this routine, for example, to output (A/X), or to output fac#1.

```
., bdc2  a9 71     lda #$71
., bdc4  a0 a3     ldy #$a3     ;$A371 points to message 'IN'
., bdc6  20 da bd jsr $bdda     ;output string
., bdc9  a5 3a     lda $3a      ;CURLIN - current line number
., bdcb  a6 39     ldx $39
., bdcd  85 62     sta $62      ;FACHO1 - fac#1 mantissa
., bdcf  86 63     stx $63      ;FACHO2
., bdd1  a2 90     ldx #$90
., bdd3  38        sec
., bdd4  20 49 bc jsr $bc49     ;convert integer to flpt
., bdd7  20 df bd jsr $bddf     ;convert fac#1 to ASCII
                                string
., bdda  4c 1e ab jmp $ab1e     ;output string
```

48605 FOUT: CONVERT FAC#1 TO ASCII STRING

This routine converts the flpt number in fac#1 to an ASCII string starting at $0100. This is a reserved area of memory at the low end of the stack. The string is terminated with a zero byte. The two tables of constants at $BDB3 and $BF11

are used in the conversion process, the three constants at $BDB3 being used to determine the use of scientific notation. Note that the contents of fac#1 are destroyed in the process. On exit, (A/Y) points to the start of the string.

```
., bddd  a0 01     ldy #$01
., bddf  a9 20     lda #$20       ;ASCII space
., bde1  24 66     bit $66        ;FACSGN - fac#1 sign
., bde3  10 02     bpl $bde7
., bde5  a9 2d     lda #$2d       ;ASCII minus
., bde7  99 ff 00  sta $00ff,y    ;store minus or space in
                                  buffer
., bdea  85 66     sta $66        ;FACSGN
., bdec  84 71     sty $71        ;FBUFPT
., bdee  c8        iny
., bdef  a9 30     lda #$30       ;ASCII 0
., bdf1  a6 61     ldx $61        ;FACEXP - fac#1 exponent
., bdf3  d0 03     bne $bdf8      ;not zero
., bdf5  4c 04 bf  jmp $bf04      ;store zero and end
., bdf8  a9 00     lda #$00
., bdfa  e0 80     cpx #$80
., bdfc  f0 02     beq $be00
., bdfe  b0 09     bcs $be09
., be00  a9 bd     lda #$bd
., be02  a0 bd     ldy #$bd       ;$BDBD=conversion constant 3
., be04  20 28 ba  jsr $ba28      ;multiply fac#1 by flpt at
                                  (A/Y)
., be07  a9 f7     lda #$f7
., be09  85 5d     sta $5d
., be0b  a9 b8     lda #$b8
., be0d  a0 bd     ldy #$bd       ;$BDB8=conversion constant 2
., be0f  20 5b bc  jsr $bc5b      ;compare fac#1 with flpt at
                                  (A/Y)
., be12  f0 1e     beq $be32
., be14  10 12     bpl $be28
., be16  a9 b3     lda #$b3
., be18  a0 bd     ldy #$bd       ;$BDB3=conversion constant 1
., be1a  20 5b bc  jsr $bc5b      ;compare fac#1 with flpt at
                                  (A/Y)
., be1d  f0 02     beq $be21
., be1f  10 0e     bpl $be2f
., be21  20 e2 ba  jsr $bae2      ;multiply fac#1 by 10
., be24  c6 5d     dec $5d
., be26  d0 ee     bne $be16      ;compare with constant 1
., be28  20 fe ba  jsr $bafe      ;divide fac#1 by 10
., be2b  e6 5d     inc $5d
., be2d  d0 dc     bne $be0b      ;compare with constant 2
., be2f  20 49 b8  jsr $b849      ;add 0.5 to fac#1
```

```
., be32  20 9b bc jsr $bc9b      ;convert fac#1 to integer in
                                  fac#1
., be35  a2 01    ldx #$01
., be37  a5 5d    lda $5d
., be39  18       clc
., be3a  69 0a    adc #$0a
., be3c  30 09    bmi $be47
., be3e  c9 0b    cmp #$0b
., be40  b0 06    bcs $be48
., be42  69 ff    adc #$ff
., be44  aa       tax
., be45  a9 02    lda #$02
., be47  38       sec
., be48  e9 02    sbc #$02
., be4a  85 5e    sta $5e
., be4c  86 5d    stx $5d
., be4e  8a       txa
., be4f  f0 02    beq $be53
., be51  10 13    bpl $be66
., be53  a4 71    ldy $71         ;FBUFPT
., be55  a9 2e    lda #$2e        ;ASCII decimal
., be57  c8       iny
., be58  99 ff 00 sta $00ff,y     ;store character in buffer
., be5b  8a       txa
., be5c  f0 06    beq $be64
., be5e  a9 30    lda #$30        ;ASCII 0
., be60  c8       iny
., be61  99 ff 00 sta $00ff,y     ;store character in buffer
., be64  84 71    sty $71         ;FBUFPT
., be66  a0 00    ldy #$00
```

48744 FOUTIM: CONVERT TI TO STRING

This routine converts the three byte value of the real-time clock, TI into an ASCII string starting at $0100. In the process, the table of constants at $BF11 is used. On exit, (A/Y) points to the start of the string.

```
., be68  a2 80    ldx #$80
., be6a  a5 65    lda $65         ;FACHO4 - fac#1 mantissa
., be6c  18       clc
., be6d  79 19 bf adc $bf19,y
., be70  85 65    sta $65
., be72  a5 64    lda $64         ;FACHO3
., be74  79 18 bf adc $bf18,y
., be77  85 64    sta $64
., be79  a5 63    lda $63         ;FACHO2
., be7b  79 17 bf adc $bf17,y
., be7e  85 63    sta $63
```

```
., be80  a5 62     lda $62      ;FACHO1
., be82  79 16 bf  adc $bf16,y
., be85  85 62     sta $62
., be87  e8        inx
., be88  b0 04     bcs $be8e
., be8a  10 de     bpl $be6a
., be8c  30 02     bmi $be90
., be8e  30 da     bmi $be6a
., be90  8a        txa
., be91  90 04     bcc $be97
., be93  49 ff     eor #$ff
., be95  69 0a     adc #$0a
., be97  69 2f     adc #$2f
., be99  c8        iny
., be9a  c8        iny
., be9b  c8        iny
., be9c  c8        iny
., be9d  84 47     sty $47      ;VARPNT - pointer to variable
                                data
., be9f  a4 71     ldy $71      ;FBUFPT
., bea1  c8        iny
., bea2  aa        tax
., bea3  29 7f     and #$7f
., bea5  99 ff 00  sta $00ff,y  ;store string character in
                                buffer
., bea8  c6 5d     dec $5d
., beaa  d0 06     bne $beb2
., beac  a9 2e     lda #$2e     ;ASCII decimal point
., beae  c8        iny
., beaf  99 ff 00  sta $00ff,y  ;store string character in
                                buffer
., beb2  84 71     sty $71      ;FBUFPT
., beb4  a4 47     ldy $47      ;VARPNT
., beb6  8a        txa
., beb7  49 ff     eor #$ff
., beb9  29 80     and #$80
., bebb  aa        tax
., bebc  c0 24     cpy #$24
., bebe  f0 04     beq $bec4
., bec0  c0 3c     cpy #$3c
., bec2  d0 a6     bne $be6a
., bec4  a4 71     ldy $71      ;FBUFPT
., bec6  b9 ff 00  lda $00ff,y  ;get string character from
                                buffer
., bec9  88        dey
., beca  c9 30     cmp #$30     ;ASCII 0?
., becc  f0 f8     beq $bec6    ;yes - get previous character
., bece  c9 2e     cmp #$2e     ;ASCII decimal?
., bed0  f0 01     beq $bed3    ;yes
```

```
., bed2  c8        iny
., bed3  a9 2b     lda #$2b      ;ASCII +
., bed5  a6 5e     ldx $5e
., bed7  f0 2e     beq $bf07
., bed9  10 08     bpl $bee3
., bedb  a9 00     lda #$00
., bedd  38        sec
., bede  e5 5e     sbc $5e
., bee0  aa        tax
., bee1  a9 2d     lda #$2d      ;ASCII minus
., bee3  99 01 01  sta $0101,y   ;store character in buffer
., bee6  a9 45     lda #$45      ;<VARNAM
., bee8  99 00 01  sta $0100,y   ;store character in buffer
., beeb  8a        txa
., beec  a2 2f     ldx #$2f
., beee  38        sec
., beef  e8        inx
., bef0  e9 0a     sbc #$0a
., bef2  b0 fb     bcs $beef
., bef4  69 3a     adc #$3a
., bef6  99 03 01  sta $0103,y   ;store character in buffer
., bef9  8a        txa
., befa  99 02 01  sta $0102,y   ;store character in buffer
., befd  a9 00     lda #$00
., beff  99 04 01  sta $0104,y   ;store zero terminator
., bf02  f0 08     beq $bf0c     ;set pointer to start of
                                 string
., bf04  99 ff 00  sta $00ff,y
., bf07  a9 00     lda #$00
., bf09  99 00 01  sta $0100,y   ;store zero terminator
., bf0c  a9 00     lda #$00
., bf0e  a0 01     ldy #$01      ;$0100 is the start of the
                                 string
., bf10  60        rts
```

48913 FHALF: TABLE OF CONSTANTS

These are constants for string conversion, TI$, SQR and rounding. They include 0.5, 0 (zero) in 5-byte flpt, plus 4-byte powers of 10 and constants for 1 hour, 1 minute and 1 second.

```
.:bf11 80 00 00 00 00 fa 0a 1f
.:bf19 00 00 98 96 80 ff f0 bd
.:bf21 c0 00 01 86 a0 ff ff d8
.:bf29 f0 00 00 03 e8 ff ff ff
.:bf31 9c 00 00 00 0a ff ff ff
.:bf39 ff ff df 0a 80 00 03 4b
.:bf41 c0 ff ff 73 60 00 00 0e
```

```
.:bf49 10 ff ff fd a8 00 00 00
.:bf51 3c ec aa aa aa aa aa aa
.:bf59 aa aa aa aa aa aa aa aa
.:bf61 aa aa aa aa aa aa aa aa
.:bf69 aa aa aa aa aa aa aa aa
```

49009 SQR: PERFORM SQR

Fac#1 is copied into fac#2 and then loaded with the flpt value 0.5 from the above table, before control is dropped through to PERFORM POWER.

```
., bf71  20 0c bc jsr $bc0c    ;copy fac#1 to fac#2
., bf74  a9 11    lda #$11
., bf76  a0 bf    ldy #$bf     ;$BF11 = flpt constant 0.5
., bf78  20 a2 bb jsr $bba2    ;load fac#1 with flpt at
                               (A/Y)
```

49019 FPWRT: PERFORM POWER ~

fac#1 = fac#2 ~ fac#1. On entry at $BF7B, (A) must hold FACEXP. Alternately, by entering at $BF78 in the previous routine, fac#1 can be loaded from (A/Y). If fac#1 is found to be 0, then it is set to 1 and the routine exits. Similarly, if fac#2 is found to be 0, then fac#1 is set to zero. The power is obtained by first saving fac#1 in memory at $004E onwards, taking the LOG of fac#2, multiplying it by the value of fac#1 previously saved, then taking the exponent of the result.

```
., bf7b  f0 70    beq $bfed    ;do EXP ie set fac#1 = 1
., bf7d  a5 69    lda $69      ;ARGEXP - fac#2 exponent
., bf7f  d0 03    bne $bf84
., bf81  4c f9 b8 jmp $b8f9    ;add fac#2 to fac#1
., bf84  a2 4e    ldx #$4e
., bf86  a0 00    ldy #$00     ;set (X/Y) to $004E
., bf88  20 d4 bb jsr $bbd4    ;store fac#1 at (X/Y)
., bf8b  a5 6e    lda $6e      ;ARGSGN - fac#2 sign
., bf8d  10 0f    bpl $bf9e
., bf8f  20 cc bc jsr $bccc    ;do INT
., bf92  a9 4e    lda #$4e
., bf94  a0 00    ldy #$00     ;set (A/Y) to $004E
., bf96  20 5b bc jsr $bc5b    ;compare fac#1 with flpt at
                               (A/Y)
., bf99  d0 03    bne $bf9e
., bf9b  98       tya
., bf9c  a4 07    ldy $07      ;CHARAC
., bf9e  20 fe bb jsr $bbfe    ;copy fac#2 into fac#1
., bfa1  98       tya
```

```
., bfa2  48        pha
., bfa3  20 ea b9  jsr $b9ea     ;do LOG
., bfa6  a9 4e     lda #$4e
., bfa8  a0 00     ldy #$00      ;set (A/Y) to $004E
., bfaa  20 28 ba  jsr $ba28     ;multiply fac#1 by flpt at
                                 (A/Y)
., bfad  20 ed bf  jsr $bfed     ;do EXP - e ^ fac#1
., bfb0  68        pla
., bfb1  4a        lsr
., bfb2  90 0a     bcc $bfbe     ;RTS
```

49076 NEGOP: NEGATE FAC#1

Fac#1 is first checked to see if it is zero. If not, then the sign byte is reversed, so that if it was 0 it is now #FF, and if #FF, now 0.

```
., bfb4  a5 61     lda $61       ;FACEXP - fac#1 exponent
., bfb6  f0 06     beq $bfbe     ;fac#1 is zero - RTS
., bfb8  a5 66     lda $66       ;FACSGN - fac#1 sign
., bfba  49 ff     eor #$ff      ;reverse sign byte
., bfbc  85 66     sta $66
., bfbe  60        rts
```

49087 LOGEB2: TABLE OF CONSTANTS

The table holds the following 5 byte flpt constants:
$BFBF = 1.44269504 (1/LOG to base 2 e)
$BFC4 = #07 (1 byte counter for EXP series)

```
                                  $BFC5 = 2.149876 E-5
                                  (constant 1)
                                  $BFCA = 1.435231 E-4
                                  (constant 2)
.:bfbf  81  38  aa  3b 29 07 71 34  $BFCF  =  1.342263   E-3
                                  (constant 3)
.:bfc7  58  3e  56  74 16 7e b3 1b  $BFD4  =  9.6414017  E-3
                                  (constant 4)
.:bfcf  77  2f  ee  e3 85 7a 1d 84  $BFD9  =  5.550513   E-2
                                  (constant 5)
.:bfd7  1c  2a  7c  63 59 58 0a 7e  $BFDE  =  2.402263   E-4
                                  (constant 6)
.:bfdf  75  fd  e7  c6 80 31 72 18  $BFE3  =  6.931471   E-1
                                  (constant 7)
.:bfe7 10 81 00 00 00 00           $BFE8 = 1
```

49133 EXP: PERFORM EXP

fac#1 = e ^ fac#1. This routine is unique in that it spans over two roms - the start is in the BASIC rom and the finish

in the KERNAL rom.

```
., bfed  a9 bf     lda #$bf
., bfef  a0 bf     ldy #$bf       ;$BFBF points to constant for
                                  EXP
., bff1  20 28 ba jsr $ba28       ;multiply fac#1 by flpt at
                                  (A/Y)
., bff4  a5 70     lda $70        ;FACOV - fac#1 rounding
., bff6  69 50     adc #$50
., bff8  90 03     bcc $bffd
., bffa  20 23 bc jsr $bc23       ;add a bit to fac#1
., bffd  4c 00 e0 jmp $e000       ;continue in KERNAL rom
., e000  85 56     sta $56
., e002  20 0f bc jsr $bc0f       ;copy fac#1 to fac#2
., e005  a5 61     lda $61        ;FACEXP - fac#1 exponent
., e007  c9 88     cmp #$88
., e009  90 03     bcc $e00e
., e00b  20 d4 ba jsr $bad4       ;do overflow/underflow
., e00e  20 cc bc jsr $bccc       ;do INT
., e011  a5 07     lda $07        ;CHARAC
., e013  18        clc
., e014  69 81     adc #$81
., e016  f0 f3     beq $e00b
., e018  38        sec
., e019  e9 01     sbc #$01
., e01b  48        pha
., e01c  a2 05     ldx #$05       ;set loop counter
., e01e  b5 69     lda $69,x      ;swap over accumulators
., e020  b4 61     ldy $61,x
., e022  95 61     sta $61,x      ;store fac#2 in fac#1
., e024  94 69     sty $69,x      ;store fac#1 in fac#2
., e026  ca        dex
., e027  10 f5     bpl $e01e      :do loop
., e029  a5 56     lda $56
., e02b  85 70     sta $70        ;FACOV
., e02d  20 53 b8 jsr $b853       ;subtract fac#1 from fac#2
., e030  20 b4 bf jsr $bfb4       ;negate fac#1
., e033  a9 c4     lda #$c4
., e035  a0 bf     ldy #$bf       ;$BFC4 - counter for EXP
                                  series
., e037  20 59 e0 jsr $e059       ;evaluate series function
., e03a  a9 00     lda #$00
., e03c  85 6f     sta $6f        ;ARISGN - sign comparison
                                  result
., e03e  68        pla
., e03f  20 b9 ba jsr $bab9       ;test both accumulators
., e042  60        rts
```

57411 POLYX: SERIES EVALUATION

This routine evaluates all of the mathematical functions, eg

LOG, SIN etc. On entry, (A/Y) points to the number of constants in the series. After the argument has been set to a suitable range, the series is evaluated through the range of constants, both by adding and multiplying. The result is finally modified to give the correct exponent, sign etc.

```
., e043  85 71     sta $71       ;FBUFPT
., e045  84 72     sty $72
., e047  20 ca bb  jsr $bbca     ;store fac#1 at $0057 to
                                 $005b
., e04a  a9 57     lda #$57
., e04c  20 28 ba  jsr $ba28     ;multiply fac#1 by flpt at
                                 $0057
., e04f  20 5d e0  jsr $e05d     ;evaluate series
., e052  a9 57     lda #$57
., e054  a0 00     ldy #$00      ;set (A/Y) = $0057
., e056  4c 28 ba  jmp $ba28     ;multiply fac#1 by flpt at
                                 (A/Y)
., e059  85 71     sta $71
., e05b  84 72     sty $72       ;(A/Y) points to # constants
                                 in series
., e05d  20 c7 bb  jsr $bbc7     ;store fac#1 at $005c - $0060
., e060  b1 71     lda ($71),y   ;get number of constants
., e062  85 67     sta $67       ;SCNFLG - pointer for series
                                 eval const
., e064  a4 71     ldy $71       ;point to first constant
., e066  c8        iny
., e067  98        tya
., e068  d0 02     bne $e06c
., e06a  e6 72     inc $72
., e06c  85 71     sta $71
., e06e  a4 72     ldy $72
., e070  20 28 ba  jsr $ba28     ;multiply fac#1 by series
                   constant
., e073  a5 71     lda $71
., e075  a4 72     ldy $72
., e077  18        clc
., e078  69 05     adc #$05      ;point to next constant
., e07a  90 01     bcc $e07d
., e07c  c8        iny
., e07d  85 71     sta $71       ;store pointer in FBUFPT
., e07f  84 72     sty $72
., e081  20 67 b8  jsr $b867     ;add constant to fac31
., e084  a9 5c     lda #$5c
., e086  a0 00     ldy #$00
., e088  c6 67     dec $67       ;SCNFLG
., e08a  d0 e4     bne $e070     ;process next constant in
```

```
                                  series
., e08c  60         rts
```

57485 RMULC: CONSTANTS FOR RND

Two flpt constants for performing RND. The first is 11879546.4 (multiplicative) and the second is 3.92767778E-8 (additive)

```
.:e08d 98 35 44 7a 00 68 28 b1
.:e095 46 00
```

57495 RND: PERFORM RND

The sign of fac#1 is tested for the processing of the three possible types of argument. If zero, then fac#1 is loaded from the CIA#1 timer A and TOD clock. Since these values will change at every clock cycle, this argument can be considered to produce a truly random result. However, the TOD clock must be started by the programmer, since it is never started by the operating system. Also, the clock counts in BCD rather than true binary. If the sign is positive, then the random number seed in zero page is loaded into fac#1, and the first RND constant is multiplied and the second added to it. Control is then passed on to process the argument in the third section. If the sign is negative, the 4 bytes of FACHO are swapped over; 1 for 4 and 2 for 3. Finally, all three methods have a common exit section. This makes fac#1 positive, and ensures that an exact zero result does not occur. The final result is then stored in zero page as the seed for the next random number.

```
., e097  20 2b bc jsr $bc2b    ;check sign of fac#1
., e09a  30 37    bmi $e0d3    ;negative
., e09c  d0 20    bne $e0be    ;positive
., e09e  20 f3 ff jsr $fff3    ;IOBASE - read I/O base
                               address
., e0a1  86 22    stx $22      ;INDEX1 - holds I/O base
                               address
., e0a3  84 23    sty $23
., e0a5  a0 04    ldy #$04
., e0a7  b1 22    lda ($22),y  ;read CIA#1 timer A lo
., e0a9  85 62    sta $62      ;FACHO1 - fac#1 mantissa
., e0ab  c8       iny
., e0ac  b1 22    lda ($22),y  ;read CIA#1 timer A hi
., e0ae  85 64    sta $64      ;FACHO3
., e0b0  a0 08    ldy #$08
., e0b2  b1 22    lda ($22),y  ;read CIA#1 TOD 10ths sec
                               register
```

```
., e0b4  85 63     sta $63       ;FACHO2
., e0b6  c8        iny
., e0b7  b1 22     lda ($22),y   ;read CIA#1 TOD secs register
., e0b9  85 65     sta $65       ;FACHO4
., e0bb  4c e3 e0  jmp $e0e3     ;do common exit section
., e0be  a9 8b     lda #$8b
., e0c0  a0 00     ldy #$00      ;(A/Y) points to RNDX - seed
                                 value
., e0c2  20 a2 bb  jsr $bba2     ;load fac#1 with flpt at
                                 (A/Y)
., e0c5  a9 8d     lda #$8d
., e0c7  a0 e0     ldy #$e0      ;(A/Y) points to 1st RND
                                 constant
., e0c9  20 28 ba  jsr $ba28     ;multiply flpt at (A/Y) by
                                 fac#1
., e0cc  a9 92     lda #$92
., e0ce  a0 e0     ldy #$e0      ;(A/Y) points to 2nd RND
                                 constant
., e0d0  20 67 b8  jsr $b867     ;add flpt at (A/Y) to fac#1
., e0d3  a6 65     ldx $65       ;swap FACHO1 and FACHO4
., e0d5  a5 62     lda $62
., e0d7  85 65     sta $65
., e0d9  86 62     stx $62
., e0db  a6 63     ldx $63       ;swap FACHO2 and FACHO3
., e0dd  a5 64     lda $64
., e0df  85 63     sta $63
., e0e1  86 64     stx $64
., e0e3  a9 00     lda #$00
., e0e5  85 66     sta $66       ;FACSGN - set fac#1 positive
., e0e7  a5 61     lda $61       ;FACEXP - fac#1 exponent
., e0e9  85 70     sta $70       ;FACOV - fac#1 rounding
., e0eb  a9 80     lda #$80
., e0ed  85 61     sta $61       ;FACEXP
., e0ef  20 d7 b8  jsr $b8d7     ;adjust fac#1
., e0f2  a2 8b     ldx #$8b
., e0f4  a0 00     ldy #$00      ;(A/Y) points to RNDX - seed
                                 value
., e0f6  4c d4 bb  jmp $bbd4     ;store fac#1 at (A/Y)
```

57593 BIOERR: HANDLE I/O ERROR IN BASIC

This routine is called whenever BASIC wishes to call one of the KERNAL I/O routines. It is also used to handle I/O errors in BASIC.

```
., e0f9  c9 f0     cmp #$f0
., e0fb  d0 07     bne $e104
., e0fd  84 38     sty $38       ;MEMSIZ - highest address in
                                 BASIC
```

```
., e0ff  86 37    stx $37
., e101  4c 63 a6 jmp $a663   ;do CLR without aborting I/O
., e104  aa       tax         ;put error flag in (X)
., e105  d0 02    bne $e109
., e107  a2 1e    ldx #$1e
., e109  4c 37 a4 jmp $a437   ;do error
```

57612 BCHOUT: OUTPUT CHARACTER

This routine uses the KERNAL routine CHROUT to output the character in (A) to an available output channel. A test is made for possible I/O error.

```
., e10c  20 d2 ff jsr $ffd2   ;CHROUT - output character in
                              (A)
., e10f  b0 e8    bcs $e0f9   ;handle I/O error
., e111  60       rts
```

57618 BCHIN: INPUT CHARACTER

This routine uses the KERNAL routine CHRIN to input a character to (A) from an available input channel. A test is made for possible I/O error.

```
., e112  20 cf ff jsr $ffcf   ;CHRIN - get character from
                              I/P channel
., e115  b0 e2    bcs $e0f9   ;handle I/O error
., e117  60       rts
```

57624 BCKOUT: SET UP FOR OUTPUT

This routine opens an output channel ready for future output and tests for a possible I/O error. On entry, (X) must hold the logical file number as used in OPEN.

```
., e118  20 ad e4 jsr $e4ad   ;open output channel via
                              CHKOUT
., e11b  b0 dc    bcs $e0f9   ;handle I/O error
., e11d  60       rts
```

57630 BCKIN: SET UP FOR INPUT

The KERNAL routine CHKIN is used to open a channel for future input. A test is made for possible I/O error.

```
., e11e  20 c6 ff jsr $ffc6   ;CHKIN - open channel for
                              input
., e121  b0 d6    bcs $e0f9   ;handle I/O error
., e123  60       rts
```

57636 BGETIN: GET ONE CHARACTER

The KERNAL routine GETIN is used to get a character from the keyboard buffer into (A). A test is made for possible I/O error.

```
.,  e124  20 e4 ff jsr $ffe4     ;GETIN - get char from
                                 keyboard queue
.,  e127  b0 d0    bcs $e0f9     ;handle I/O error
.,  e129  60       rts
```

57642 SYS: PERFORM SYS

This is the routine which enables user machine language routines to be executed from BASIC. It is possible for initial values of all 6510 internal registers to be set by placing their values in $030C - $030F. The return address to this routine is pushed onto the stack, the register values set up and an indirect jump made to LINNUM, which holds the address of the user routine. On return, the register values are taken and stored in RAM.

```
.,  e12a  20 8a ad jsr $ad8a     ;evaluate text & confirm
                                 numeric
.,  e12d  20 f7 b7 jsr $b7f7     ;convert fac#1 to +ve integer
                                 in LINNUM
.,  e130  a9 e1    lda #$e1      ;push return address = $E146
.,  e132  48       pha
.,  e133  a9 46    lda #$46
.,  e135  48       pha
.,  e136  ad 0f 03 lda $030f     ;SPREG - user flag register
.,  e139  48       pha
.,  e13a  ad 0c 03 lda $030c     ;SAREG - user (A) register
.,  e13d  ae 0d 03 ldx $030d     ;SXREG - user (X) register
.,  e140  ac 0e 03 ldy $030e     ;SYREG - user (Y) register
.,  e143  28       plp           ;pull SPREG into (P)
.,  e144  6c 14 00 jmp ($0014)   ;execute user code -
                                 terminate with RTS
.,  e147  08       php
.,  e148  8d 0c 03 sta $030c     ;store (A) register
.,  e14b  8e 0d 03 stx $030d     ;store (X) register
.,  e14e  8c 0e 03 sty $030e     ;store (Y) register
.,  e151  68       pla
.,  e152  8d 0f 03 sta $030f     ;store (P) register
.,  e155  60       rts
```

57686 SAVET: PERFORM SAVE

The parameters for SAVE - filename, device number etc. are

obtained from text. The start and end addresses are obtained from TXTTAB and VARTAB respectively, then the KERNAL SAVE routine is called. Finally a test is made for any I/O error.

```
., e156  20 d4 e1 jsr $e1d4    ;get SAVE parameters from
                               text
., e159  a6 2d    ldx $2d      ;VARTAB - start of variables
., e15b  a4 2e    ldy $2e
., e15d  a9 2b    lda #$2b     ;<TXTTAB - start of BASIC
                               text
., e15f  20 d8 ff jsr $ffd8    ;SAVE - KERNAL save routine
., e162  b0 95    bcs $e0f9    ;handle I/O error
., e164  60       rts
```

57701 VERFYT: PERFORM VERIFY/LOAD

This routine is essentially the same for both LOAD and VERIFY. The entry point determines which is carried out, and sets VERCK accordingly. The LOAD/VERIFY parameters - filename, device number etc. are obtained from text before the KERNAL LOAD routine is called. A test is made for an I/O error. At this point, the two functions are distinguished. VERIFY reads the I/O status word and prints the message OK or ?VERIFY error depending on the result of the test. LOAD reads the I/O status word for a possible ?LOAD error, then updates the pointers to text and variables, exiting via CLR.

```
., e165  a9 01    lda #$01     ;flag VERIFY
., e167  2c a9 00 bit $00a9    ;mask - flag LOAD
., e16a  85 0a    sta $0a      ;VERCK - LOAD/VERIFY flag
., e16c  20 d4 e1 jsr $e1d4    ;get LOAD parameters
., e16f  a5 0a    lda $0a      ;VERCK
., e171  a6 2b    ldx $2b      ;TXTTAB - start of BASIC text
., e173  a4 2c    ldy $2c
., e175  20 d5 ff jsr $ffd5    ;LOAD - KERNAL routine
., e178  b0 57    bcs $e1d1    ;handle I/O error
., e17a  a5 0a    lda $0a      ;VERCK
., e17c  f0 17    beq $e195    ;handle LOAD separately
., e17e  a2 1c    ldx #$1c     ;flag ?VERIFY
., e180  20 b7 ff jsr $ffb7    ;READST - read I/O status
                               word
., e183  29 10    and #$10     ;data mismatch?
., e185  d0 17    bne $e19e    ;yes - do error
., e187  a5 7a    lda $7a      ;<TXTPTR
., e189  c9 02    cmp #$02
., e18b  f0 07    beq $e194    ;RTS
```

```
., e18d  a9 64     lda #$64
., e18f  a0 a3     ldy #$a3      ;$A364 = message 'OK
., e191  4c 1e ab jmp $ab1e      ;output string at (A/Y)
., e194  60        rts
., e195  20 b7 ff jsr $ffb7      ;READST
., e198  29 bf     and #$bf      ;any error?
., e19a  f0 05     beq $e1a1     ;no
., e19c  a2 1d     ldx #$1d      ;flag ?LOAD
., e19e  4c 37 a4 jmp $a437      ;do error
., e1a1  a5 7b     lda $7b       ;>TXTPTR
., e1a3  c9 02     cmp #$02
., e1a5  d0 0e     bne $e1b5
., e1a7  86 2d     stx $2d       ;VARTAB - start of variables
., e1a9  84 2e     sty $2e
., e1ab  a9 76     lda #$76
., e1ad  a0 a3     ldy #$a3      ;$A376 = message 'READY
., e1af  20 1e ab jsr $ab1e      ;output string at (A/Y)
., e1b2  4c 2a a5 jmp $a52a      ;do CLR & restart BASIC
., e1b5  20 8e a6 jsr $a68e      ;reset TXTPTR
., e1b8  20 33 a5 jsr $a533      ;rechain BASIC lines
., e1bb  4c 77 a6 jmp $a677      ;do RESTORE & reset OLDTXT
```

57790 OPENT: PERFORM OPEN

The parameters for OPEN are obtained from text, then the KERNAL OPEN routine is performed. A test is made for an I/O error.

```
., e1be  20 19 e2 jsr $e219      ;get parameters for OPEN
., e1c1  20 c0 ff jsr $ffc0      ;OPEN - KERNAL routine
., e1c4  b0 0b     bcs $e1d1     ;handle I/O error
., e1c6  60        rts
```

57799 CLOSET: PERFORM CLOSE

The parameters for CLOSE are obtained from text, and the logical file number placed in (A). The KERNAL CLOSE routine is performed and a test is made for an I/O error.

```
., e1c7  20 19 e2 jsr $e219      ;get parameters for CLOSE
., e1ca  a5 49     lda $49       ;<FORPNT - holds logical file
                                  number
., e1cc  20 c3 ff jsr $ffc3      ;CLOSE - KERNAL routine
., e1cf  90 c3     bcc $e194     ;RTS
., e1d1  4c f9 e0 jmp $e0f9      ;handle I/O error
```

57812 SLPARA: GET PARAMETERS FOR LOAD/SAVE

This routine gets the filename, device number and secondary

address for LOAD/SAVE operations. The KERNAL routines SETNAM and SETLFS are used to do this. Firstly the default values are set = null filename, device = cassette, secondary address = 0, then tests are made to see if any of the parameters were given. If so, these are set up in the same way. <FORPNT holds the device number on exit.

```
., e1d4  a9 00     lda #$00      ;null filename length
., e1d6  20 bd ff  jsr $ffbd     ;SETNAM - set up filename
., e1d9  a2 01     ldx #$01      ;default device = cassette
., e1db  a0 00     ldy #$00      ;SA = 0
., e1dd  20 ba ff  jsr $ffba     ;SETLFS - set up logical file
., e1e0  20 06 e2  jsr $e206     ;check default parameters
., e1e3  20 57 e2  jsr $e257     ;set up given file name
., e1e6  20 06 e2  jsr $e206     ;check default parameters
., e1e9  20 00 e2  jsr $e200     ;obtain given device number
., e1ec  a0 00     ldy #$00
., e1ee  86 49     stx $49       ;<FORPNT - holds device
                                  number
., e1f0  20 ba ff  jsr $ffba     ;SETLFS
., e1f3  20 06 e2  jsr $e206     ;check default parameters
., e1f6  20 00 e2  jsr $e200     ;obtain given secondary
                                  address
., e1f9  8a        txa
., e1fa  a8        tay
., e1fb  a6 49     ldx $49       ;<FORPNT
., e1fd  4c ba ff  jmp $ffba     ;SETLFS
```

57856 COMBYT: GET NEXT ONE BYTE PARAMETER

This short routine checks thsat the next character of text is a comma, and then inputs the parameter following it into (X).

```
., e200  20 0e e2  jsr $e20e     ;check for comma
., e203  4c 9e b7  jmp $b79e     ;input 1 byte parameter to
                                  (X)
```

57862 DEFLT: CHECK DEFAULT PARAMETERS

This routine tests CHRGOT to see if a particular optional parameter was included in text. If it was, the routine exits. If it was not, the return address on the stack is discarded, and the routine exits both this and the calling routine.

```
., e206  20 79 00  jsr $0079     ;CHRGOT
., e209  d0 02     bne $e20d     ;parameter is in text
., e20b  68        pla
```

```
., e20c  68        pla
., e20d  60        rts
```

57870 CMMERR: CHECK FOR COMMA

This routine confirms that the next character in text is a comma, and also that the comma is not immediately followed by a terminator, otherwise ?SYNTAX error.

```
., e20e  20 fd ae jsr $aefd     ;confirm comma
., e211  20 79 00 jsr $0079     ;CHRGOT
., e214  d0 f7    bne $e20d     ;no terminator - ok
., e216  4c 08 af jmp $af08     ;?SYNTAX error
```

57881 OCPARA: GET PARAMETERS FOR OPEN/CLOSE

This routine gets the logical file number, device number, secondary address and filename for OPEN/CLOSE operations. Initially the default filename is set null, and device = cassette. The logical file number is compulsory, and is obtained from text and placed in <FORPNT. The other parameters are optional and are obtained if present. The device number is stored in >FORPNT. The parameters are set via the KERNAL routines SETNAM and SETLFS.

```
., e219  a9 00    lda #$00      ;null default filename
., e21b  20 bd ff jsr $ffbd     ;SETNAM - set up filename
., e21e  20 11 e2 jsr $e211     ;confirm TXTPTR is no
                                terminator
., e221  20 9e b7 jsr $b79e     ;input 1 byte parameter to
                                (X)
., e224  86 49    stx $49       ;<FORPNT - logical file
                                number
., e226  8a       txa
., e227  a2 01    ldx #$01      ;default device = cassette
., e229  a0 00    ldy #$00      ;default SA = 0
., e22b  20 ba ff jsr $ffba     ;SETLFS - set up logical file
., e22e  20 06 e2 jsr $e206     ;check default parameters
., e231  20 00 e2 jsr $e200     ;obtain given device number
., e234  86 4a    stx $4a       ;>FORPNT - device number
., e236  a0 00    ldy #$00
., e238  a5 49    lda $49       ;<FORPNT
., e23a  e0 03    cpx #$03
., e23c  90 01    bcc $e23f
., e23e  88       dey
., e23f  20 ba ff jsr $ffba     ;SETLFS
., e242  20 06 e2 jsr $e206     ;check default parameters
., e245  20 00 e2 jsr $e200     ;obtain given secondary
                                address
```

```
., e248  8a        txa
., e249  a8        tay
., e24a  a6 4a     ldx $4a      ;FORPNT
., e24c  a5 49     lda $49
., e24e  20 ba ff  jsr $ffba    ;SETLFS
., e251  20 06 e2  jsr $e206    ;check default parameters
., e254  20 0e e2  jsr $e20e    ;check for comma
., e257  20 9e ad  jsr $ad9e    ;evaluate expression in text
., e25a  20 a3 b6  jsr $b6a3    ;do string housekeeping
., e25d  a6 22     ldx $22      ;INDEX1 - points to given
                                 filename
., e25f  a4 23     ldy $23
., e261  4c bd ff  jmp $ffbd    ;SETNAM
```

57956 COS: PERFORM COS

Fac#1 = SIN(fac#1 + PI/2). PI/2 is added to the argument in fac#1 and SIN performed. Note: the argument must be in radians.

```
., e264  a9 e0     lda #$e0
., e266  a0 e2     ldy #$e2     ;$E2E0 = pi/2 in flpt
., e268  20 67 b8  jsr $b867    ;add flpt at (A/Y) to fac#1
```

57963 SIN: PERFORM SIN

Fac#1 = SIN(fac#1). Note: the argument in fac#1 must be in radians.

```
., e26b  20 0c bc  jsr $bc0c    ;copy fac#1 to fac#2
., e26e  a9 e5     lda #$e5
., e270  a0 e2     ldy #$e2     ;$E2E5 = 2*pi in flpt
., e272  a6 6e     ldx $6e      ;ARGSGN - fac#2 sign
., e274  20 07 bb  jsr $bb07    ;divide fac#2 by flpt at
                                 A/Y)
., e277  20 0c bc  jsr $bc0c    ;copy fac#1 to fac#2
., e27a  20 cc bc  jsr $bccc    ;do INT
., e27d  a9 00     lda #$00
., e27f  85 6f     sta $6f      ;ARISGN - sign comparison
                                 result
., e281  20 53 b8  jsr $b853    ;subtract fac#1 from fac#2
., e284  a9 ea     lda #$ea
., e286  a0 e2     ldy #$e2     ;$E2EA = 0.25 in flpt
., e288  20 50 b8  jsr $b850    ;subtract fac#1 from flpt at
                                 (A/Y)
., e28b  a5 66     lda $66      ;FACSGN - fac#1 sign
., e28d  48        pha
., e28e  10 0d     bpl $e29d
., e290  20 49 b8  jsr $b849    ;add 0.5 to fac#1
```

```
.,  e293   a5 66      lda $66       ;FACSGN
.,  e295   30 09      bmi $e2a0
.,  e297   a5 12      lda $12       ;TANSGN - comparison result
.,  e299   49 ff      eor #$ff
.,  e29b   85 12      sta $12
.,  e29d   20 b4 bf   jsr $bfb4     ;negate fac#1
.,  e2a0   a9 ea      lda #$ea
.,  e2a2   a0 e2      ldy #$e2      ;$E2EA = 0.25 in flpt
.,  e2a4   20 67 b8   jsr $b867     ;add flpt at (A/Y) to fac#1
.,  e2a7   68         pla
.,  e2a8   10 03      bpl $e2ad
.,  e2aa   20 b4 bf   jsr $bfb4     ;negate fac#1
.,  e2ad   a9 ef      lda #$ef
.,  e2af   a0 e2      ldy #$e2      ;$E2EF = counter for SIN
                                    series
.,  e2b1   4c 43 e0   jmp $e043     ;evaluate series for function
```

58036 TAN: PERFORM TAN

fac#1 = TAN(fac#1). This routine finds the sine of fac#1, and then divides it by the cosine of fac#1. Intermediate products are stored in zero page locations. Note: the argument must be given in radians.

```
.,  e2b4   20 ca bb   jsr $bbca     ;store fac#1 in memory
.,  e2b7   a9 00      lda #$00
.,  e2b9   85 12      sta $12       ;TANSGN - tan sign
.,  e2bb   20 6b e2   jsr $e26b     ;do SIN
.,  e2be   a2 4e      ldx #$4e
.,  e2c0   a0 00      ldy #$00
.,  e2c2   20 f6 e0   jsr $e0f6     ;store fac#1 at (A/Y)
.,  e2c5   a9 57      lda #$57
.,  e2c7   a0 00      ldy #$00
.,  e2c9   20 a2 bb   jsr $bba2     ;load fac#1 with flpt at
                                    (A/Y)
.,  e2cc   a9 00      lda #$00
.,  e2ce   85 66      sta $66       ;FACSGN - fac#1 sign
.,  e2d0   a5 12      lda $12       ;TANSGN
.,  e2d2   20 dc e2   jsr $e2dc     ;do COS
.,  e2d5   a9 4e      lda #$4e
.,  e2d7   a0 00      ldy #$00
.,  e2d9   4c 0f bb   jmp $bb0f     ;do divide
.,  e2dc   48         pha
.,  e2dd   4c 9d e2   jmp $e29d
```

58080 PI2: TABLE OF TRIG CONSTANTS

The following constants are held in 5 byte flpt for trig evaluation:

```
$E2E0 = 1.570796327 (pi/2)
$E2E5 = 6.28318531  (pi*2)
$E2EA = 0.25
$E2EF = #05 (1 byte counter for SIN series)
```

```
.:e2e0  81   49 0f da a2 83 49 0f  $E2F0 = -14.3813907    (SIN
                                   constant 1)
.:e2e8  da   a2 7f 00 00 00 00 05  $E2F5 -  42.0077971    (SIN
                                   constant 2)
.:e2f0  84   e6 1a 2d 1b 86 28 07  $E2FA = -76.7041703    (SIN
                                   constant 3)
.:e2f8  fb   f8 87 99 68 89 01 87  $E2FF =  81.6052237    (SIN
                                   constant 4)
.:e300  23   35 df e1 86 a5 5d e7  $E304 = -41.3417021    (SIN
                                   constant 5)
.:e308  28   83 49 0f da a2 a5 66  $E309 =    6.28318531  (SIN
                                   constant 6)
```

58126 ATN: PERFORM ATN

fac#1 = ATN(fac#1). This produces an answer in radians. The calculation is based on a long but very simple series of 12 terms. The series follows the pattern X - X /3 + X /5 - ... + X /21 - X /23

```
., e30e  a5 66     lda $66      ;push FACSGN - fac#1 sign
., e310  48        pha
., e311  10 03     bpl $e316
., e313  20 b4 bf  jsr $bfb4    ;negate fac#1
., e316  a5 61     lda $61      ;push FACEXP - fac#1 exponent
., e318  48        pha
., e319  c9 81     cmp #$81
., e31b  90 07     bcc $e324
., e31d  a9 bc     lda #$bc
., e31f  a0 b9     ldy #$b9     ;$B9BC = 1 in flpt
., e321  20 0f bb  jsr $bb0f    ;divide flpt at (A/Y) by
                                fac#1
., e324  a9 3e     lda #$3e
., e326  a0 e3     ldy #$e3     ;$E33E = counter for ATN
                                series
., e328  20 43 e0  jsr $e043    ;evaluate series for function
., e32b  68        pla
., e32c  c9 81     cmp #$81
., e32e  90 07     bcc $e337
., e330  a9 e0     lda #$e0
```

```
., e332  a0 e2     ldy #$e2        ;$E2E0 = pi/2 in flpt
., e334  20 50 b8  jsr $b850       ;sub fac#1 from flpt at A/Y)
., e337  68        pla
., e338  10 03     bpl $e33d
., e33a  4c b4 bf  jmp $bfb4       ;negate fac#1
., e33d  60        rts
```

58174 ATNCON: TABLE OF ATN CONSTANTS

The table holds a 1 byte counter an the following 5 byte flpt constants:

```
            $E33E = #0B (counter for ATN series)
            $E33F = -0.000684793912 (constant 1)
            $E344 =  0.00485094216  (constant 2)
            $E349 = -0.161117018    (constant 3)
            $E34E =  0.034209638    (constant 5)
            $E353 = -0.0542791328   (constant 6)
            $E358 =  0.0724571965   (constant 7)
            $E35D = -0.0898023954   (constant 8)
            $E362 =  0.110932413    (constant 9)
            $E367 = -0.142839808    (constant 10)
            $E36C =  0.19999912     (constant 11)
            $E371 = -0.333333316    (constant 12)
            $E374 =  1              (constant 13)
```

```
.:e33e 0b 76 b3 83 bd d3 79 1e
.:e346 f4 a6 f5 7b 83 fc b0 10
.:e34e 7c 0c 1f 67 ca 7c de 53
.:e356 cb c1 7d 14 64 70 4c 7d
.:e35e b7 ea 51 7a 7d 63 30 88
.:e366 7e 7e 92 44 99 3a 7e 4c
.:e36e cc 91 c7 7f aa aa aa 13
.:e376 81 00 00 00 00 20 cc ff
```

58235 BASSFT: BASIC WARM RESTART

This is the BASIC warm restart routine that is vectored at the very start of the BASIC ROM. The routine is called by the 6510 BRK instruction, or STOP/RESTORE being pressed. It outputs the READY prompt via the vector at ($0300). Note: If bit 7 of (X) was not a 1, then an error message would be produced.

```
., e37b  20 cc ff  jsr $ffcc       ;CLRCHN - close I/O channels
., e37e  a9 00     lda #$00
., e380  85 13     sta $13         ;input prompt flag
., e382  20 7a a6  jsr $a67a       ;do CLR
., e385  58        cli
., e386  a2 80     ldx #$80
```

```
., e388  6c 00 03 jmp ($0300)  ;IERROR - vector to error
                                routine
., e38b  8a       txa
., e38c  30 03    bmi $e391
., e38e  4c 3a a4 jmp $a43a    ;do error
., e391  4c 74 a4 jmp $a474    ;restart BASIC
```

58260 INIT: BASIC COLD RESTART

This is the BASIC cold restart routine that is vectored at the very start of the BASIC ROM. BASIC vectors and variables are initialised, the power-up message is output and BASIC restarted.

```
., e394  20 53 e4 jsr $e453    ;initialise vectors
., e397  20 bf e3 jsr $e3bf    ;initialise BASIC
., e39a  20 22 e4 jsr $e422    ;output power-up message
., e39d  a2 fb    ldx #$fb     ;reset stack
., e39f  9a       txs
., e3a0  d0 e4    bne $e386    ;restart BASIC
```

58274 INITAT: CHRGET FOR ZERO PAGE

This is the CHRGET routine which is transferred to RAM starting at $0073 on power-up or reset.

```
., e3a2  e6 7a    inc $7a
., e3a4  d0 02    bne $e3a8
., e3a6  e6 7b    inc $7b
., e3a8  ad 60 ea lda $ea60
., e3ab  c9 3a    cmp #$3a
., e3ad  b0 0a    bcs $e3b9
., e3af  c9 20    cmp #$20
., e3b1  f0 ef    beq $e3a2
., e3b3  38       sec
., e3b4  e9 30    sbc #$30
., e3b6  38       sec
., e3b7  e9 d0    sbc #$d0
., e3b9  60       rts
```

58298 RNDSED: RND SEED FOR ZERO PAGE

This is the initial value of the seed for the random number function. It is copied into RAM from $008B - $008F. Its flpt value is 0.811635157.

```
.:e3ba 80 4f c7 52 58
```

58303 INITCZ: INITIALISE BASIC RAM

This routine sets the USR jump instruction to point to

?ILLEGAL QUANTITY error, sets ADRAY1 and ADRAY2, copies CHRGET and the initial RND seed value into RAM, sets up the start and end locations for BASIC text and sets the first text byte to zero.

```
., e3bf  a9 4c     lda #$4c
., e3c1  85 54     sta $54
., e3c3  8d 10 03 sta $0310    ;USRPOK - set USR JMP
                               instruction
., e3c6  a9 48     lda #$48
., e3c8  a0 b2     ldy #$b2
., e3ca  8d 11 03 sta $0311    ;set USRADD = $B248 ie
                               ?ILLEGAL QTY
., e3cd  8c 12 03 sty $0312
., e3d0  a9 91     lda #$91
., e3d2  a0 b3     ldy #$b3
., e3d4  85 05     sta $05     ;set ADRAY2 = $B391
., e3d6  84 06     sty $06
., e3d8  a9 aa     lda #$aa
., e3da  a0 b1     ldy #$b1
., e3dc  85 03     sta $03     ;set ADRAY1 = $B1AA
., e3de  84 04     sty $04
., e3e0  a2 1c     ldx #$1c
., e3e2  bd a2 e3 lda $e3a2,x  ;store CHRGET routine in RAM
., e3e5  95 73     sta $73,x
., e3e7  ca        dex
., e3e8  10 f8     bpl $e3e2
., e3ea  a9 03     lda #$03
., e3ec  85 53     sta $53
., e3ee  a9 00     lda #$00
., e3f0  85 68     sta $68     ;BITS - fac#1 overflow
., e3f2  85 13     sta $13     ;input prompt flag
., e3f4  85 18     sta $18     ;LASTPT
., e3f6  a2 01     ldx #$01
., e3f8  8e fd 01 stx $01fd
., e3fb  8e fc 01 stx $01fc
., e3fe  a2 19     ldx #$19
., e400  86 16     stx $16     ;TEMPPT - pointer to
                               descriptor stack
., e402  38        sec
., e403  20 9c ff jsr $ff9c    ;read MEMBOT
., e406  86 2b     stx $2b     ;set TXTTAB = bottom of RAM
., e408  84 2c     sty $2c
., e40a  38        sec
., e40b  20 99 ff jsr $ff99    ;read MEMTOP
., e40e  86 37     stx $37     ;set MEMSIZ = MEMTOP
., e410  84 38     sty $38
```

```
., e412  86 33     stx $33       ;set FRETOP = MEMTOP
., e414  84 34     sty $34
., e416  a0 00     ldy #$00
., e418  98        tya
., e419  91 2b     sta ($2b),y   ;place zero at start of BASIC
., e41b  e6 2b     inc $2b       ;increment TXTTAB
., e41d  d0 02     bne $e421
., e41f  e6 2c     inc $2c
., e421  60        rts
```

58402 INITMS: OUTPUT POWER-UP MESSAGE

This routine outputs the message **** COMMODORE BASIC V2 **** <CR> 64K RAM SYSTEM <CR>. It then calculates the number of bytes free by subtracting TXTTAB from MEMSIZ, outputs this number and prints the BASIC BYTES FREE message. The routine exits via NEW.

```
., e422  a5 2b     lda $2b       ;TXTTAB - start of BASIC
., e424  a4 2c     ldy $2c
., e426  20 08 a4  jsr $a408     ;check memory for overlap
., e429  a9 73     lda #$73
., e42b  a0 e4     ldy #$e4      ;$E473 = 'CBM BASIC message
., e42d  20 1e ab  jsr $ab1e     :output string at (A/Y)
., e430  a5 37     lda $37       ;MEMSIZ - highest address in
                                 BASIC
., e432  38        sec
., e433  e5 2b     sbc $2b       ;subtract TXTTAB - gives
                                 bytes free
., e435  aa        tax
., e436  a5 38     lda $38
., e438  e5 2c     sbc $2c
., e43a  20 cd bd  jsr $bdcd     ;output number in (A/X)
., e43d  a9 60     lda #$60
., e43f  a0 e4     ldy #$e4      ;$E460 = message 'BASIC BYTES
                                 FREE
., e441  20 1e ab  jsr $ab1e     ;output message from table
., e444  4c 44 a6  jmp $a644     ;perform NEW
```

58439 BVTRS: TABLE OF BASIC VECTORS

This table contains the addresses of the vectored BASIC routines. They are copied into RAM, starting at $0300 by the next routine.

```
.:e447 8b e3 83 a4 7c a5 1a a7
.:e44f e4 a7 86 ae
```

58451 INITV: INITIALISE VECTORS

This routine copies the vectors from the table starting at

$E447 into RAM, starting at $0300.

```
., e453  a2 0b     ldx #$0b     ;counter for vector table
., e455  bd 47 e4  lda $e447,x
., e458  9d 00 03  sta $0300,x  ;store vectors from $0300 -
                                 $030B
., e45b  ca        dex
., e45c  10 f7     bpl $e455
., e45e  60        rts
```

58463 WORDS: POWER UP MESSAGE

This is the message displayed on the screen when the Commodore 64 is first switched on. The byte after the message string ($E4AC = #5C) appears to be a checksum or identification code. See also appendix 1.

```
.:e45f 00 20 42 41 53 49 43 20    basic
.:e467 42 59 54 45 53 20 46 52  bytes fr
.:e46f 45 45 0d 00 93 0d 20 20  ee
.:e477 20 20 2a 2a 2a 2a 20 43    **** c
.:e47f 4f 4d 4d 4f 44 4f 52 45  ommodore
.:e487 20 36 34 20 42 41 53 49   64 basi
.:e48f 43 20 56 32 20 2a 2a 2a  c v2 ***
.:e497 2a 0d 0d 20 36 34 4b 20  *   64k
.:e49f 52 41 4d 20 53 59 53 54  ram syst
.:e4a7 45 4d 20 20 00 5c        em
```

58541 PATCH FOR BASIC CHKOUT CALL

This is a short patch added to the KERNAL ROM to preserve (A) when there was no error returned from BASIC calling the CHKOUT routine. This corrects a bug in the early versions of PRINT# and CMD.

```
., e4ad  48        pha
., e4ae  20 c9 ff  jsr $ffc9    ;KERNAL routine CHKOUT
., e4b1  aa        tax
., e4b2  68        pla
., e4b3  90 01     bcc $e4b6
., e4b5  8a        txa
., e4b6  60        rts
```

58551 UNUSED BYTES FOR LATER PATCH ROUTINES

```
.:e4b7 aa aa aa aa aa aa aa aa
.:e4bf aa aa aa aa aa aa aa aa
```

```
.:e4c7 aa aa aa aa aa aa aa aa
.:e4cf aa aa aa aa aa aa aa aa
.:e4d7 aa aa aa
```

58586 RESET CHARACTER COLOUR

This patch routine sets the current character colour code equal to the background colour. The routine is called by 'clear a screen line', and is the source of the apparent bug which makes characters POKEd to the screen invisible. This bug has been fixed in the new version 3 ROM (see appendix 1).

```
., e4da  ad 21 d0 lda $d021      ;background colour
., e4dd  91 f3    sta ($f3),y   ;current screen colour
., e4df  60       rts
```

58592 PAUSE AFTER FINDING TAPE FILE

When a file is searched for on tape, the message SEARCHING... is displayed. Once the file is found, the message FOUND... is displayed. At this point, the original KERNAL routines required that the CBM key be hit before the file would be loaded. This patch enables the load to continue after a short pause regardless of the keypress.

```
., e4e0  69 02    adc #$02
., e4e2  a4 91    ldy $91        ;STKEY - stop/rvs flag
., e4e4  c8       iny
., e4e5  d0 04    bne $ed
., e4e7  c5 a1    cmp $a1
., e4e9  d0 f7    bne $ed
., e4eb  60       rts
```

58604 RS-232 TIMING TABLE - PAL

This table contains the prescaler values for setting up the standard RS-232 baud rates. The table corresponds exactly with the NTSC table at $FEC2. The RS-232 baud rates need separate prescaler tables because European PAL machines operate at a slightly lower system clock frequency than American NTSC machines.

```
.:e4ec 19 26 44 19 1a 11 e8 0d
.:e4f4 70 0c 06 06 d1 02 37 01
.:e4fc ae 00 69 00
```

58624 IOBASE: GET I/O ADDRESS

The KERNAL routine IOBASE ($FFF3) jumps to this routine. It

returns the base address ($DC00) of the I/O registers in (X/Y). The format is Lo/Hi.

```
., e500  a2 00    ldx #$00
., e502  a0 dc    ldy #$dc       ;set (X/Y) = $DC00
., e504  60       rts
```

58629 SCREEN: GET SCREEN SIZE

The KERNAL routine SCREEN ($FFED) jumps to this routine. It returns the screen size; columns in (X) =#28 =40, and rows in (Y) =#19 =25.

```
., e505  a2 28    ldx #$28       ;40 columns
., e507  a0 19    ldy #$19       ;25 rows
., e509  60       rts
```

58634 PLOT: PUT/GET ROW AND COLUMN

The KERNAL routine PLOT ($FFF0) jumps to this routine. The option taken depends on the state of the carry flag on entry. If it is set, the column is placed in (Y) and the row in (X). If it is clear, the cursor position is read from (X/Y) and the screen pointers are set.

```
., e50a  b0 07    bcs $e513      ;get row/column
., e50c  86 d6    stx $d6        ;TBLX - current physical line
                                 number
., e50e  84 d3    sty $d3        ;PNTR - cursor column on line
., e510  20 6c e5 jsr $e56c      ;set screen pointers
., e513  a6 d6    ldx $d6        ;TBLX
., e515  a4 d3    ldy $d3        ;PNTR
., e517  60       rts
```

58648 CINT1: INITIALISE I/O

This routine is part of the KERNAL CINT initialisation routine. The I/O default values are set. Shift keys are disabled and the cursor switched off. The vector to the keyboard decode table is set up and the length of the keyboard buffer is set to 10 characters. The cursor colour is set to light blue and the repeat parameters set up.

```
., e518  20 a0 e5 jsr $e5a0      ;set I/O defaults
., e51b  a9 00    lda #$00
., e51d  8d 91 02 sta $0291      ;MODE -  disable shift keys
., e520  85 cf    sta $cf        ;BLNON - cursor blink flag
., e522  a9 48    lda #$48
```

```
., e524  8d 8f 02 sta $028f     ;KEYLOG - vector to keybd
                                decode
., e527  a9 eb    lda #$eb      ;vector = $EB48
., e529  8d 90 02 sta $0290
., e52c  a9 0a    lda #$0a
., e52e  8d 89 02 sta $0289     ;XMAX-size of keybd buffer=10
., e531  8d 8c 02 sta $028c     ;DELAY - repeat delay counter
., e534  a9 0e    lda #$0e      ;flag light blue
., e536  8d 86 02 sta $0286     ;COLOR - current character
                                colour
., e539  a9 04    lda #$04
., e53b  8d 8b 02 sta $028b     ;KOUNT - repeat speed counter
., e53e  a9 0c    lda #$0c
., e540  85 cd    sta $cd       ;BLNCT - cursor toggle timer
., e542  85 cc    sta $cc       ;BLNSW - cursor enable
```

58692 CLEAR SCREEN

The page number at which the top of the screen starts is ORed with #80 and the screen line link table reset. Finally, the screen is cleared line by line, starting at the bottom and working up. Once this is finished, the next routine is used to home the cursor.

```
., e544  ad 88 02 lda $0288     ;HIBASE - top of screen RAM
                                (page)
., e547  09 80    ora #$80
., e549  a8       tay
., e54a  a9 00    lda #$00
., e54c  aa       tax
., e54d  94 d9    sty $d9,x     ;LDTB1 - screen line link
                                table
., e54f  18       clc
., e550  69 28    adc #$28      ;add 40 to screen address
., e552  90 01    bcc $e555
., e554  c8       iny           ;address for line is on next
                                page
., e555  e8       inx
., e556  e0 1a    cpx #$1a      ;last line done?
., e558  d0 f3    bne $e54d     ;no - set line link
., e55a  a9 ff    lda #$ff
., e55c  95 d9    sta $d9,x     ;last pointer
., e55e  a2 18    ldx #$18
., e560  20 ff e9 jsr $e9ff     ;clear screen line
., e563  ca       dex
., e564  10 fa    bpl $e560     ;clear next line up
```

58726 HOME CURSOR

The pointers to the cursor line and column are set to zero.

```
., e566  a0 00     ldy #$00
., e568  84 d3     sty $d3       ;PNTR - cursor column on line
., e56a  84 d6     sty $d6       ;TBLX - current physical line
                                 number
```

58732 SET SCREEN POINTERS

This routine positions the cursor on the screen and sets up the screen pointers. On entry, TBLX must hold the line number, and PNTR the column number of the cursor position.

```
., e56c  a6 d6     ldx $d6       ;TBLX - current physical line
                                 number
., e56e  a5 d3     lda $d3       ;PNTR - cursor column on line
., e570  b4 d9     ldy $d9,x     ;LDTB1 - screen line link
                                 table
., e572  30 08     bmi $e57c
., e574  18        clc
., e575  69 28     adc #$28
., e577  85 d3     sta $d3       ;PNTR
., e579  ca        dex
., e57a  10 f4     bpl $e570
., e57c  b5 d9     lda $d9,x     ;LDTB1
., e57e  29 03     and #$03
., e580  0d 88 02 ora $0288      ;HIBASE - page for top of
                                 screen RAM
., e583  85 d2     sta $d2       ;>PNT - current screen line
                                 address
., e585  bd f0 ec lda $ecf0,x    ;table of screen line address
                                 lo bytes
., e588  85 d1     sta $d1       ;<PNT
., e58a  a9 27     lda #$27
., e58c  e8        inx
., e58d  b4 d9     ldy $d9,x     ;LDTB1
., e58f  30 06     bmi $e597
., e591  18        clc
., e592  69 28     adc #$28
., e594  e8        inx
., e595  10 f6     bpl $e58d
., e597  85 d5     sta $d5       ;LNMX - physical screen line
                                 length
., e599  60        rts
```

58778 SET I/O DEFAULTS

The default output device is set to 3 (screen), and the default input device is set to 0 (keyboard). The VIC II

chip registers are set from the video chip setup table. The cursor is then set to the home position.

```
., e59a  20 a0 e5 jsr $e5a0    ;set I/O defaults
., e59d  4c 66 e5 jmp $e566    ;home cursor
., e5a0  a9 03    lda #$03
., e5a2  85 9a    sta $9a      ;DFLTO - default output
                               device
., e5a4  a9 00    lda #$00
., e5a6  85 99    sta $99      ;DFLTN - default input device
., e5a8  a2 2f    ldx #$2f
., e5aa  bd b8 ec lda $ecb8,x  ;VIC II chip setup table
., e5ad  9d ff cf sta $cfff,x  ;VIC II chip I/O registers
., e5b0  ca       dex
., e5b1  d0 f7    bne $e5aa
., e5b3  60       rts
```

58804 LP2: GET CHARACTER FROM KEYBOARD BUFFER

It is assumed that there is at least one character in the keyboard queue. This character is obtained and the rest of the queue is moved up by one to overwrite it. On exit, the character is in (A).

```
., e5b4  ac 77 02 ldy $0277    ;KEYD - keyboard buffer queue
., e5b7  a2 00    ldx #$00
., e5b9  bd 78 02 lda $0278,x  ;overwrite with next in queue
., e5bc  9d 77 02 sta $0277,x
., e5bf  e8       inx
., e5c0  e4 c6    cpx $c6      ;NDX - number of characters
                               in queue
., e5c2  d0 f5    bne $e5b9    ;move up next character
., e5c4  c6 c6    dec $c6      ;NDX
., e5c6  98       tya
., e5c7  58       cli
., e5c8  18       clc
., e5c9  60       rts
```

58826 INPUT FROM KEYBOARD

This routine uses the previous routine to get characters from the keyboard buffer. Each character is output to the screen, unless it is <shift/RUN> when the contents of the keyboard buffer are replaced with LOAD <CR> RUN <CR>. The routine ends when a carriage return is encountered.

```
., e5ca  20 16 e7 jsr $e716    ;output to screen
., e5cd  a5 c6    lda $c6      ;NDX - # characters in
                               keyboard queue
```

```
., e5cf  85 cc    sta $cc       ;BLNSW - cursor blink enable
., e5d1  8d 92 02 sta $0292     ;AUTODN - auto scroll down
                                flag
., e5d4  f0 f7    beq $e5cd
., e5d6  78       sei
., e5d7  a5 cf    lda $cf       ;BLNON - last cursor blink
                                (on/off)
., e5d9  f0 0c    beq $e5e7
., e5db  a5 ce    lda $ce       ;GDBLN - character under
                                cursor
., e5dd  ae 87 02 ldx $0287     ;GDCOL - background colour
                                under cursor
., e5e0  a0 00    ldy #$00
., e5e2  84 cf    sty $cf       ;BLNON
., e5e4  20 13 ea jsr $ea13     ;print to screen
., e5e7  20 b4 e5 jsr $e5b4     ;get character from keyboard
                                buffer
., e5ea  c9 83    cmp #$83      ;shift/RUN pressed?
., e5ec  d0 10    bne $e5fe     ;no
., e5ee  a2 09    ldx #$09
., e5f0  78       sei
., e5f1  86 c6    stx $c6       ;NDX
., e5f3  bd e6 ec lda $ece6,x   ;shift/RUN equivalent message
., e5f6  9d 76 02 sta $0276,x   ;KEYD - keyboard buffer
., e5f9  ca       dex
., e5fa  d0 f7    bne $e5f3     ;write next character to
                                buffer
., e5fc  f0 cf    beq $e5cd     ;finished - restart input
., e5fe  c9 0d    cmp #$0d      ;carriage return pressed?
., e600  d0 c8    bne $e5ca     ;no - restart routine
., e602  a4 d5    ldy $d5       ;LNMX - screen line length
., e604  84 d0    sty $d0       ;CRSW - flag INPUT/GET from
                                keyboard
., e606  b1 d1    lda ($d1),y   ;PNT - screen address
., e608  c9 20    cmp #$20      ;space?
., e60a  d0 03    bne $e60f     ;no
., e60c  88       dey
., e60d  d0 f7    bne $e606
., e60f  c8       iny
., e610  84 c8    sty $c8       ;INDX - end of logical line
                                for input
., e612  a0 00    ldy #$00
., e614  8c 92 02 sty $0292     ;AUTODN
., e617  84 d3    sty $d3       ;PNTR - cursor column
., e619  84 d4    sty $d4       ;QTSW - editor not in quotes
mode.
., e61b  a5 c9    lda $c9       ;LXSP - cursor X-Y position
at start
., e61d  30 1b    bmi $e63a
```

```
., e61f  a6 d6     ldx $d6      ;TBLX - cursor line number
., e621  20 ed e6  jsr $e6ed    ;retreat cursor
., e624  e4 c9     cpx $c9      ;LXSP
., e626  d0 12     bne $e63a
., e628  a5 ca     lda $ca
., e62a  85 d3     sta $d3      ;PNTR
., e62c  c5 c8     cmp $c8      ;INDX
., e62e  90 0a     bcc $e63a
., e630  b0 2b     bcs $e65d
```

58930 INPUT FROM SCREEN OR KEYBOARD

This routine is used by INPUT to input data from devices not on the Commodore serial bus. ie. from the screen or keyboard. On entry, both the (X) and (Y) registers are preserved. A test is made to determine which device the input is to be from. If it is the screen, then quotes and <RVS> are tested for and the character is echoed on the screen. Keyboard inputs make use of the previous routine.

```
., e632  98        tya          ;preserve (X) and (Y)
                                registers
., e633  48        pha
., e634  8a        txa
., e635  48        pha
., e636  a5 d0     lda $d0      ;CRSW - flag INPUT/GET from
                                keyboard
., e638  f0 93     beq $e5cd    ;input from keyboard
., e63a  a4 d3     ldy $d3      ;PNTR - cursor column
., e63c  b1 d1     lda ($d1),y  ;current screen address
., e63e  85 d7     sta $d7      ;temp data
., e640  29 3f     and #$3f
., e642  06 d7     asl $d7
., e644  24 d7     bit $d7
., e646  10 02     bpl $e64a
., e648  09 80     ora #$80
., e64a  90 04     bcc $e650
., e64c  a6 d4     ldx $d4      ;QTSW - editor in quotes
                                mode?
., e64e  d0 04     bne $e654    ;yes
., e650  70 02     bvs $e654
., e652  09 40     ora #$40
., e654  e6 d3     inc $d3      ;PNTR
., e656  20 84 e6  jsr $e684    ;do quotes test
., e659  c4 c8     cpy $c8      ;INDX - end of logical line
                                for input
., e65b  d0 17     bne $e674
., e65d  a9 00     lda #$00
., e65f  85 d0     sta $d0      ;CRSW
```

```
., e661  a9 0d     lda #$0d
., e663  a6 99     ldx $99        ;DFLTN - default input device
., e665  e0 03     cpx #$03       ;screen?
., e667  f0 06     beq $e66f      ;yes
., e669  a6 9a     ldx $9a        ;DFLTO - default output
                                  device
., e66b  e0 03     cpx #$03       ;screen?
., e66d  f0 03     beq $e672      ;yes
., e66f  20 16 e7  jsr $e716      ;output to screen
., e672  a9 0d     lda #$0d
., e674  85 d7     sta $d7
., e676  68        pla
., e677  aa        tax
., e678  68        pla
., e679  a8        tay
., e67a  a5 d7     lda $d7
., e67c  c9 de     cmp #$de
., e67e  d0 02     bne $e682
., e680  a9 ff     lda #$ff
., e682  18        clc
., e683  60        rts
```

59012 QUOTES TEST

If (A) holds ASCII quotes, then the editor quotes flag is toggled on/off, ie from 0 to 1 or vice versa. Since the contents of (A) are destroyed in this process, (A) is restored to #22 on exit.

```
., e684  c9 22     cmp #$22       ;ASCII quotes?
., e686  d0 08     bne $e690      ;no - RTS
., e688  a5 d4     lda $d4        ;QTSW - editor in quotes mode
                                  flag
., e68a  49 01     eor #$01       ;toggle quotes mode on/off
., e68c  85 d4     sta $d4
., e68e  a9 22     lda #$22       ;restore value in (A)
., e690  60        rts
```

59025 SET UP SCREEN PRINT

The RVS flag is tested to see if reversed characters are to be printed, and if they are, bit 7 is set to 1. If insert mode is on, then the insert counter is decremented by 1. The character colour code is placed in (X) and the character is printed to the screen and the cursor advanced.

```
., e691  09 40     ora #$40
., e693  a6 c7     ldx $c7        ;RVS - print reverse
                                  characters?
```

```
., e695  f0 02    beq $e699   ;no
., e697  09 80    ora #$80    ;set bit 7 to reverse
                              character
., e699  a6 d8    ldx $d8     ;INSRT - insert mode on?
., e69b  f0 02    beq $e69f   ;no
., e69d  c6 d8    dec $d8     ;now one less insert
., e69f  ae 86 02 ldx $0286   ;COLOR - current character
                              colour code
., e6a2  20 13 ea jsr $ea13   ;print to screen
., e6a5  20 b6 e6 jsr $e6b6   ;advance cursor
., e6a8  68       pla
., e6a9  a8       tay
., e6aa  a5 d8    lda $d8     ;INSRT
., e6ac  f0 02    beq $e6b0
., e6ae  46 d4    lsr $d4     ;QTSW - editor in quotes flag
., e6b0  68       pla
., e6b1  aa       tax
., e6b2  68       pla
., e6b3  18       clc
., e6b4  58       cli
., e6b5  60       rts
```

59062 ADVANCE CURSOR

The cursor is advanced by one position on the screen. If this puts it beyond the 40th column, then it is placed at the beginning of the next line. If the length of the screen line <80, then the new line is linked to the previous one. A space is opened if data already exists on the new line. If the cursor has reached the bottom of the screen, then the screen is scrolled down.

```
., e6b6  20 b3 e8 jsr $e8b3   ;check line increment
., e6b9  e6 d3    inc $d3     ;PNTR - cursor column on
                              current line
., e6bb  a5 d5    lda $d5     ;LNMX - physical screen line
                              length
., e6bd  c5 d3    cmp $d3     ;PNTR
., e6bf  b0 3f    bcs $e700   ;not beyond end of line - RTS
., e6c1  c9 4f    cmp #$4f    ;79?
., e6c3  f0 32    beq $e6f7   ;put cursor on next line
., e6c5  ad 92 02 lda $0292   ;AUTODN - auto scroll down
                              flag
., e6c8  f0 03    beq $e6cd   ;auto scroll is on
., e6ca  4c 67 e9 jmp $e967   ;open a space on the screen
., e6cd  a6 d6    ldx $d6     ;TBLX - current line number
., e6cf  e0 19    cpx #$19    ;25?
., e6d1  90 07    bcc $e6da
., e6d3  20 ea e8 jsr $e8ea   ;scroll screen
```

```
.,  e6d6   c6 d6     dec $d6        ;TBLX
.,  e6d8   a6 d6     ldx $d6
.,  e6da   16 d9     asl $d9,x      ;LDTB1 - screen line link
                                    table
.,  e6dc   56 d9     lsr $d9,x
.,  e6de   e8        inx
.,  e6df   b5 d9     lda $d9,x
.,  e6e1   09 80     ora #$80
.,  e6e3   95 d9     sta $d9,x
.,  e6e5   ca        dex
.,  e6e6   a5 d5     lda $d5        ;LNMX
.,  e6e8   18        clc
.,  e6e9   69 28     adc #$28
.,  e6eb   85 d5     sta $d5
```

59117 RETREAT CURSOR

The screen line link table is searched, and then the start of line is set. The rest of the routine sets the cursor onto the next line for the previous routine.

```
.,  e6ed   b5 d9     lda $d9,x      ;LDTB1 - screen line link
                                    table
.,  e6ef   30 03     bmi $e6f4
.,  e6f1   ca        dex
.,  e6f2   d0 f9     bne $e6ed
.,  e6f4   4c f0 e9  jmp $e9f0      ;set start of line
.,  e6f7   c6 d6     dec $d6        ;TBLX - current physical line
                                    number
.,  e6f9   20 7c e8  jsr $e87c      ;go to next line
.,  e6fc   a9 00     lda #$00
.,  e6fe   85 d3     sta $d3        ;PNTR - cursor column on line
.,  e700   60        rts
```

59137 BACK ON TO PREVIOUS LINE

This routine is called when using <DEL> and <cursor LEFT>. The line number is tested, and if the cursor is already on the top line, then no further action is taken. The screen pointers are set up and the cursor placed at the end of the previous line.

```
.,  e701   a6 d6     ldx $d6        ;TBLX - current physical line
                                    number
.,  e703   d0 06     bne $e70b
.,  e705   86 d3     stx $d3        ;PNTR - cursor column on line
.,  e707   68        pla
.,  e708   68        pla
.,  e709   d0 9d     bne $e6a8
```

```
., e70b  ca        dex
., e70c  86 d6     stx $d6       ;TBLX
., e70e  20 6c e5  jsr $e56c     ;set screen pointers
., e711  a4 d5     ldy $d5       ;LNMX - physical screen line
                                 length
., e713  84 d3     sty $d3       ;PNTR
., e715  60        rts
```

59158 OUTPUT TO SCREEN

This routine is part of the main KERNAL CHROUT routine. It prints CBM ASCII characters to the screen and takes care of all the screen editing characters. The cursor is automatically updated and scrolling occurs where necessary. On entry, (A) must hold the character to be output. For convenience, the routine is split into sections showing the processing of both shifted and unshifted characters.

```
., e716  48        pha
., e717  85 d7     sta $d7       ;temp data area
., e719  8a        txa
., e71a  48        pha
., e71b  98        tya
., e71c  48        pha
., e71d  a9 00     lda #$00
., e71f  85 d0     sta $d0       ;CRSW - flag INPUT/GET from
                                 keyboard
., e721  a4 d3     ldy $d3       ;PNTR - cursor column on line
., e723  a5 d7     lda $d7       ;temp - character for output
., e725  10 03     bpl $e72a     ;do unshifted characters
., e727  4c d4 e7  jmp $e7d4     ;do shifted characters
```

UNSHIFTED CHARACTERS. Ordinary unshifted ASCII characters and PET graphics are output directly to the screen. The following control codes are trapped and processed: <RETURN>, <DEL>, <HOME>, <CRSR RIGHT> <CRSR DOWN>. If either insert mode is on or quotes are open (except for <DEL>) then the control characters are not processed but output as reversed ASCII literals.

```
., e72a  c9 0d     cmp #$0d      ;<RETURN>?
., e72c  d0 03     bne $e731     ;no
., e72e  4c 91 e8  jmp $e891     ;do <RETURN>
., e731  c9 20     cmp #$20      ;ASCII space?
., e733  90 10     bcc $e745
., e735  c9 60     cmp #$60      ;1st PET graphic character?
., e737  90 04     bcc $e73d
., e739  29 df     and #$df
., e73b  d0 02     bne $e73f
```

```
., e73d  29 3f     and #$3f
., e73f  20 84 e6  jsr $e684     ;do quotes test
., e742  4c 93 e6  jmp $e693     ;set up screen print
., e745  a6 d8     ldx $d8       ;INSRT - insert mode flag
., e747  f0 03     beq $e74c     ;mode not set
., e749  4c 97 e6  jmp $e697     ;output reversed character
., e74c  c9 14     cmp #$14      ;<DEL>?
., e74e  d0 2e     bne $e77e     ;no
., e750  98        tya           ;(Y) holds cursor column
., e751  d0 06     bne $e759     ;not start of line
., e753  20 01 e7  jsr $e701     ;back on to previous line
., e756  4c 73 e7  jmp $e773
., e759  20 a1 e8  jsr $e8a1     ;check line decrement
., e75c  88        dey
., e75d  84 d3     sty $d3       ;PNTR
., e75f  20 24 ea  jsr $ea24     ;synchronise colour pointer
., e762  c8        iny
., e763  b1 d1     lda ($d1),y   ;move character back 1
., e765  88        dey
., e766  91 d1     sta ($d1),y
., e768  c8        iny
., e769  b1 f3     lda ($f3),y   ;move colour back 1
., e76b  88        dey
., e76c  91 f3     sta ($f3),y
., e76e  c8        iny
., e76f  c4 d5     cpy $d5       ;LNMX - physical screen line
                                 length
., e771  d0 ef     bne $e762     ;move next character back
., e773  a9 20     lda #$20
., e775  91 d1     sta ($d1),y   ;store delete character on
                                 screen
., e777  ad 86 02  lda $0286     ;COLOR - current character
                                 colour
., e77a  91 f3     sta ($f3),y   ;put it in colour RAM
., e77c  10 4d     bpl $e7cb
., e77e  a6 d4     ldx $d4       ;QTSW- editor in quotes mode?
., e780  f0 03     beq $e785     ;no
., e782  4c 97 e6  jmp $e697     ;output reversed character
., e785  c9 12     cmp #$12      ;<RVS>?
., e787  d0 02     bne $e78b     ;no
., e789  85 c7     sta $c7       ;RVS - print reversed
                                 characters flag
., e78b  c9 13     cmp #$13      ;<HOME>?
., e78d  d0 03     bne $e792     ;no
., e78f  20 66 e5  jsr $e566     ;home cursor
., e792  c9 1d     cmp #$1d      ;<CRSR RIGHT>?
., e794  d0 17     bne $e7ad     ;no
., e796  c8        iny
., e797  20 b3 e8  jsr $e8b3     ;check line increment
```

```
., e79a  84 d3    sty $d3       ;PNTR
., e79c  88       dey
., e79d  c4 d5    cpy $d5       ;LNMX
., e79f  90 09    bcc $e7aa
., e7a1  c6 d6    dec $d6       ;TBLX - current physical line
                                number
., e7a3  20 7c e8 jsr $e87c     ;go to next line
., e7a6  a0 00    ldy #$00
., e7a8  84 d3    sty $d3       ;PNTR
., e7aa  4c a8 e6 jmp $e6a8     ;finish screen print
., e7ad  c9 11    cmp #$11      ;<CRSR DOWN>?
., e7af  d0 1d    bne $e7ce     ;no
., e7b1  18       clc
., e7b2  98       tya           ;(Y) holds cursor column
., e7b3  69 28    adc #$28
., e7b5  a8       tay
., e7b6  e6 d6    inc $d6       ;TBLX
., e7b8  c5 d5    cmp $d5       ;LNMX
., e7ba  90 ec    bcc $e7a8     ;finish screen print
., e7bc  f0 ea    beq $e7a8
., e7be  c6 d6    dec $d6       ;TBLX
., e7c0  e9 28    sbc #$28
., e7c2  90 04    bcc $e7c8
., e7c4  85 d3    sta $d3       ;PNTR
., e7c6  d0 f8    bne $e7c0
., e7c8  20 7c e8 jsr $e87c     ;go to next line
., e7cb  4c a8 e6 jmp $e6a8     ;finish screen print
., e7ce  20 cb e8 jsr $e8cb     ;set colour code
., e7d1  4c 44 ec jmp $ec44     do graphics/text control
```

SHIFTED CHARACTERS. These are dealt with in the following order: Shifted ordinary ASCII and PET graphics characters, <shift RETURN>, <INST>, <CRSR UP>, <RVS OFF>, <CRSR LEFT>, <CLR>. If either insert mode is on, or quotes are open then the control character is not processed but a reversed ASCII literal is printed.

```
., e7d4  29 7f    and #$7f
., e7d6  c9 7f    cmp #$7f
., e7d8  d0 02    bne $e7dc
., e7da  a9 5e    lda #$5e
., e7dc  c9 20    cmp #$20
., e7de  90 03    bcc $e7e3
., e7e0  4c 91 e6 jmp $e691     ;set up screen print
., e7e3  c9 0d    cmp #$0d      ;<shift RETURN>?
., e7e5  d0 03    bne $e7ea     ;no
., e7e7  4c 91 e8 jmp $e891     ;do <RETURN>
., e7ea  a6 d4    ldx $d4       ;QTSW
., e7ec  d0 3f    bne $e82d     ;editor is in quotes mode
```

```
., e7ee  c9 14     cmp #$14     ;<INST>?
., e7f0  d0 37     bne $e829    ;no
., e7f2  a4 d5     ldy $d5      ;LNMX
., e7f4  b1 d1     lda ($d1),y  ;get screen character
., e7f6  c9 20     cmp #$20     ;space?
., e7f8  d0 04     bne $e7fe    ;no
., e7fa  c4 d3     cpy $d3      ;PNTR
., e7fc  d0 07     bne $e805
., e7fe  c0 4f     cpy #$4f     ;79?
., e800  f0 24     beq $e826    ;end of logical line - can't
                                insert
., e802  20 65 e9  jsr $e965    ;open space on screen
., e805  a4 d5     ldy $d5      ;LNMX
., e807  20 24 ea  jsr $ea24    ;synchronise colour pointer
., e80a  88        dey
., e80b  b1 d1     lda ($d1),y  ;move character right 1
., e80d  c8        iny
., e80e  91 d1     sta ($d1),y
., e810  88        dey
., e811  b1 f3     lda ($f3),y  ;move screen colour right 1
., e813  c8        iny
., e814  91 f3     sta ($f3),y
., e816  88        dey
., e817  c4 d3     cpy $d3      ;PNTR
., e819  d0 ef     bne $e80a    ;move next character along
., e81b  a9 20     lda #$20
., e81d  91 d1     sta ($d1),y
., e81f  ad 86 02  lda $0286
., e822  91 f3     sta ($f3),y
., e824  e6 d8     inc $d8
., e826  4c a8 e6  jmp $e6a8    ;finish screen print
., e829  a6 d8     ldx $d8      ;INSRT
., e82b  f0 05     beq $e832    ;insert mode is off
., e82d  09 40     ora #$40
., e82f  4c 97 e6  jmp $e697    ;set up screen print
., e832  c9 11     cmp #$11     ;<CRSR UP>?
., e834  d0 16     bne $e84c    ;no
., e836  a6 d6     ldx $d6      ;TBLX
., e838  f0 37     beq $e871    ;top line - do nothing
., e83a  c6 d6     dec $d6      ;TBLX
., e83c  a5 d3     lda $d3      ;PNTR
., e83e  38        sec
., e83f  e9 28     sbc #$28     ;back 40 columns for double
                                lines
., e841  90 04     bcc $e847
., e843  85 d3     sta $d3      ;PNTR
., e845  10 2a     bpl $e871    ;finish screen print
., e847  20 6c e5  jsr $e56c    ;set screen pointers
., e84a  d0 25     bne $e871    ;finish screen print
```

```
.,  e84c  c9 12     cmp #$12     ;<RVS OFF>?
.,  e84e  d0 04     bne $e854    ;no
.,  e850  a9 00     lda #$00
.,  e852  85 c7     sta $c7      ;RVS - disable reverse print
.,  e854  c9 1d     cmp #$1d     ;<CRSR LEFT>?
.,  e856  d0 12     bne $e86a    ;no
.,  e858  98        tya
.,  e859  f0 09     beq $e864
.,  e85b  20 a1 e8  jsr $e8a1    ;check line decrement
.,  e85e  88        dey
.,  e85f  84 d3     sty $d3      ;PNTR
.,  e861  4c a8 e6  jmp $e6a8    ;finish screen print
.,  e864  20 01 e7  jsr $e701    ;back on to previous line
.,  e867  4c a8 e6  jmp $e6a8    ;finish screen print
.,  e86a  c9 13     cmp #$13     ;<CLR>?
.,  e86c  d0 06     bne $e874    ;no
.,  e86e  20 44 e5  jsr $e544    ;clear screen
.,  e871  4c a8 e6  jmp $e6a8    ;finish screen print
.,  e874  09 80     ora #$80
.,  e876  20 cb e8  jsr $e8cb    ;set colour code
.,  e879  4c 4f ec  jmp $ec4f    ;set graphics/text mode
```

59516 GO TO NEXT LINE

The cursor is placed at the start of the next logical screen line. This involves moving down two lines for a linked line. If this places the cursor below the bottom of the screen, then the screen is scrolled.

```
.,  e87c  46 c9     lsr $c9      ;LXSP - cursor X-Y position
                                 at start
.,  e87e  a6 d6     ldx $d6      ;TBLX - current line number
.,  e880  e8        inx
.,  e881  e0 19     cpx #$19     ;26th line?
.,  e883  d0 03     bne $e888    ;no - scroll is not needed
.,  e885  20 ea e8  jsr $e8ea    ;scroll screen
.,  e888  b5 d9     lda $d9,x    ;LDTB1 - screen line link
                                 table
.,  e88a  10 f4     bpl $e880
.,  e88c  86 d6     stx $d6      ;TBLX
.,  e88e  4c 6c e5  jmp $e56c    ;set screen pointers
```

59537 OUTPUT <CARRIAGE RETURN>

All the editor modes are switched off and the cursor placed at the start of the next line.

```
.,  e891  a2 00     ldx #$00
.,  e893  86 d8     stx $d8      ;INSRT - disable insert mode
```

```
., e895  86 c7     stx $c7      ;RVS - disable reverse field
                                print
., e897  86 d4     stx $d4      ;QTSW - disable quotes mode
., e899  86 d3     stx $d3      ;PNTR - put cursor at 1st
                                column
., e89b  20 7c e8  jsr $e87c    ;go to next line
., e89e  4c a8 e6  jmp $e6a8    ;finish screen print
```

59553 CHECK LINE DECREMENT

When the cursor is at the beginning of a screen line, if it is moved backwards, this routine places it at the end of the line above.

```
., e8a1  a2 02     ldx #$02
., e8a3  a9 00     lda #$00
., e8a5  c5 d3     cmp $d3      ;PNTR - cursor column on
                                current line
., e8a7  f0 07     beq $e8b0    ;start of line
., e8a9  18        clc
., e8aa  69 28     adc #$28     ;add 40 to column
., e8ac  ca        dex
., e8ad  d0 f6     bne $e8a5
., e9af  60        rts
., e8b0  c6 d6     dec $d6      ;TBLX - current line number
., e8b2  60        rts
```

59571 CHECK LINE INCREMENT

When the cursor is at the end of a screen line, if it is moved forwards, this routine places it at the start of the line below.

```
., e8b3  a2 02     ldx #$02
., e8b5  a9 27     lda #$27
., e8b7  c5 d3     cmp $d3      ;PNTR - cursor column on
                                current line
., e8b9  f0 07     beq $e8c2
., e8bb  18        clc
., e8bc  69 28     adc #$28
., e8be  ca        dex
., e8bf  d0 f6     bne $e8b7
., e8c1  60        rts
., e8c2  a6 d6     ldx $d6      ;TBLX - current line number
., e8c4  e0 19     cpx #$19     ;25?
., e8c6  f0 02     beq $e8ca    ;yes
., e8c8  e6 d6     inc $d6      ;TBLX
., e8ca  60        rts
```

59595 SET COLOUR CODE

This routine is called by the output to screen routine. The ASCII code in (A) is compared with the ASCII colour codes table. If a match is found, then the table offset (and hence the colour value) is stored in COLOR

```
., e8cb  a2 0f     ldx #$0f
., e8cd  dd da e8  cmp $e8da,x   ;colour code table
., e8d0  f0 04     beq $e8d6     ;set colour code
., e8d2  ca        dex
., e8d3  10 f8     bpl $e8cd     ;try next value in table
., e8d5  60        rts
., e8d6  8e 86 02  stx $0286     ;COLOR - current character
                                 colour
., e8d9  60        rts
```

59610 COLOUR CODE TABLE

This is a table of 16 CBM ASCII codes representing the 16 available colours. Thus red is represented as #1c in the table (in the third position) and would be obtained by PRINT CHR$(28) or POKE 646,2.

```
.:e8da 90 05 1c 9f 9c 1e 1f 9e
.:e8e2 81 95 96 97 98 99 9a 9b
```

59626 SCROLL SCREEN

This routine scrolls the screen down by one line. If the top two lines are linked together then the scroll down is repeated. The screen line link pointers are updated, each screen line is cleared and the line below is moved up into it. The keyboard is directly read from CIA #1 by setting the keyboard row in port A, and reading the column from port B. If <CTRL> was pressed, then there is a 0.5 second delay before the routine exits.

```
., e8ea  a5 ac     lda $ac       ;push SAL - scrolling pointer
., e8ec  48        pha
., e8ed  a5 ad     lda $ad
., e8ef  48        pha
., e8f0  a5 ae     lda $ae       ;push EAL - end of program
., e8f2  48        pha
., e8f3  a5 af     lda $af
., e8f5  48        pha
., e8f6  a2 ff     ldx #$ff
., e8f8  c6 d6     dec $d6       ;TBLX - current line number
., e8fa  c6 c9     dec $c9       ;LXSP - cursor X-Y position
```

```
                                         at start
., e8fc  ce a5 02 dec $02a5     ;temp for line index
., e8ff  e8       inx
., e900  20 f0 e9 jsr $e9f0     ;set start of line
., e903  e0 18    cpx #$18
., e905  b0 0c    bcs $e913
., e907  bd f1 ec lda $ecf1,x
., e90a  85 ac    sta $ac
., e90c  b5 da    lda $da,x
., e90e  20 c8 e9 jsr $e9c8     ;move a screen line
., e911  30 ec    bmi $e8ff
., e913  20 ff e9 jsr $e9ff     ;clear a screen line
., e916  a2 00    ldx #$00
., e918  b5 d9    lda $d9,x     ;LDTB1 - screen line link
                                 table
., e91a  29 7f    and #$7f
., e91c  b4 da    ldy $da,x
., e91e  10 02    bpl $e922
., e920  09 80    ora #$80
., e922  95 d9    sta $d9,x
., e924  e8       inx
., e925  e0 18    cpx #$18
., e927  d0 ef    bne $e918
., e929  a5 f1    lda $f1       ;bottom line link
., e92b  09 80    ora #$80      ;unlink it
., e92d  85 f1    sta $f1
., e92f  a5 d9    lda $d9       ;top line link
., e931  10 c3    bpl $e8f6     ;line is linked - scroll
                                 again
., e933  e6 d6    inc $d6       ;TBLX
., e935  ee a5 02 inc $02a5     ;temp for line index
., e938  a9 7f    lda #$7f
., e93a  8d 00 dc sta $dc00     ;set keyboard decode row
., e93d  ad 01 dc lda $dc01     ;read keyboard decode column
., e940  c9 fb    cmp #$fb      ;<CTRL>?
., e942  08       php
., e943  a9 7f    lda #$7f
., e945  8d 00 dc sta $dc00     ;store row in decode register
., e948  28       plp
., e949  d0 0b    bne $e956     ;key was not pressed
., e94b  a0 00    ldy #$00      ;half second delay loop
., e94d  ea       nop
., e94e  ca       dex
., e94f  d0 fc    bne $e94d
., e951  88       dey
., e952  d0 f9    bne $e94d     ;end of delay loop
., e954  84 c6    sty $c6       ;NDX - # characters in keyb'd
                                 buffer
., e956  a6 d6    ldx $d6       ;TBLX
```

```
., e958  68        pla
., e959  85 af     sta $af        ;pull EAL
., e95b  68        pla
., e95c  85 ae     sta $ae
., e95e  68        pla
., e95f  85 ad     sta $ad        ;pull SAL
., e961  68        pla
., e962  85 ac     sta $ac
., e964  60        rts
```

59749 OPEN A SPACE ON THE SCREEN

This routine opens up a space on the screen for use with <INST>. If needed, the screen is then scrolled down, otherwise the screen line is moved and cleared. Finally the screen line link table is adjusted and updated.

```
., e965  a6 d6     ldx $d6        ;TBLX - current cursor line
                                   number
., e967  e8        inx
., e968  b5 d9     lda $d9,x      ;LDTB1 - screen line link
                                   table
., e96a  10 fb     bpl $e967
., e96c  8e a5 02  stx $02a5      ;temp line for index
., e96f  e0 18     cpx #$18       ;bottom of screen?
., e971  f0 0e     beq $e981      ;yes
., e973  90 0c     bcc $e981      ;above bottom line
., e975  20 ea e8  jsr $e8ea      ;scroll screen up
., e978  ae a5 02  ldx $02a5      ;temp line for index
., e97b  ca        dex
., e97c  c6 d6     dec $d6        ;TBLX
., e97e  4c da e6  jmp $e6da      ;adjust link table and end
., e981  a5 ac     lda $ac        ;push SAL - scrolling pointer
., e983  48        pha
., e984  a5 ad     lda $ad
., e986  48        pha
., e987  a5 ae     lda $ae        ;push EAL - end of program
., e989  48        pha
., e98a  a5 af     lda $af
., e98c  48        pha
., e98d  a2 19     ldx #$19
., e98f  ca        dex
., e990  20 f0 e9  jsr $e9f0      ;set start of line
., e993  ec a5 02  cpx $02a5      ;temp line for index
., e996  90 0e     bcc $e9a6
., e998  f0 0c     beq $e9a6
., e99a  bd ef ec  lda $ecef,x    ;screen line address table
., e99d  85 ac     sta $ac        ;SAL
., e99f  b5 d8     lda $d8,x      ;LDTB1
```

```
., e9a1  20 c8 e9 jsr $e9c8     ;move screen line
., e9a4  30 e9    bmi $e98f
., e9a6  20 ff e9 jsr $e9ff     ;clear screen line
., e9a9  a2 17    ldx #$17
., e9ab  ec a5 02 cpx $02a5     ;temp line for index
., e9ae  90 0f    bcc $e9bf
., e9b0  b5 da    lda $da,x     ;LDTB1
., e9b2  29 7f    and #$7f
., e9b4  b4 d9    ldy $d9,x
., e9b6  10 02    bpl $e9ba
., e9b8  09 80    ora #$80
., e9ba  95 da    sta $da,x
., e9bc  ca       dex
., e9bd  d0 ec    bne $e9ab
., e9bf  ae a5 02 ldx $02a5     ;temp line for index
., e9c2  20 da e6 jsr $e6da     ;adjust link table
., e9c5  4c 58 e9 jmp $e958     ;pull SAL and EAL
```

59848 MOVE A SCREEN LINE

This routine synchronises colour transfer, and then moves the screen line pointed to vertically character by character. The colour codes for each character are also moved in the same way.

```
., e9c8  29 03    and #$03
., e9ca  0d 88 02 ora $0288     ;HIBASE - top of screen page
., e9cd  85 ad    sta $ad       ;>SAL - screen scroll pointer
., e9cf  20 e0 e9 jsr $e9e0     ;synchronise colour transfer
., e9d2  a0 27    ldy #$27      ;offset for character on
                                screen line
., e9d4  b1 ac    lda ($ac),y   ;move screen character
., e9d6  91 d1    sta ($d1),y
., e9d8  b1 ae    lda ($ae),y   ;move character colour
., e9da  91 f3    sta ($f3),y
., e9dc  88       dey
., e9dd  10 f5    bpl $e9d4     ;next character on screen
                                line
., e9df  60       rts
```

59872 SYNCHRONISE COLOUR TRANSFER

This subroutine sets up a temporary pointer in EAL ($AE) to the colour RAM address that corresponds to the temporary screen address held in EAL ($AC).

```
., e9e0  20 24 ea jsr $ea24     ;synchronise colour pointer
., e9e3  a5 ac    lda $ac       ;SAL - pointer for screen
                                scroll
```

```
.,  e9e5  85 ae     sta $ae       ;EAL
.,  e9e7  a5 ad     lda $ad
.,  e9e9  29 03     and #$03
.,  e9eb  09 d8     ora #$d8
.,  e9ed  85 af     sta $af
.,  e9ef  60        rts
```

59888 SET START OF LINE

On entry, (X) holds the line number. The low byte of the address is set from the ROM table and the high byte derived from the screen link and HIBASE.

```
.,  e9f0  bd f0 ec lda $ecf0,x  ;table of screen line lo
                                 bytes
.,  e9f3  85 d1     sta $d1      ;<PNT - current screen line
                                 address
.,  e9f5  b5 d9     lda $d9,x    ;LDTB1 - screen line link
                                 table
.,  e9f7  29 03     and #$03
.,  e9f9  0d 88 02 ora $0288    ;HIBASE - page for top of
                                 screen
.,  e9fc  85 d2     sta $d2      ;>PNT
.,  e9fe  60        rts
```

59903 CLEAR SCREEN LINE

The start of line is set and the screen line is cleared by filling it with ASCII spaces. The corresponding line of colour RAM is also cleared to the value held in $D021.

```
.,  e9ff  a0 27     ldy #$27
.,  ea01  20 f0 e9 jsr $e9f0    ;set start of line
.,  ea04  20 24 ea jsr $ea24    ;synchronise colour pointer
.,  ea07  a9 20     lda #$20     ;ASCII space
.,  ea09  91 d1     sta ($d1),y  ;store character on screen
.,  ea0b  20 da e4 jsr $e4da    ;set character colour =
                                 screen colour
.,  ea0e  ea        nop
.,  ea0f  88        dey
.,  ea10  10 f5     bpl $ea07
.,  ea12  60        rts
```

59923 PRINT TO SCREEN

The colour pointer is synchronised, and the character in (A) directly stored in screen RAM. The character colour in (X) is stored at the equivalent point in colour RAM.

```
., ea13  a8        tay           ;put print character in (Y)
., ea14  a9 02     lda #$02
., ea16  85 cd     sta $cd       ;BLNCT - timer to toggle
                                 cursor
., ea18  20 24 ea jsr $ea24      ;synchronise colour pointer
., ea1b  98        tya           ;print character back in (A)
., ea1c  a4 d3     ldy $d3       ;PNTR - cursor column on line
., ea1e  91 d1     sta ($d1),y   ;store character on screen
., ea20  8a        txa
., ea21  91 f3     sta ($f3),y   ;store character colour
., ea23  60        rts
```

59940 SYNCHRONISE COLOUR POINTER

The pointer to the current byte of colour RAM is set to equal the current screen line address (modified to take into account the different start locations of the two areas of RAM).

```
., ea24  a5 d1     lda $d1       ;PNT - current screen line
                                 address
., ea26  85 f3     sta $f3       ;USER - current colour RAM
                                 location
., ea28  a5 d2     lda $d2
., ea2a  29 03     and #$03
., ea2c  09 d8     ora #$d8      ;modify hi byte to point to
                                 colour RAM
., ea2e  85 f4     sta $f4
., ea30  60        rts
```

59953 MAIN IRQ ENTRY POINT

This routine services an IRQ request unless the vector CINV ($0314) is altered or the interrupt is masked by the SEI instruction. The first function performed is to test the <STOP> key and update the real-time clock. This routine can be bypassed by advancing the IRQ vector by 3, resulting in the <STOP> key being disabled. The next function updates the cursor if BLNSW indicates that it is enabled. BLNCT is decremented. If this timer has reached zero, then the cursor is toggled. The final function of the service routine is to read the keyboard. (A), (X) and (Y) registers are restored on exit. See also $FF48.

```
., ea31  20 ea ff jsr $ffea      ;UDTIM - update real-time
                                 clock
., ea34  a5 cc     lda $cc       ;BLNSW - cursor blink enable
., ea36  d0 29     bne $ea61     ;cursor is off
., ea38  c6 cd     dec $cd       ;BLNCT - timer to toggle
```

```
                                       cursor
., ea3a  d0 25     bne $ea61
., ea3c  a9 14     lda #$14           ;cursor timeout - reset timer
., ea3e  85 cd     sta $cd
., ea40  a4 d3     ldy $d3            ;PNTR - cursor column
., ea42  46 cf     lsr $cf            ;BLNON - flag last cursor
                                       blink on/off
., ea44  ae 87 02 ldx $0287           ;GDCOL - background colour
                                       under cursor
., ea47  b1 d1     lda ($d1),y        ;get screen character
., ea49  b0 11     bcs $ea5c
., ea4b  e6 cf     inc $cf            ;BLNON
., ea4d  85 ce     sta $ce            ;GDBLN - character under
                                       cursor
., ea4f  20 24 ea jsr $ea24           ;synchronise colour pointer
., ea52  b1 f3     lda ($f3),y        ;get character colour
., ea54  8d 87 02 sta $0287           ;GDCOL
., ea57  ae 86 02 ldx $0286           ;COLOR - current character
                                       colour code
., ea5a  a5 ce     lda $ce            ;GDBBLN
., ea5c  49 80     eor #$80           ;toggle cursor
., ea5e  20 1c ea jsr $ea1c           ;print to screen
., ea61  a5 01     lda $01            ;R6510 - processor onboard
                                       I/O register
., ea63  29 10     and #$10
., ea65  f0 0a     beq $ea71
., ea67  a0 00     ldy #$00
., ea69  84 c0     sty $c0            ;CAS1 - tape motor interlock
., ea6b  a5 01     lda $01            ;R6510
., ea6d  09 20     ora #$20
., ea6f  d0 08     bne $ea79
., ea71  a5 c0     lda $c0            ;CAS1
., ea73  d0 06     bne $ea7b
., ea75  a5 01     lda $01
., ea77  29 1f     and #$1f
., ea79  85 01     sta $01
., ea7b  20 87 ea jsr $ea87           ;scan keyboard
., ea7e  ad 0d dc lda $dc0d           ;clear CIA #1 I.C.R.
., ea81  68        pla
., ea82  a8        tay
., ea83  68        pla
., ea84  aa        tax
., ea85  68        pla
., ea86  40        rti
```

60039 SCNKEY: SCAN KEYBOARD

The KERNAL routine SCNKEY ($FF9F) jumps to this routine. Firstly, SHFLAG is zeroed and the key image is set to 'no

key'. The keyboard is then scanned. The keyboard is set up in an 8 by 8 matrix and is read by writing a zero to the required row and reading the column in which a key was pressed. Therefore a column reading of #FF indicates that no key was pressed, #7F that a key in column 7 was pressed and so on. A small debounce loop is included to allow the key value to settle. A larger loop scans through each row until a keypress is found. The key image is then passed to the next section for decoding.

```
., ea87  a9 00     lda #$00
., ea89  8d 8d 02  sta $028d      ;SHFLAG - flag SHIFT/CTRL/CBM
., ea8c  a0 40     ldy #$40
., ea8e  84 cb     sty $cb        ;SFDX - print shifted
                                  characters
., ea90  8d 00 dc  sta $dc00      ;keyboard write register
., ea93  ae 01 dc  ldx $dc01      ;keyboard read register
., ea96  e0 ff     cpx #$ff       ;no key pressed?
., ea98  f0 61     beq $eafb      ;so end
., ea9a  a8        tay
., ea9b  a9 81     lda #$81
., ea9d  85 f5     sta $f5        ;KEYTAB - keyboard decode
                                  table vector
., ea9f  a9 eb     lda #$eb
., eaa1  85 f6     sta $f6        ;vector = $EB81
., eaa3  a9 fe     lda #$fe
., eaa5  8d 00 dc  sta $dc00      ;write keyboard row
., eaa8  a2 08     ldx #$08       ;counter to write 8 rows
., eaaa  48        pha
., eaab  ad 01 dc  lda $dc01      ;read keyboard column
., eaae  cd 01 dc  cmp $dc01
., eab1  d0 f8     bne $eaab      ;wait for value to settle
., eab3  4a        lsr
., eab4  b0 16     bcs $eacc
., eab6  48        pha
., eab7  b1 f5     lda ($f5),y    ;get value from decode table
., eab9  c9 05     cmp #$05
., eabb  b0 0c     bcs $eac9
., eabd  c9 03     cmp #$03
., eabf  f0 08     beq $eac9
., eac1  0d 8d 02  ora $028d      ;SHFLAG
., eac4  8d 8d 02  sta $028d
., eac7  10 02     bpl $eacb
., eac9  84 cb     sty $cb
., eacb  68        pla
., eacc  c8        iny
., eacd  c0 41     cpy #$41
., eacf  b0 0b     bcs $eadc
., ead1  ca        dex
```

```
., ead2  d0 df     bne $eab3
., ead4  38        sec
., ead5  68        pla
., ead6  2a        rol
., ead7  8d 00 dc  sta $dc00     ;keyboard write register
., eada  d0 cc     bne $eaa8
., eadc  68        pla
```

60125 PROCESS KEY IMAGE

The keypress is decoded into an ASCII value by use of 4 lookup tables. If the key pressed is the same as the last image generated by the previous interrupt, then the key repeat section is entered (if repeats are enabled on that particular key). The new key is stored in the keyboard buffer and its relevant pointers are updated. This does not occur if the buffer is already full.

```
., eadd  6c 8f 02  jmp ($028f)   ;vector KEYLOG - points to
                                 $EAE0
., eae0  a4 cb     ldy $cb       ;SFDX
., eae2  b1 f5     lda ($f5),y   ;get value from decode table
., eae4  aa        tax
., eae5  c4 c5     cpy $c5       ;LSTX - current key pressed
., eae7  f0 07     beq $eaf0     ;same key
., eae9  a0 10     ldy #$10
., eaeb  8c 8c 02  sty $028c     ;DELAY - repeat delay counter
., eaee  d0 36     bne $eb26
., eaf0  29 7f     and #$7f
., eaf2  2c 8a 02  bit $028a     ;RPTFLG - repeat key flag
., eaf5  30 16     bmi $eb0d     ;repeat all keys
., eaf7  70 49     bvs $eb42     ;repeat none - end
., eaf9  c9 7f     cmp #$7f
., eafb  f0 29     beq $eb26
., eafd  c9 14     cmp #$14      ;<DEL>?
., eaff  f0 0c     beq $eb0d
., eb01  c9 20     cmp #$20      ;<space>?
., eb03  f0 08     beq $eb0d
., eb05  c9 1d     cmp #$1d      ;<CRSR RIGHT/LEFT>?
., eb07  f0 04     beq $eb0d
., eb09  c9 11     cmp #$11      ;<CRSR DOWN/UP>?
., eb0b  d0 35     bne $eb42     ;end
., eb0d  ac 8c 02  ldy $028c     ;DELAY
., eb10  f0 05     beq $eb17
., eb12  ce 8c 02  dec $028c     ;DELAY
., eb15  d0 2b     bne $eb42     ;end
., eb17  ce 8b 02  dec $028b     ;KOUNT - repeat speed counter
., eb1a  d0 26     bne $eb42     ;end
., eb1c  a0 04     ldy #$04
```

```
.,  eb1e   8c 8b 02 sty $028b    ;KOUNT
.,  eb21   a4 c6    ldy $c6      ;NDX - # keys in keyboard
                                 queue
.,  eb23   88       dey
.,  eb24   10 1c    bpl $eb42    ;end
.,  eb26   a4 cb    ldy $cb      ;SFDX
.,  eb28   84 c5    sty $c5      ;LSTX
.,  eb2a   ac 8d 02 ldy $028d    ;SHFLAG
.,  eb2d   8c 8e 02 sty $028e    ;LSTSHF - last keyboard shift
                                 pattern
.,  eb30   e0 ff    cpx #$ff     ;no key pressed
.,  eb32   f0 0e    beq $eb42    ;end
.,  eb34   8a       txa
.,  eb35   a6 c6    ldx $c6      ;NDX
.,  eb37   ec 89 02 cpx $0289    ;XMAX - size of keyboard
                                 buffer
.,  eb3a   b0 06    bcs $eb42
.,  eb3c   9d 77 02 sta $0277,x  ;KEYD - keyboard buffer queue
.,  eb3f   e8       inx
.,  eb40   86 c6    stx $c6      ;NDX
.,  eb42   a9 7f    lda #$7f
.,  eb44   8d 00 dc sta $dc00    ;keyboard write register
.,  eb47   60       rts
.,  eb48   ad 8d 02 lda $028d    ;SHFLAG
.,  eb4b   c9 03    cmp #$03
.,  eb4d   d0 15    bne $eb64
.,  eb4f   cd 8e 02 cmp $028e    ;LSTSHF
.,  eb52   f0 ee    beq $eb42    ;same - end
.,  eb54   ad 91 02 lda $0291    ;MODE - shift key enable flag
.,  eb57   30 1d    bmi $eb76    ;keys enabled - process key
                                 image
.,  eb59   ad 18 d0 lda $d018    ;VIC II memory control
                                 register
.,  eb5c   49 02    eor #$02     ;toggle character set
.,  eb5e   8d 18 d0 sta $d018
.,  eb61   4c 76 eb jmp $eb76    ;process key image
.,  eb64   0a       asl
.,  eb65   c9 08    cmp #$08
.,  eb67   90 02    bcc $eb6b
.,  eb69   a9 06    lda #$06
.,  eb6b   aa       tax
.,  eb6c   bd 79 eb lda $eb79,x  ;keyboard table select
                                 vectors
.,  eb6f   85 f5    sta $f5      ;KEYTAB - decode table vector
.,  eb71   bd 7a eb lda $eb7a,x
.,  eb74   85 f6    sta $f6
.,  eb76   4c e0 ea jmp $eae0    ;process key image
```

60281 KEYBOARD SELECT VECTORS

This is a table of vectors pointing to the start of the four keyboard decode tables.

```
.:eb79 81 eb c2 eb 03 ec 78 ec
```

60289 KEYBOARD 1 - UNSHIFTED

This is the first of four keyboard decode tables. The ASCII code for the key pressed is at the intersection of the row (written to $DC00) and column (read from $DC01). The matrix values are as shown:

```
.:eb81 14 0d 1d 88 85 86 87 11
.:eb89 33 57 41 34 5a 53 45 01
.:eb91 35 52 44 36 43 46 54 58
.:eb99 37 59 47 38 42 48 55 56
.:eba1 39 49 4a 30 4d 4b 4f 4e
.:eba9 2b 50 4c 2d 2e 3a 40 2c
.:ebb1 5c 2a 3b 13 01 3d 5e 2f
.:ebb9 31 5f 04 32 20 02 51 03
.:ebc1 ff
```

READ FROM $DC01

WRITE TO $DC00	#FE	#FD	#FB	#F7	#EF	#DF	#8F	#7F
#FE	DEL	RETURN	CRSR RIGHT	F7	F1	F3	F5	CRSR DOWN
#FD	3	w	a	4	z	s	e	LEFT SHIFT
#FB	5	r	d	6	c	f	t	x
#F7	7	y	g	8	b	h	u	v
#EF	9	i	j	Ø	m	k	o	n
#DF	+	p	l	—	.	:	@	,
#8F	£	*	;	HOME	RIGHT SHIFT	=	↑	/
#7F	1	←	CTRL	2	SPACE	CBM	q	STOP

60354 KEYBOARD 2 - SHIFTED

This is the second of four keyboard decode tables. The ASCII code for the key pressed is at the intersection of the row (written to $DC00) and column (read from $DC01). The matrix values are as shown:

```
.:ebc2 94 8d 9d 8c 89 8a 8b 91
.:ebca 23 d7 c1 24 da d3 c5 01
.:ebd2 25 d2 c4 26 c3 c6 d4 d8
.:ebda 27 d9 c7 28 c2 c8 d5 d6
.:ebe2 29 c9 ca 30 cd cb cf ce
.:ebea db d0 cc dd 3e 5b ba 3c
.:ebf2 a9 c0 5d 93 01 3d de 3f
.:ebfa 21 5f 04 22 a0 02 d1 83
.:ec02 ff
```

READ FROM $DC01

WRITE TO $DC00

	#FE	#FD	#FB	#F7	#EF	#DF	#BF	#7F
#FE	INST	RETURN	CRSR LEFT	F8	F2	F4	F6	CRSR UP
#FD	#	W	A	$	Z	S	E	LEFT SHIFT
#FB	%	R	D	&	C	F	T	X
#F7	'	Y	G	(	B	H	U	V
#EF	)	I	J	0	M	K	O	N
#DF	⊞	P	L	◫	>	[	❏	<
#BF	◩	⊟	]	CLR	RIGHT SHIFT	=	π	?
#7F	!	←	CTRL	"	SPACE	CBM	Q	RUN

60419 KEYBOARD 3 - COMMODORE

This is the third of four keyboard decode tables. The ASCII code for the key pressed is at the intersection of the row (written to $DC00) and column (read from $DC01). The matrix values are as shown:

```
.:ec03 94 8d 9d 8c 89 8a 8b 91
.:ec0b 96 b3 b0 97 ad ae b1 01
.:ec13 98 b2 ac 99 bc bb a3 bd
.:ec1b 9a b7 a5 9b bf b4 b8 be
.:ec23 29 a2 b5 30 a7 a1 b9 aa
.:ec2b a6 af b6 dc 3e 5b a4 3c
.:ec33 a8 df 5d 93 01 3d de 3f
.:ec3b 81 5f 04 95 a0 02 ab 83
.:ec43 ff
```

READ FROM $DC01

WRITE TO $DC00

	#FE	#FD	#FB	#F7	#EF	#DF	#BF	#7F
#FE	INST	RETURN	CRSR LEFT	F8	F2	F4	F6	CRSR UP
#FD	PINK			GREY 1				LEFT SHIFT
#FB	GREY 2			LIGHT GREEN				
#F7	LIGHT BLUE			GREY3				
#EF	)			Ø				
#DF					>	[		<
#BF			]	CLR	RIGHT SHIFT	=	π	?
#7F	ORANGE	←	CTRL	BROWN	SPACE	CBM		RUN

60484 GRAPHICS / TEXT CONTROL

This routine is used to toggle between text and graphics character sets, and to enable/disable the <shift-CBM> keys. The routine is called by the main 'output to screen' routine, and (A) holds a CBM ASCII code on entry.

```
., ec44  c9 0e     cmp #$0e    ;<switch to lower case>?
., ec46  d0 07     bne $ec4f   ;no
., ec48  ad 18 d0  lda $d018   ;VIC memory control register
., ec4b  09 02     ora #$02    ;set lower case character ROM
., ec4d  d0 09     bne $ec58
., ec4f  c9 8e     cmp #$8e    ;<switch to upper case>?
., ec51  d0 0b     bne $ec5e   ;no
., ec53  ad 18 d0  lda $d018   ;VIC memory control register
```

```
., ec56  29 fd     and #$fd      ;set upper case character ROM
., ec58  8d 18 d0  sta $d018
., ec5b  4c a8 e6  jmp $e6a8     ;finish screen print
., ec5e  c9 08     cmp #$08      ;<disable <shift-CBM>>?
., ec60  d0 07     bne $ec69     ;no
., ec62  a9 80     lda #$80
., ec64  0d 91 02  ora $0291     ;MODE - disable shift keys
., ec67  30 09     bmi $ec72
., ec69  c9 09     cmp #$09      ;<enable <shift-CBM>>?
., ec6b  d0 ee     bne $ec5b     ;no - finish screen print
., ec6d  a9 7f     lda #$7f
., ec6f  2d 91 02  and $0291     ;MODE
., ec72  8d 91 02  sta $0291
., ec75  4c a8 e6  jmp $e6a8     ;finish screen print
```

60536 KEYBOARD 4 - CONTROL

This is the fourth of four keyboard decode tables. The ASCII code for the key pressed is at the intersection of the row (written to $DC00) and column (read from $DC01). The matrix values are as shown:

```
.:ec78 ff ff ff ff ff ff ff ff
.:ec80 1c 17 01 9f 1a 13 05 ff
.:ec88 9c 12 04 1e 03 06 14 18
.:ec90 1f 19 07 9e 02 08 15 16
.:ec98 12 09 0a 92 0d 0b 0f 0e
.:eca0 ff 10 0c ff ff 1b 00 ff
.:eca8 1c ff 1d ff ff 1f 1e ff
.:ecb0 90 06 ff 05 ff ff 11 ff
.:ecb8 ff
```

READ FROM $DC01

WRITE TO $DC00

	#FE	#FD	#FB	#F7	#EF	#DF	#BF	#7F
#FE								
#FD	RED	W	A	CYAN	Z	HOME	WHITE	
#FB	PURPLE	RVS ON	CTRL	GREEN	STOP	F	DEL	X
#F7	BLUE	Y	G	YELLOW	CBM	DISABLE	U	V
#EF	RVS ON	ENABLE	J	RVS OFF	RETURN	K	O	LOWER
#DF		P	L			]	@	
#BF	RED		CRSR RIGHT			BLUE	GREEN	
#7F	BLACK	←		WHITE			CRSR DOWN	

60601 VIDEO CHIP SET UP TABLE

This is a table of the initial values for the VIC II chip I/O registers.

```
.:ecb9 00 00 00 00 00 00 00 00
.:ecc1 00 00 00 00 00 00 00 00
.:ecc9 00 9b 37 00 00 00 08 00
.:ecd1 14 0f 00 00 00 00 00 00
.:ecd9 0e 06 01 02 03 04 00 01
.:ece1 02 03 04 05 06 07
```

60647 SHIFT-RUN EQUIVALENT

This is the message LOAD <CR> RUN <CR> which is placed in the keyboard buffer when <shift-RUN> is pressed.

```
.:ece7 4c 4f 41 44 0d 52 55 4e
.:ecef 0d
```

60656 LOW BYTE SCREEN LINE ADDRESSES

This is a table of the low bytes of screen line addresses. The high byte of the address is obtained by derivation from the page on which the screen starts. There was an additional table of high byte addresses on fixed screen model PETs.

```
.:ecf0 00 28 50 78 a0 c8 f0 18
.:ecf8 40 68 90 b8 e0 08 30 58
.:ed00 80 a8 d0 f8 20 48 70 98
.:ed08 c0
```

60681 TALK: SEND 'TALK'/'LISTEN'

The KERNAL routines TALK ($FFB4) and LISTEN ($FFB1) are vectored here. The routine sends the command 'TALK' or 'LISTEN' on the serial bus. On entry (A) must hold the device number to which the command will be sent. The two entry points differ only in that to TALK, (A) is ORed with #40, and to LISTEN, (A) is ORed with #20. The UNTALK (#3F) and UNLISTEN (#5F) commands are also sent via this routine, but their values are set on entry. If there is a character waiting to go out on the bus, then this is output. Handshaking is performed, and ATN (ATteNtion) is set low so that the byte is interpreted as a command. The routine drops through to the next one to output the byte on the serial bus. Note that on conclusion, ATN must be set high.

```
., ed09  09 40    ora #$40      ;set 'TALK' flag
., ed0b  2c 09 20 bit $2009     ;mask - ORA #$20 - set
                                'LISTEN' flag
., ed0e  20 a4 f0 jsr $f0a4     ;check serial bus idle
., ed11  48       pha
., ed12  24 94    bit $94       ;C3PO - character in serial
                                buffer?
., ed14  10 0a    bpl $ed20     ;no
., ed16  38       sec
., ed17  66 a3    ror $a3       ;temp data area
., ed19  20 40 ed jsr $ed40     ;send data to serial bus
., ed1c  46 94    lsr $94       ;C3PO
., ed1e  46 a3    lsr $a3
., ed20  68       pla
., ed21  85 95    sta $95       ;BSOUR - buffered character
                                for bus
., ed23  78       sei
., ed24  20 97 ee jsr $ee97     ;set data 1
., ed27  c9 3f    cmp #$3f      ;UNTALK?
., ed29  d0 03    bne $ed2e     ;no
., ed2b  20 85 ee jsr $ee85     ;set CLK 1
., ed2e  ad 00 dd lda $dd00     ;serial bus I/O port
., ed31  09 08    ora #$08      ;clear ATN - prepare for
                                command
., ed33  8d 00 dd sta $dd00
., ed36  78       sei
., ed37  20 8e ee jsr $ee8e     ;set CLK 0
., ed3a  20 97 ee jsr $ee97     ;set data 1
., ed3d  20 b3 ee jsr $eeb3     ;delay 1 mS
```

60736 SEND DATA ON SERIAL BUS

The byte of data to be output on the serial bus must have been previously stored in the serial buffer, BSOUR. An initial test is made for bus activity, and if none is detected then ST is set to #80, ie. ?DEVICE NOT PRESENT. The byte is output by rotating it right and sending the state of the carry flag. This is done 8 times until the whole byte has been sent. The CIA timer is then set to 65 milliseconds and the bus is checked for 'data accepted'. If timeout occurs before this happens then ST is set to #03, ie Write Timeout.

```
., ed40  78       sei
., ed41  20 97 ee jsr $ee97     ;set data 1
., ed44  20 a9 ee jsr $eea9     ;get serial in and clock
., ed47  b0 64    bcs $edad     ;no activity - device not
                                present
., ed49  20 85 ee jsr $ee85     ;set CLK 1
```

```
., ed4c  24 a3     bit $a3       ;temp data area
., ed4e  10 0a     bpl $ed5a
., ed50  20 a9 ee  jsr $eea9     ;get serial in and clock
., ed53  90 fb     bcc $ed50
., ed55  20 a9 ee  jsr $eea9     ;get serial in and clock
., ed58  b0 fb     bcs $ed55
., ed5a  20 a9 ee  jsr $eea9     ;get serial in and clock
., ed5d  90 fb     bcc $ed5a
., ed5f  20 8e ee  jsr $ee8e     ;set CLK 0
., ed62  a9 08     lda #$08
., ed64  85 a5     sta $a5       ;CNTDN - counter for O/P byte
., ed66  ad 00 dd  lda $dd00     ;serial bus I/O port
., ed69  cd 00 dd  cmp $dd00
., ed6c  d0 f8     bne $ed66     ;wait for port to settle
., ed6e  0a        asl
., ed6f  90 3f     bcc $edb0     ;set timeout error
., ed71  66 95     ror $95       ;BSOUR - character for serial
                                 bus
., ed73  b0 05     bcs $ed7a     ;write 1 to bus
., ed75  20 a0 ee  jsr $eea0     ;set data 0
., ed78  d0 03     bne $ed7d
., ed7a  20 97 ee  jsr $ee97     ;set data 1
., ed7d  20 85 ee  jsr $ee85     ;set CLK 1
., ed80  ea        nop
., ed81  ea        nop
., ed82  ea        nop
., ed83  ea        nop
., ed84  ad 00 dd  lda $dd00     ;serial bus I/O port
., ed87  29 df     and #$df      ;set data 1
., ed89  09 10     ora #$10      ;set CLK 0
., ed8b  8d 00 dd  sta $dd00
., ed8e  c6 a5     dec $a5       ;CNTDN
., ed90  d0 d4     bne $ed66     ;output next bit in byte
., ed92  a9 04     lda #$04
., ed94  8d 07 dc  sta $dc07     ;CIA timer A high byte
., ed97  a9 19     lda #$19
., ed99  8d 0f dc  sta $dc0f     ;set 1 shot & start timer
., ed9c  ad 0d dc  lda $dc0d     ;CIA ICR
., ed9f  ad 0d dc  lda $dc0d
., eda2  29 02     and #$02      ;timeout?
., eda4  d0 0a     bne $edb0     ;yes
., eda6  20 a9 ee  jsr $eea9     ;get serial in and clock
., eda9  b0 f4     bcs $ed9f
., edab  58        cli
., edac  60        rts
```

60845 FLAG ERRORS

(A) is loaded with one of two error flags, depending on the

entry point. #80 signifies the device was not present, and #03 signifies a write timeout. The value is then set into the I/O status word, ST. The routine exits by clearing ATN and giving the final handshake.

```
., edad  a9 80     lda #$80     ;flag ?DEVICE NOT PRESENT
., edaf  2c a9 03 bit $03a9    ;mask - flag write timeout
., edb2  20 1c fe jsr $fe1c    ;set I/O status word
., edb5  58        cli
., edb6  18        clc
., edb7  90 4a     bcc $ee03    ;do final handshake
```

60857 SECOND: SEND LISTEN SA

The KERNAL routine SECOND ($FF93) is vectored here. On entry, (A) holds the secondary address. This is placed in the serial buffer and sent to the serial bus "under attention". Finally the routine drops through to the next routine to set ATN false.

```
., edb9  85 95     sta $95      ;BSOUT - character for serial
                                bus
., edbb  20 36 ed jsr $ed36    ;handshake and send byte
```

60862 CLEAR ATN

The ATN (ATteNtion) line on the serial bus is set to 1, ie. ATN is now false and data sent on the bus will not be interpreted as a command.

```
., edbe  ad 00 dd lda $dd00    ;serial bus I/O port
., edc1  29 f7     and #$f7     ;set ATN 1
., edc3  8d 00 dd sta $dd00
., edc6  60        rts
```

60871 TKSA: SEND TALK SA

The KERNAL routine TKSA ($FF96) is vectored here. On entry, (A) holds the secondary address. This is placed in the serial buffer and sent out to the serial bus "under attention". The routine drops through to the next routine to wait for CLK and clear ATN.

```
., edc7  85 95     sta $95      ;BSOUR - character for serial
                                bus
., edc9  20 36 ed jsr $ed36    ;handshake and send byte to
                                bus
```

60876 WAIT FOR CLOCK

This routine sets data = 0, ATN = 1 and CLK = 1. It then waits to receive CLK = 0 from the serial bus.

```
., edcc  78        sei
., edcd  20 a0 ee  jsr $eea0     ;set data 0
., edd0  20 be ed  jsr $edbe     ;set ATN 1
., edd3  20 85 ee  jsr $ee85     ;set CLK 1
., edd6  20 a9 ee  jsr $eea9     ;get serial in and clock
., edd9  30 fb     bmi $edd6
., eddb  58        cli
., eddc  60        rts
```

60893 CIOUT: SEND SERIAL DEFERRED

The KERNAL routine CIOUT ($FFA8) jumps to this routine. If there is a character awaiting output in the buffer, then it is sent on the bus. The output flag, C3PO is set (ie. bit 7 = 1) and the contents of (A) placed in the serial buffer.

```
., eddd  24 94     bit $94       ;C3PO - character in serial
                                 buffer?
., eddf  30 05     bmi $ede6     ;yes
., ede1  38        sec
., ede2  66 94     ror $94       ;C3PO - set flag to send
                                 character
., ede4  d0 05     bne $edeb
., ede6  48        pha
., ede7  20 40 ed  jsr $ed40     ;send data to serial bus
., edea  68        pla
., edeb  85 95     sta $95       ;BSOUR - buffered character
                                 for bus
., eded  18        clc
., edee  60        rts
```

60911 UNTLK: SEND 'UNTALK'/'UNLISTEN'

The KERNAL routines UNTALK ($FFAB) and UNLSTN ($FFAE) are vectored here. CLK is set to 0 and ATN set to 0. (A) is loaded with #5F for 'UNTALK' and #3F for 'UNLISTEN'. The command is then sent to the serial bus via the 'send TALK/LISTEN' routine. Finally ATN is set to 1, and after a short delay, CLK and DATA are both set to 1.

```
., edef  78        sei
., edf0  20 8e ee  jsr $ee8e     ;set CLK 0
., edf3  ad 00 dd  lda $dd00     ;serial bus I/O port
., edf6  09 08     ora #$08      ;set ATN 0
., edf8  8d 00 dd  sta $dd00
```

```
., edfb  a9 5f     lda #$5f      ;flag UNTALK
., edfd  2c a9 3f bit $3fa9      ;mask - LDA #$3F - flag
                                 UNLISTEN
., ee00  20 11 ed jsr $ed11      ;send command to serial bus
., ee03  20 be ed jsr $edbe      ;clear ATN
., ee06  8a       txa
., ee07  a2 0a    ldx #$0a       ;do delay
., ee09  ca       dex
., ee0a  d0 fd    bne $ee09
., ee0c  aa       tax
., ee0d  20 85 ee jsr $ee85      ;set CLK 1
., ee10  4c 97 ee jmp $ee97      ;set data 1
```

60947 ACPTR: RECEIVE FROM SERIAL BUS

The KERNAL routine ACPTR ($FFA5) jumps to this routine. It is essentially the reverse of the 'send data to serial bus' routine. A timing loop is entered using the CIA timer, and if a byte is not received within 65 milliseconds, ST is set to #02, ie. a read timeout. A test is made for EOI and if this occurs, ST is set to #40, indicating end of file. The byte is then received from the serial bus and built up bit by bit in the temporary store, $A4. This is transferred to (A) on exit, unless EOI has occurred.

```
., ee13  78       sei
., ee14  a9 00    lda #$00
., ee16  85 a5    sta $a5        ;CNTDN - counter
., ee18  20 85 ee jsr $ee85      ;set CLK 1
., ee1b  20 a9 ee jsr $eea9      ;get serial in and clock
., ee1e  10 fb    bpl $ee1b      ;wait for CLK =1
., ee20  a9 01    lda #$01
., ee22  8d 07 dc sta $dc07      ;CIA timer B high byte
., ee25  a9 19    lda #$19
., ee27  8d 0f dc sta $dc0f      ;start timer on 1 shot
., ee2a  20 97 ee jsr $ee97      ;set data 1
., ee2d  ad 0d dc lda $dc0d      ;CIA ICR
., ee30  ad 0d dc lda $dc0d
., ee33  29 02    and #$02       ;timeout?
., ee35  d0 07    bne $ee3e      ;yes
., ee37  20 a9 ee jsr $eea9      ;get serial in and clock
., ee3a  30 f4    bmi $ee30      ;CLK 1
., ee3c  10 18    bpl $ee56      ;CLK 0
., ee3e  a5 a5    lda $a5
., ee40  f0 05    beq $ee47
., ee42  a9 02    lda #$02       ;flag read timeout
., ee44  4c b2 ed jmp $edb2      ;set I/O status word
., ee47  20 a0 ee jsr $eea0      ;set data 1
., ee4a  20 85 ee jsr $ee85      ;set CLK 1
```

```
., ee4d  a9 40    lda #$40     ;flag EOI
., ee4f  20 1c fe jsr $fe1c    ;set I/O status word
., ee52  e6 a5    inc $a5      ;CNTDN
., ee54  d0 ca    bne $ee20
., ee56  a9 08    lda #$08
., ee58  85 a5    sta $a5      ;CNTDN
., ee5a  ad 00 dd lda $dd00    ;serial bus I/O port
., ee5d  cd 00 dd cmp $dd00
., ee60  d0 f8    bne $ee5a    ;wait for bus to settle
., ee62  0a       asl
., ee63  10 f5    bpl $ee5a    ;wait for data in = 1
., ee65  66 a4    ror $a4      ;temp data area
., ee67  ad 00 dd lda $dd00    ;serial bus I/O port
., ee6a  cd 00 dd cmp $dd00
., ee6d  d0 f8    bne $ee67    ;wait for bus to settle
., ee6f  0a       asl
., ee70  30 f5    bmi $ee67    ;wait for data in = 0
., ee72  c6 a5    dec $a5      ;CNTDN
., ee74  d0 e4    bne $ee5a
., ee76  20 a0 ee jsr $eea0    ;set data 1
., ee79  24 90    bit $90      ;STATUS - I/O status byte
., ee7b  50 03    bvc $ee80    ;not EOI
., ee7d  20 06 ee jsr $ee06    ;handshake & exit without
                               byte
., ee80  a5 a4    lda $a4      ;temp - holds received byte
., ee82  58       cli
., ee83  18       clc
., ee84  60       rts
```

61061 SERIAL CLOCK ON

This routine sets the clock out line on the serial bus to 1. By convention, this means writing a 0 to the port. This value is reversed by hardware on the bus.

```
., ee85  ad 00 dd lda $dd00    ;serial bus I/O register
., ee88  29 ef    and #$ef     ;set CLK out = 1
., ee8a  8d 00 dd sta $dd00
., ee8d  60       rts
```

61070 SERIAL CLOCK OFF

This routine sets the clock out line on the serial bus to 0. By convention, this means writing a 1 to the port. This value is reversed by hardware on the bus.

```
., ee8e  ad 00 dd lda $dd00    ;serial port I/O register
., ee91  09 10    ora #$10     ;set CLK out = 0
., ee93  8d 00 dd sta $dd00
```

```
., ee96  60        rts
```

61079 SERIAL OUTPUT 1

This routine sets the data out line on the serial bus to 1. By convention, this means writing a 0 to the port. This value is reversed by hardware on the bus.

```
., ee97  ad 00 dd lda $dd00    ;serial bus I/O register
., ee9a  29 df    and #$df     ;set data out = 1
., ee9c  8d 00 dd sta $dd00
., ee9f  60       rts
```

61088 SERIAL OUTPUT 0

This routine sets the data out line on the serial bus to 0. By convention, this means writing a 1 to the port. This value is reversed by hardware on the bus.

```
., eea0  ad 00 dd lda $dd00    ;serial bus I/O register
., eea3  09 20    ora #$20     ;set data out = 0
., eea5  8d 00 dd sta $dd00
., eea8  60       rts
```

61097 GET SERIAL DATA AND CLK IN

The serial port I/O register is stabilised and read. The data is shifted into carry and CLK into bit 7, thus the state of both inputs can be determined by flags in the status register. Note that the values read are true and do not need to be reversed in the same way that the output lines do.

```
., eea9  ad 00 dd lda $dd00    ;serial port I/O register
., eeac  cd 00 dd cmp $dd00
., eeaf  d0 f8    bne $eea9    ;wait for value to settle
., eeb1  0a       asl          ;put data in C, and CLK in
bit 7
., eeb2  60       rts
```

61107 DELAY 1 MS.

This is a software delay loop where (X) is repeatedly decremented for a period of 1 millisecond.

```
., eeb3  8a       txa
., eeb4  a2 b8    ldx #$b8
., eeb6  ca       dex
., eeb7  d0 fd    bne $eeb6
```

```
.,  eeb9  aa        tax
.,  eeba  60        rts
```

61115 RS-232 SEND

This routine is concerned with sending a byte on the RS-232 port. The data is actually written to the port under NMI interrupt control (the CTS line generates an NMI when the port is ready for the data). If all bits in the byte have been sent, then a new RS-232 byte is set up. Otherwise, this routine calculates parity and number of stop bits set up in the OPEN command. These bits are added to the end of the byte being sent.

```
.,  eebb  a5 b4     lda $b4      ;BITTS - RS-232 out bit count
.,  eebd  f0 47     beq $ef06    ;send new RS-232 byte
.,  eebf  30 3f     bmi $ef00
.,  eec1  46 b6     lsr $b6      ;RODATA - RS-232 out byte
                                 buffer
.,  eec3  a2 00     ldx #$00
.,  eec5  90 01     bcc $eec8
.,  eec7  ca        dex
.,  eec8  8a        txa
.,  eec9  45 bd     eor $bd      ;ROPRTY - RS-232 out parity
.,  eecb  85 bd     sta $bd
.,  eecd  c6 b4     dec $b4      ;BITTS
.,  eecf  f0 06     beq $eed7
.,  eed1  8a        txa
.,  eed2  29 04     and #$04
.,  eed4  85 b5     sta $b5      ;NXTBIT - next RS-232 bit to
                                 send
.,  eed6  60        rts
.,  eed7  a9 20     lda #$20
.,  eed9  2c 94 02  bit $0294    ;M51CDR - 6551 command
                                 register image
.,  eedc  f0 14     beq $eef2    ;no parity
.,  eede  30 1c     bmi $eefc    ;mark/space transmit
.,  eee0  70 14     bvs $eef6    ;even parity
.,  eee2  a5 bd     lda $bd      ;ROPRTY - out parity
.,  eee4  d0 01     bne $eee7
.,  eee6  ca        dex
.,  eee7  c6 b4     dec $b4      ;BITTS - out bit count
.,  eee9  ad 93 02  lda $0293    ;M51CTR - 6551 control
                                 register image
.,  eeec  10 e3     bpl $eed1    ;one stop bit only
.,  eeee  c6 b4     dec $b4      ;BITTS
.,  eef0  d0 df     bne $eed1
.,  eef2  e6 b4     inc $b4      ;BITTS
.,  eef4  d0 f0     bne $eee6
```

```
., eef6  a5 bd     lda $bd       ;ROPRTY
., eef8  f0 ed     beq $eee7
., eefa  d0 ea     bne $eee6
., eefc  70 e9     bvs $eee7
., eefe  50 e6     bvc $eee6
., ef00  e6 b4     inc $b4       ;BITTS
., ef02  a2 ff     ldx #$ff
., ef04  d0 cb     bne $eed1
```

61190 SEND NEW RS-232 BYTE

This routine sets up the system variables ready to send a new byte to the RS-232 port. A test is made for 3 line or X line mode. In X line mode, DSR and CTS are checked.

```
., ef06  ad 94 02 lda $0294      ;M51CDR - 6551 command
                                 register image
., ef09  4a        lsr           ;test handshake mode
., ef0a  90 07     bcc $ef13     ;3 line mode (no handshake)
., ef0c  2c 01 dd bit $dd01      ;RS-232 port
., ef0f  10 1d     bpl $ef2e     ;no DSR - error
., ef11  50 1e     bvc $ef31     ;no CTS - error
., ef13  a9 00     lda #$00
., ef15  85 bd     sta $bd       ;ROPRTY - RS-232 out parity
., ef17  85 b5     sta $b5       ;NXTBIT - next bit to send
., ef19  ae 98 02 ldx $0298      ;BITNUM - number of bits left
                                 to send
., ef1c  86 b4     stx $b4       ;BITTS - RS-232 out bit count
., ef1e  ac 9d 02 ldy $029d      ;RODBS - start page of out
                                 buffer
., ef21  cc 9e 02 cpy $029e      ;RODBE - index to end if out
                                 buffer
., ef24  f0 13     beq $ef39     ;disable timer
., ef26  b1 f9     lda ($f9),y   ;RS-232 out buffer
., ef28  85 b6     sta $b6       ;RODATA - RS-232 out byte
                                 buffer
., ef2a  ee 9d 02 inc $029d      ;RODBS
., ef2d  60        rts
```

61230 NO DSR / CTS ERROR

(A) is loaded with the error flag - #40 for no DSR and #10 for no CTS. This is then ORed with the 6551 status image and stored in RSSTAT.

```
., ef2e  a9 40     lda #$40      ;flag no DSR
., ef30  2c a9 10 bit $10a9      ;mask - flag no CTS
., ef33  0d 97 02 ora $0297      ;RSSTAT - 6522 status
                                 register image
```

```
., ef36  8d 97 02 sta $0297
```

61241 DISABLE TIMER

This routine sets the interrupt mask on CIA#2 timer B. It also clears the NMI flag.

```
., ef39  a9 01    lda #$01
., ef3b  8d 0d dd sta $dd0d     ;CIA interrupt control
                                register
., ef3e  4d a1 02 eor $02a1     ;ENABL - RS-232 enables
., ef41  09 80    ora #$80
., ef43  8d a1 02 sta $02a1     ;ENABL
., ef46  8d 0d dd sta $dd0d     ;CIA I.C.R.
., ef49  60       rts
```

61258 COMPUTE BIT COUNT

This routine computes the number of bits in the word to be sent. The word length information is held in bits 5 & 6 of M51CTR. Bit 7 of this register indicates the number of stop bits. On exit, the number of bits is held in (X).

```
., ef4a  a2 09    ldx #$09
., ef4c  a9 20    lda #$20
., ef4e  2c 93 02 bit $0293     ;M51CTR - 6551 control
                                register image
., ef51  f0 01    beq $ef54
., ef53  ca       dex
., ef54  50 02    bvc $ef58
., ef56  ca       dex
., ef57  ca       dex
., ef58  60       rts
```

61273 RS-232 RECEIVE

This routine builds up the input byte from the RS-232 port in RIDATA. Each bit is input from the port under NMI interrupt control. The bit is placed in INBIT befoore being passed to this routine, where it is shifted into the carry flag and then rotated into RIDATA. The bit count is decremented and parity updated.

```
., ef59  a6 a9    ldx $a9       ;RINONE - check for start
                                bit?
., ef5b  d0 33    bne $ef90
., ef5d  c6 a8    dec $a8       ;BITC1 - RS-232 in bit count
., ef5f  f0 36    beq $ef97     ;process received byte
., ef61  30 0d    bmi $ef70
```

```
., ef63  a5 a7     lda $a7       ;INBIT - RS-232 in bits
., ef65  45 ab     eor $ab       ;RIPRTY - RS-232 in parity
., ef67  85 ab     sta $ab
., ef69  46 a7     lsr $a7       ;INBIT - put input bit into
                                 carry
., ef6b  66 aa     ror $aa       ;RIDATA - RS-232 in byte
                                 buffer
., ef6d  60        rts
., ef6e  c6 a8     dec $a8       ;BITC1
., ef70  a5 a7     lda $a7       ;INBIT
., ef72  f0 67     beq $efdb
., ef74  ad 93 02 lda $0293      ;M51CTR - 6551 control
                                 register image
., ef77  0a        asl
., ef78  a9 01     lda #$01
., ef7a  65 a8     adc $a8       ;BITC1
., ef7c  d0 ef     bne $ef6d     ;end
```

61310 SET UP TO RECEIVE

This routine sets up the I.C.R. to wait for the receiver edge, and flags this into ENABL. It then flags the check for a start bit.

```
., ef7e  a9 90     lda #$90
., ef80  8d 0d dd sta $dd0d      ;CIA I.C.R.
., ef83  0d a1 02 ora $02a1      ;ENABL - RS-232 enables
., ef86  8d a1 02 sta $02a1
., ef89  85 a9     sta $a9       ;RINONE - check for start bit
., ef8b  a9 02     lda #$02
., ef8d  4c 3b ef jmp $ef3b      ;disable timer & end
```

61328 PROCESS RS-232 BYTE

The byte received from the RS-232 port is checked against parity. This involves checking the input parity options selected, and then verifying the parity bit calculated against that input. If the test is passed, then the byte is stored in the in-buffer. Otherwise an error is flagged into RSSTAT.

```
., ef90  a5 a7     lda $a7       ;INBIT - RS-232 in bits
., ef92  d0 ea     bne $ef7e     ;set up to receive
., ef94  85 a9     sta $a9       ;RINONE - check for start bit
., ef96  60        rts
., ef97  ac 9b 02 ldy $029b      ;RIDBE - index to end of in
                                 buffer
., ef9a  c8        iny
., ef9b  cc 9c 02 cpy $029c      ;RIDBS - start page of in
```

```
                                   buffer
., ef9e  f0 2a    beq $efca        ;receive overflow error
., efa0  8c 9b 02 sty $029b        ;RIDBE
., efa3  88       dey
., efa4  a5 aa    lda $aa          ;RIDATA - RS-232 in byte
                                   buffer
., efa6  ae 98 02 ldx $0298        ;BITNUM - number of bits left
                                   to send
., efa9  e0 09    cpx #$09         ;full word to come?
., efab  f0 04    beq $efb1y       ;yes
., efad  4a       lsr
., efae  e8       inx
., efaf  d0 f8    bne $efa9
., efb1  91 f7    sta ($f7),y      ;RIBUF - RS-232 in buffer
                                   (store byte)
., efb3  a9 20    lda #$20
., efb5  2c 94 02 bit $0294        ;M51CDR - 6551 command
                                   register image
., efb8  f0 b4    beq $ef6e        ;parity disabled
., efba  30 b1    bmi $ef6d        ;parity check disabled - RTS
., efbc  a5 a7    lda $a7          ;INBIT - parity bit
., efbe  45 ab    eor $ab          ;RIPRTY - RS-232 in parity
., efc0  f0 03    beq $efc5        ;receive parity error
., efc2  70 a9    bvs $ef6d
., efc4  2c 50 a6 bit $a650        ;mask - receive parity error
., efc7  a9 01    lda #$01
., efc9  2c a9 04 bit $04a9        ;mask - receive overflow
., efcc  2c a9 80 bit $80a9        ;mask - receive break
., efcf  2c a9 02 bit $02a9        ;mask - framing error
., efd2  0d 97 02 ora $0297        ;RSSTAT - 6551 status
                                   register image
., efd5  8d 97 02 sta $0297
., efd8  4c 7e ef jmp $ef7e        ;set up to receive
., efdb  a5 aa    lda $aa          ;RIDATA
., efdd  d0 f1    bne $efd0        ;framing error
., efdf  f0 ec    beq $efcd        ;receive break
```

<u>61409 SUBMIT TO RS-232</u>

This routine is called when data is required from the RS-232 port. Its function is to perform the handshaking on the port needed to receive the data. If 3 line mode is being used, then no handshaking is implemented and the routine exits.

```
., efe1  85 9a    sta $9a          ;DFLTO - default output
                                   device
., efe3  ad 94 02 lda $0294        ;M51CDR - 6551 command
                                   register image
```

```
., efe6  4a        lsr
., efe7  90 29     bcc $f012     ;3 line RS-232 - no
                                 handshaking
., efe9  a9 02     lda #$02
., efeb  2c 01 dd bit $dd01     ;RS-232 I/O port
., efee  10 1d     bpl $f00d     ;no DSR - error
., eff0  d0 20     bne $f012
., eff2  ad a1 02 lda $02a1     ;ENABL - RS-232 enables
., eff5  29 02     and #$02
., eff7  d0 f9     bne $eff2
., eff9  2c 01 dd bit $dd01     ;RS-232 I/O port
., effc  70 fb     bvs $eff9     ;wait for no CTS
., effe  ad 01 dd lda $dd01
., f001  09 02     ora #$02
., f003  8d 01 dd sta $dd01     ;set RTS (Request To Send)
., f006  2c 01 dd bit $dd01
., f009  70 07     bvs $f012     ;CTS set
., f00b  30 f9     bmi $f006     ;wait for no DSR
```

61453 NO DSR ERROR

This routine sets the 6551 Status register image to #40 when a no DSR (Data Set Ready) error has occurred.

```
., f00d  a9 40     lda #$40
., f00f  8d 97 02 sta $0297     ;RSSTAT - 6551 status
                                 register image
., f012  18        clc
., f013  60        rts
```

61463 SEND TO RS-232 BUFFER

Note: The entry point to the routine is at $F017, not $F014. The byte of data being held in PTR1 is placed in the RS-232 out buffer. A test is then made to see if the port is in transmit mode. If so then no further action is taken. If not, the baud rate is set in the CIA timer and CTS is forced.

```
., f014  20 28 f0 jsr $f028     ;test for transmit mode
., f017  ac 9e 02 ldy $029e     ;RODBE - index to RS-232 out
                                 buffer
., f01a  c8        iny
., f01b  cc 9d 02 cpy $029d     ;RODBS - start page of output
                                 buffer
., f01e  f0 f4     beq $f014
., f020  8c 9e 02 sty $029e     ;RODBE
., f023  88        dey
., f024  a5 9e     lda $9e       ;PTR1
```

```
., f026  91 f9    sta ($f9),y  ;store character in buffer
., f028  ad a1 02 lda $02a1    ;ENABL - RS-232 enables
., f02b  4a       lsr
., f02c  b0 1e    bcs $f04c    ;transmit mode on - end
., f02e  a9 10    lda #$10     ;force load timer A
., f030  8d 0e dd sta $dd0e    ;CIA control register A
., f033  ad 99 02 lda $0299    ;<BAUDOF - RS-232 baud rate
., f036  8d 04 dd sta $dd04    ;timer A low
., f039  ad 9a 02 lda $029a    ;>BAUDOF
., f03c  8d 05 dd sta $dd05    ;timer A high
., f03f  a9 81    lda #$81
., f041  20 3b ef jsr $ef3b    ;set interrupt control
                               register
., f044  20 06 ef jsr $ef06    ;send RS-232
., f047  a9 11    lda #$11
., f049  8d 0e dd sta $dd0e    ;force timer out on RS-232
                               port (CTS)
., f04c  60       rts
```

61517 INPUT FROM RS-232

The RS-232 port is prepared for input. The data is actually input from the port via the NMI interrupt service routine. A test is made for 3 line/X line mode and full/half duplex. In half duplex mode, the port is set to transmit data, and a loop waits for the outgoing data to be cleared before continuing. RTS is cleared and DTR waited for, before the NMI control register is set up.

```
., f04d  85 99    sta $99      ;DFLTN - default input device
., f04f  ad 94 02 lda $0294    ;M51CDR - 6551 command
                               register image
., f052  4a       lsr
., f053  90 28    bcc $f07d    ;3 line mode - no handshaking
., f055  29 08    and #$08     ;test duplex
., f057  f0 24    beq $f07d    ;full duplex
., f059  a9 02    lda #$02
., f05b  2c 01 dd bit $dd01    ;RS-232 I/O port
., f05e  10 ad    bpl $f00d    ;no DSR - error
., f060  f0 22    beq $f084
., f062  ad a1 02 lda $02a1    ;ENABL - RS-232 enables
., f065  4a       lsr
., f066  b0 fa    bcs $f062    ;system transmitting - wait
., f068  ad 01 dd lda $dd01    ;RS-232 I/O port
., f06b  29 fd    and #$fd     ;clear RTS (Request To Send)
., f06d  8d 01 dd sta $dd01
., f070  ad 01 dd lda $dd01    ;RS-232 I/O port
., f073  29 04    and #$04
., f075  f0 f9    beq $f070    ;wait for DTR (Data Terminal
```

```
                                      Ready)
., f077  a9 90     lda #$90
., f079  18        clc
., f07a  4c 3b ef  jmp $ef3b         ;set CIA interrupt control
                                      register
., f07d  ad a1 02  lda $02a1         ;ENABL
., f080  29 12     and #$12          ;test for port
                                      recieving/awaiting data
., f082  f0 f3     beq $f077         ;neither
., f084  18        clc
., f085  60        rts
```

61574 GET FROM RS-232

This routine returns a byte from the RS-232 in buffer. If the buffer is empty, this state is flagged in RSSTAT and a null byte returned.

```
., f086  ad 97 02  lda $0297         ;RSSTAT - 6551 status
                                      register image
., f089  ac 9c 02  ldy $029c         ;RIDBS - start page of RS-232
                                      in buffer
., f08c  cc 9b 02  cpy $029b         ;RIDBE - index to end of in
                                      buffer
., f08f  f0 0b     beq $f09c
., f091  29 f7     and #$f7          ;clear break flag bit
., f093  8d 97 02  sta $0297         ;RSSTAT
., f096  b1 f7     lda ($f7),y       ;get character from in buffer
., f098  ee 9c 02  inc $029c         ;RIDBS
., f09b  60        rts
., f09c  09 08     ora #$08          ;flag receiver buffer empty
., f09e  8d 97 02  sta $0297         ;RSSTAT
., f0a1  a9 00     lda #$00
., f0a3  60        rts
```

61604 SERIAL BUS IDLE

This routine checks the RS-232 bus for data transmission/reception. The routine waits for any activity on the bus to end before setting the I.C.R. The routine is called by tape and serial bus routines, since these devices use IRQ generated timing, and conflicts may occur if they are all used at once.

```
., f0a4  48        pha
., f0a5  ad a1 02  lda $02a1         ;ENABL - RS-232 enables
., f0a8  f0 11     beq $f0bb         ;bus not in use
., f0aa  ad a1 02  lda $02a1         ;ENABL
., f0ad  29 03     and #$03          ;RS-232
```

```
                                  recieving/transmitting?
., f0af  d0 f9     bne $f0aa      ;yes - wait for port to clear
., f0b1  a9 10     lda #$10
., f0b3  8d 0d dd  sta $dd0d      ;CIA I.C.R.
., f0b6  a9 00     lda #$00
., f0b8  8d a1 02  sta $02a1      ;ENABL
., f0bb  68        pla
., f0bc  60        rts
```

61629 TABLE OF KERNAL I/O MESSAGES

This is a table of messages used by the Kernal in conjunction with its I/O routines. Bit 7 is set in the last character in each message as a terminator.

```
.:f0bd 0d 49 2f 4f 20 45 52 52    i/o err
.:f0c5 4f 52 20 a3 0d 53 45 41   or # sea
.:f0cd 52 43 48 49 4e 47 a0 46   rching f
.:f0d5 4f 52 a0 0d 50 52 45 53   or  pres
.:f0dd 53 20 50 4c 41 59 20 4f   s play o
.:f0e5 4e 20 54 41 50 c5 50 52   n tapepr
.:f0ed 45 53 53 20 52 45 43 4f   ess reco
.:f0f5 52 44 20 26 20 50 4c 41   rd & pla
.:f0fd 59 20 4f 4e 20 54 41 50   y on tap
.:f105 c5 0d 4c 4f 41 44 49 4e   e loadin
.:f10d c7 0d 53 41 56 49 4e 47   g saving
.:f115 a0 0d 56 45 52 49 46 59     verify
.:f11d 49 4e c7 0d 46 4f 55 4e   ing foun
.:f125 44 a0 0d 4f 4b 8d 24 9d   d  ok
```

61739 PRINT MESSAGE IF DIRECT

This is a routine to output a message from the I/O messages table starting at $F0BD. On entry, (Y) holds the offset to control which message is printed.

```
., f12b  24 9d     bit $9d        ;MSGFLG - direct/program
                                  mode?
., f12d  10 0d     bpl $f13c      ;program - don't print
                                  message
., f12f  b9 bd f0  lda $f0bd,y    ;get character from message
                                  table
., f132  08        php
., f133  29 7f     and #$7f       ;cleat bit 7
., f135  20 d2 ff  jsr $ffd2      ;CHROUT - output character
., f138  c8        iny
., f139  28        plp
., f13a  10 f3     bpl $f12f      ;get next character in
                                  message
```

```
., f13c  18        clc
., f13d  60        rts
```

61758 GETIN: GET A BYTE...

The KERNAL routine GETIN ($FFE4) is vectored to this routine. It loads (gets) a character into fac#1 from the external device indicated by DFLTN. Thus, if device = 0, GET is from the keyboard buffer (assuming that it is not empty). If device = 2, GET is from the RS-232 port. If neither of these devices then GET is further handled by the INPUT routine following on from it.

```
., f13e  a5 99     lda $99      ;DFLTN - default input device
., f140  d0 08     bne $f14a    ;not keyboard
., f142  a5 c6     lda $c6      ;NDX - no. chars. in keyboard
                                 queue
., f144  f0 0f     beq $f155    ;buffer is empty - end
., f146  78        sei
., f147  4c b4 e5  jmp $e5b4    ;get character from keyboard
                                 buffer
., f14a  c9 02     cmp #$02     ;RS-232?
., f14c  d0 18     bne $f166    ;no - try next device
., f14e  84 97     sty $97      ;temp data area
., f150  20 86 f0  jsr $f086    ;get character from RS-232
., f153  a4 97     ldy $97      ;temp data area
., f155  18        clc
., f156  60        rts
```

61783 CHRIN: INPUT A BYTE...

The KERNAL routine CHRIN ($FFCF) is vectored to this routine. It is similar in function to the GET routine above, and also provides a continuation to that routine. If the input device is 0 or 3, ie keyboard or screen, then input takes place from the screen. INPUT/GET from other devices are performed by calls to the next routine. Two bytes are input from the device so that end of file can be set if necessary (ie. ST = #40).

```
., f157  a5 99     lda $99      ;DFLTN - default input device
., f159  d0 0b     bne $f166    ;not keyboard
., f15b  a5 d3     lda $d3      ;PNTR - cursor column on
                                 screen
., f15d  85 ca     sta $ca      ;>LXSP - cursor position at
                                 start
., f15f  a5 d6     lda $d6      ;TBLX - cursor line number
., f161  85 c9     sta $c9      ;<LXSP
., f163  4c 32 e6  jmp $e632    ;input from screen or
```

```
                                                keyboard
., f166  c9 03      cmp #$03       ;screen?
., f168  d0 09      bne $f173      ;no - next device
., f16a  85 d0      sta $d0        ;CRSW - flag INPUT/GET from
                                    keyboard
., f16c  a5 d5      lda $d5        ;LNMX - physical screen line
                                    length
., f16e  85 c8      sta $c8        ;INDX - end of logical line
                                    for INPUT
., f170  4c 32 e6 jmp $e632        ;input from screen or
                                    keyboard
., f173  b0 38      bcs $f1ad
., f175  c9 02      cmp #$02       ;RS-232?
., f177  f0 3f      beq $f1b8      ;yes - get from RS-232 port
., f179  86 97      stx $97        ;temp data area
., f17b  20 99 f1 jsr $f199        ;get byte from tape/serial
                                    bus
., f17e  b0 16      bcs $f196
., f180  48         pha
., f181  20 99 f1 jsr $f199        ;get byte from tape/serial
                                    bus
., f184  b0 0d      bcs $f193
., f186  d0 05      bne $f18d
., f188  a9 40      lda #$40       ;flag EOI
., f18a  20 1c fe jsr $fe1c        ;set I/O status word
., f18d  c6 a6      dec $a6        ;BUFPNT - tape I/O buffer
                                    pointer
., f18f  a6 97      ldx $97        ;temp data area
., f191  68         pla
., f192  60         rts
., f193  aa         tax
., f194  68         pla
., f195  8a         txa
., f196  a6 97      ldx $97        ;temp data area
., f198  60         rts
```

<u>61849 GET FROM TAPE/SERIAL/RS-232</u>

This is effectively three separate routines in one, since each device is dealt with separately. It is assumed that the routine is entered from a previous routine in the GET/INPUT chain at the start point for the relevant device. In the first section, the Cassette Read advances the buffer pointer and loads another buffer of data if necessary. The serial bus section checks the state of ST. If zero, then the data is received from the bus, otherwise carriage return (#0D) is returned in (A). In the third section, the byte is read from the RS-232 port.

```
., f199  20 0d f8 jsr $f80d      ;bump tape pointer
., f19c  d0 0b    bne $f1a9      ;buffer not full
., f19e  20 41 f8 jsr $f841      ;initiate tape read
., f1a1  b0 11    bcs $f1b4      ;end
., f1a3  a9 00    lda #$00
., f1a5  85 a6    sta $a6        ;BUFPNT - tape buffer pointer
., f1a7  f0 f0    beq $f199
., f1a9  b1 b2    lda ($b2),y    ;TAPE1 - read tape buffer
., f1ab  18       clc
., f1ac  60       rts
., f1ad  a5 90    lda $90        ;STATUS - I/O status word
., f1af  f0 04    beq $f1b5      ;status ok
., f1b1  a9 0d    lda #$0d       ;return <CR> value
., f1b3  18       clc
., f1b4  60       rts
., f1b5  4c 13 ee jmp $ee13      ;receive from serial bus
., f1b8  20 4e f1 jsr $f14e      ;get from RS-232
., f1bb  b0 f7    bcs $f1b4      ;end with Carry set
., f1bd  c9 00    cmp #$00
., f1bf  d0 f2    bne $f1b3      ;end with Carry clear
., f1c1  ad 97 02 lda $0297      ;RSSTAT - 6551 status
                                 register image
., f1c4  29 60    and #$60
., f1c6  d0 e9    bne $f1b1      ;return with <CR>
., f1c8  f0 ee    beq $f1b8      ;get from RS-232
```

61898 CHROUT: OUTPUT ONE CHARACTER

The KERNAL routine CHROUT ($FFD2) is vectored to this routine. On entry, (A) must hold the character to be output. The default output device number is examined and output directed to the relevant device. The screen, serial bus and RS-232 all use previously described routines for their output. Tape writes the byte to the cassette buffer, only sending the buffer contents out to the cassette port when the buffer is full.

```
., f1ca  48       pha
., f1cb  a5 9a    lda $9a        ;DFLTO - default output
                                 device
., f1cd  c9 03    cmp #$03       ;screen?
., f1cf  d0 04    bne $f1d5      ;no
., f1d1  68       pla
., f1d2  4c 16 e7 jmp $e716      ;output to screen
., f1d5  90 04    bcc $f1db      ;device <3
., f1d7  68       pla
., f1d8  4c dd ed jmp $eddd      ;send serial deferred
., f1db  4a       lsr
., f1dc  68       pla
```

```
., f1dd  85 9e     sta $9e        ;PTR1
., f1df  8a        txa
., f1e0  48        pha
., f1e1  98        tya
., f1e2  48        pha
., f1e3  90 23     bcc $f208      ;RS-232
., f1e5  20 0d f8  jsr $f80d      ;bump tape pointer
., f1e8  d0 0e     bne $f1f8      ;buffer not full
., f1ea  20 64 f8  jsr $f864      ;initiate tape write
., f1ed  b0 0e     bcs $f1fd
., f1ef  a9 02     lda #$02
., f1f1  a0 00     ldy #$00
., f1f3  91 b2     sta ($b2),y
., f1f5  c8        iny
., f1f6  84 a6     sty $a6        ;BUFPNT - pointer to tape
                                  buffer
., f1f8  a5 9e     lda $9e        ;PTR1
., f1fa  91 b2     sta ($b2),y    ;TAPE1 - write byte to tape
                                  buffer
., f1fc  18        clc
., f1fd  68        pla
., f1fe  a8        tay
., f1ff  68        pla
., f200  aa        tax
., f201  a5 9e     lda $9e        ;PTR1
., f203  90 02     bcc $f207
., f205  a9 00     lda #$00
., f207  60        rts
., f208  20 17 f0  jsr $f017      ;send to RS-232
., f20b  4c fc f1  jmp $f1fc      ;end output
```

<u>61966 CHKIN: SET INPUT DEVICE</u>

The KERNAL routine CHKIN ($FFC6) is vectored to this routine. On entry, (X) must hold the logical file number. A test is made to see if the file is open, or ?FILE NOT OPEN. If the file is not an input file then ?NOT INPUT FILE. If the device is on the serial bus then it is commanded to TALK and the secondary address is sent. ST is then checked and if non-zero, then ?DEVICE NOT PRESENT. Finally, the device number is stored in DFLTN.

```
., f20e  20 0f f3  jsr $f30f      ;find file number
., f211  f0 03     beq $f216
., f213  4c 01 f7  jmp $f701      ;I/O error #3 - file not open
., f216  20 1f f3  jsr $f31f      ;set file values
., f219  a5 ba     lda $ba        ;FA - current device number
., f21b  f0 16     beq $f233      ;keyboard
., f21d  c9 03     cmp #$03       ;screen?
```

```
., f21f   f0 12    beq $f233     ;yes
., f221   b0 14    bcs $f237     ;serial bus device
., f223   c9 02    cmp #$02      ;RS-232?
., f225   d0 03    bne $f22a     ;no
., f227   4c 4d f0 jmp $f04d     ;input from RS-232
., f22a   a6 b9    ldx $b9       ;SA - current secondary
                                 address
., f22c   e0 60    cpx #$60
., f22e   f0 03    beq $f233
., f230   4c 0a f7 jmp $f70a     ;I/O error #6 - not output
                                 file
., f233   85 99    sta $99       ;DFLTN - default input device
., f235   18       clc
., f236   60       rts
., f237   aa       tax
., f238   20 09 ed jsr $ed09     ;send TALK to serial device
., f23b   a5 b9    lda $b9       ;SA
., f23d   10 06    bpl $f245     ;send SA
., f23f   20 cc ed jsr $edcc     ;wait for clock
., f242   4c 48 f2 jmp $f248
., f245   20 c7 ed jsr $edc7     ;send talk secondary address
., f248   8a       txa
., f249   24 90    bit $90       ;STATUS - I/O status word
., f24b   10 e6    bpl $f233     ;store default input device
., f24d   4c 07 f7 jmp $f707     ;I/O error #5 - device not
                                 present
```

62032 CHKOUT: SET OUTPUT DEVICE

The KERNAL routine CHKOUT ($FFC9) is vectored to this routine. On entry, (X) must hold the logical file number. A test is made to see if the file is open, or ?FILE NOT OPEN error. If the device is 0 (ie. the keyboard), or the file is not an output file then ?NOT OUTPUT FILE error is generated. If the device is on the serial bus then it is commanded to LISTEN and the secondary address is sent. ST is then checked and if non-zero, then ?DEVICE NOT PRESENT error. Finally, the device number is stored in DFLTO.

```
., f250   20 0f f3 jsr $f30f     ;find file number
., f253   f0 03    beq $f258
., f255   4c 01 f7 jmp $f701     ;I/O error #3 - file not open
., f258   20 1f f3 jsr $f31f     ;set file values
., f25b   a5 ba    lda $ba       ;FA - current device number
., f25d   d0 03    bne $f262
., f25f   4c 0d f7 jmp $f70d     ;I/O error #7 - not output
                                 file
., f262   c9 03    cmp #$03      ;screen?
., f264   f0 0f    beq $f275
```

```
., f266  b0 11     bcs $f279    ;serial bus device
., f268  c9 02     cmp #$02     ;RS-232?
., f26a  d0 03     bne $f26f    ;no
., f26c  4c e1 ef jmp $efe1     ;submit to RS-232
., f26f  a6 b9     ldx $b9      ;SA - current secondary
                                address
., f271  e0 60     cpx #$60
., f273  f0 ea     beq $f25f    ;not output file error
., f275  85 9a     sta $9a      ;DFLTO - default output
                                device
., f277  18        clc
., f278  60        rts
., f279  aa        tax
., f27a  20 0c ed jsr $ed0c     ;send LISTEN to serial device
., f27d  a5 b9     lda $b9      ;SA
., f27f  10 05     bpl $f286    ;send SA
., f281  20 be ed jsr $edbe     ;clear ATN
., f284  d0 03     bne $f289
., f286  20 b9 ed jsr $edb9     ;send listen secondary
                                address
., f289  8a        txa
., f28a  24 90     bit $90      ;STATUS - I/O status word
., f28c  10 e7     bpl $f275    ;set output device
., f28e  4c 07 f7 jmp $f707     ;I/O error #5 - device not
                                present
```

62097 CLOSE: CLOSE FILE

The KERNAL routine CLOSE ($FFC3) is vectored here. The file parameters are fetched, and if not found, the routine exits without action. The device number associated with the file is checked. If it is RS-232, then the RS-232 port is reset. If cassette, then a zero byte is written to the tape, and if desired, the optional End-Of-Tape header can be written. If it is a serial device (>3), then the device is UNTALKed or UNLISTENed. Finally, for all devices, the number of open logical files is decremented and the table of active file numbers updated.

```
., f291  20 14 f3 jsr $f314     ;find logical file
., f294  f0 02     beq $f298
., f296  18        clc          ;file not found
., f297  60        rts
., f298  20 1f f3 jsr $f31f     ;set file values
., f29b  8a        txa
., f29c  48        pha
., f29d  a5 ba     lda $ba      ;FA - current device number
., f29f  f0 50     beq $f2f1    ;keyboard - update file table
., f2a1  c9 03     cmp #$03     ;screen?
```

```
., f2a3  f0 4c     beq $f2f1     ;yes - update file table
., f2a5  b0 47     bcs $f2ee     ;serial bus
., f2a7  c9 02     cmp #$02      ;RS-232?
., f2a9  d0 1d     bne $f2c8     ;no
., f2ab  68        pla
., f2ac  20 f2 f2  jsr $f2f2     ;decrement file table
., f2af  20 83 f4  jsr $f483
., f2b2  20 27 fe  jsr $fe27     ;read top of memory into
                                 (X/Y)
., f2b5  a5 f8     lda $f8       ;>RIBUF - RS-232 input buffer
                                 pointer
., f2b7  f0 01     beq $f2ba
., f2b9  c8        iny
., f2ba  a5 fa     lda $fa       ;>ROBUF - RS-232 output
                                 buffer pointer
., f2bc  f0 01     beq $f2bf
., f2be  c8        iny
., f2bf  a9 00     lda #$00
., f2c1  85 f8     sta $f8       ;zero RIBUF
., f2c3  85 fa     sta $fa       ;zero ROBUF
., f2c5  4c 7d f4  jmp $f47d
., f2c8  a5 b9     lda $b9       ;SA - current secondary
                                 address
., f2ca  29 0f     and #$0f
., f2cc  f0 23     beq $f2f1     ;decrement file table
., f2ce  20 d0 f7  jsr $f7d0     ;get tape buffer address
., f2d1  a9 00     lda #$00
., f2d3  38        sec
., f2d4  20 dd f1  jsr $f1dd     ;output zero byte to tape
., f2d7  20 64 f8  jsr $f864     ;initiate tape write
., f2da  90 04     bcc $f2e0
., f2dc  68        pla
., f2dd  a9 00     lda #$00
., f2df  60        rts
., f2e0  a5 b9     lda $b9       ;SA
., f2e2  c9 62     cmp #$62
., f2e4  d0 0b     bne $f2f1     ;decrement file table
., f2e6  a9 05     lda #$05      ;flag end-of-file
., f2e8  20 6a f7  jsr $f76a     ;write tape header
., f2eb  4c f1 f2  jmp $f2f1     ;adjust file table
., f2ee  20 42 f6  jsr $f642     ;UNTALK/UNLISTEN device
., f2f1  68        pla
., f2f2  aa        tax
., f2f3  c6 98     dec $98       ;LDTND - number of open files
., f2f5  e4 98     cpx $98
., f2f7  f0 14     beq $f30d
., f2f9  a4 98     ldy $98       ;LDTND
., f2fb  b9 59 02  lda $0259,y   ;LAT - table of active file
                                 numbers
```

```
., f2fe  9d 59 02 sta $0259,x  ;update table
., f301  b9 63 02 lda $0263,y
., f304  9d 63 02 sta $0263,x
., f307  b9 6d 02 lda $026d,y
., f30a  9d 6d 02 sta $026d,x
., f30d  18       clc
., f30e  60       rts
```

62223 FIND FILE

On entry, (X) must hold the logical file number to be found. The table of file numbers is searched, and if the file is found, the Z flag is set and (X) provides the offset to the position of the file in the table. Conversely, if the file number is not found, Z=0.

```
., f30f  a9 00    lda #$00
., f311  85 90    sta $90      ;STATUS - I/O status word
., f313  8a       txa
., f314  a6 98    ldx $98      ;LDTND - number of open files
., f316  ca       dex
., f317  30 15    bmi $f32e    ;end of table - RTS
., f319  dd 59 02 cmp $0259,x  ;LAT - table of logical files
., f31c  d0 f8    bne $f316    ;no - try next file in table
., f31e  60       rts
```

62239 SET FILE VALUES

This routine sets the current logical file number, device number and secondary address from the file parameter tables. On entry, (X) must hold the offset to the position of the file in the table.

```
., f31f  bd 59 02 lda $0259,x  ;LAT - table of active
                               logical files
., f322  85 b8    sta $b8      ;LA - current logical file
                               number
., f324  bd 63 02 lda $0263,x  ;FAT - table of device
                               numbers
., f327  85 ba    sta $ba      ;LA - current device number
., f329  bd 6d 02 lda $026d,x  ;SAT - table of secondary
                               addresses
., f32c  85 b9    sta $b9      ;SA - current secondary
                               address
., f32e  60       rts
```

62255 CLALL: ABORT ALL FILES

The KERNAL routine CLALL ($FFE7) is vectored here. The

number of open files is set to zero and the next routine performed.

```
., f32f  a9 00     lda #$00
., f331  85 98     sta $98       ;LDTND - number of open files
```

62259 CLRCHN: RESTORE DEFAULT I/O

The KERNAL routine CLRCHN ($FFCC) is vectored here. The default output device is UNLISTENED if it is on the serial bus, and then the device is set to screen. The default input device is UNTALKED if it is on the serial bus, and then set to keyboard.

```
., f333  a2 03     ldx #$03
., f335  e4 9a     cpx $9a       ;DFLTO - default output
                                 device
., f337  b0 03     bcs $f33c
., f339  20 fe ed  jsr $edfe     ;send UNLISTEN to serial bus
., f33c  e4 99     cpx $99       ;DFLTN - default input device
., f33e  b0 03     bcs $f343
., f340  20 ef ed  jsr $edef     ;send UNTALK to serial bus
., f343  86 9a     stx $9a       ;DFLTO
., f345  a9 00     lda #$00
., f347  85 99     sta $99       ;DFLTN
., f349  60        rts
```

62282 OPEN: OPEN FILE

The KERNAL routine OPEN ($FFC0) is vectored here. It is assumed that the file parameters are already set on entry. The table of logical file numbers is searched in case a file already exists, in which case I/O error #2 - ?FILE OPEN is generated. If there are more than 10 logical files open then I/O error #1 - ?TOO MANY FILES is generated. The file parameters are set into their respective tables, and then the device number is checked. Keyboard and screen exit here with no further action. RS-232 is opened via a separate routine. Secondary address and file name are sent to the serial bus. If the secondary address for tape is 0, then it is a read file. The 'PRESS PLAY...' message is printed and the next header (or a specified header if a file name is given) is searched for. I/O error #4 if the file is not found. If the secondary address is 1 or 2, then it is a write file and the tape header is written.

```
., f34a  a6 b8     ldx $b8       ;LA - current logical file
                                 number
., f34c  d0 03     bne $f351
```

```
.,  f34e  4c 0a f7 jmp $f70a    ;I/O error #6 - not input
                                file
.,  f351  20 0f f3 jsr $f30f    ;find file
.,  f354  d0 03    bne $f359
.,  f356  4c fe f6 jmp $f6fe    ;I/O error #2 - file exists
.,  f359  a6 98    ldx $98      ;LDTND - number of open files
.,  f35b  e0 0a    cpx #$0a
.,  f35d  90 03    bcc $f362    ;less than 10
.,  f35f  4c fb f6 jmp $f6fb    ;I/O error #1 - too many
                                files
.,  f362  e6 98    inc $98      ;LDTND
.,  f364  a5 b8    lda $b8      ;LA
.,  f366  9d 59 02 sta $0259,x  ;LAT - table of active file
                                numbers
.,  f369  a5 b9    lda $b9      ;SA - current secondary
                                address
.,  f36b  09 60    ora #$60
.,  f36d  85 b9    sta $b9      ;SA
.,  f36f  9d 6d 02 sta $026d,x  ;SAT - table of secondary
                                addresses
.,  f372  a5 ba    lda $ba      ;FA - current device number
.,  f374  9d 63 02 sta $0263,x  ;FAT - table of device
                                numbers
.,  f377  f0 5a    beq $f3d3    ;keyboard - end
.,  f379  c9 03    cmp #$03     ;screen?
.,  f37b  f0 56    beq $f3d3    ;yes - end
.,  f37d  90 05    bcc $f384    ;not serial bus
.,  f37f  20 d5 f3 jsr $f3d5    ;send SA
.,  f382  90 4f    bcc $f3d3    ;end
.,  f384  c9 02    cmp #$02     ;RS-232?
.,  f386  d0 03    bne $f38b    ;no
.,  f388  4c 09 f4 jmp $f409    ;open RS-232 file
.,  f38b  20 d0 f7 jsr $f7d0    ;get tape buffer address
.,  f38e  b0 03    bcs $f393
.,  f390  4c 13 f7 jmp $f713    ;I/O error #9 - illegal
                                device number
.,  f393  a5 b9    lda $b9      ;SA
.,  f395  29 0f    and #$0f
.,  f397  d0 1f    bne $f3b8    ;write file
.,  f399  20 17 f8 jsr $f817    ;print "PRESS PLAY..."
                                message
.,  f39c  b0 36    bcs $f3d4    ;RTS
.,  f39e  20 af f5 jsr $f5af    ;print "SEARCHING" message
.,  f3a1  a5 b7    lda $b7      ;FNLEN - length of current
                                file name
.,  f3a3  f0 0a    beq $f3af
.,  f3a5  20 ea f7 jsr $f7ea    ;find specific tape header
.,  f3a8  90 18    bcc $f3c2
.,  f3aa  f0 28    beq $f3d4    ;RTS
```

```
., f3ac  4c 04 f7 jmp $f704     ;I/O error #4 - file not
                                found
., f3af  20 2c f7 jsr $f72c     ;find any tape header
., f3b2  f0 20    beq $f3d4     ;RTS
., f3b4  90 0c    bcc $f3c2
., f3b6  b0 f4    bcs $f3ac     ;I/O error #4 - file not
                                found
., f3b8  20 38 f8 jsr $f838     ;print "PRESS RECORD..."
                                message
., f3bb  b0 17    bcs $f3d4     ;RTS
., f3bd  a9 04    lda #$04
., f3bf  20 6a f7 jsr $f76a     ;write tape header
., f3c2  a9 bf    lda #$bf
., f3c4  a4 b9    ldy $b9       ;SA
., f3c6  c0 60    cpy #$60
., f3c8  f0 07    beq $f3d1
., f3ca  a0 00    ldy #$00
., f3cc  a9 02    lda #$02
., f3ce  91 b2    sta ($b2),y   ;TAPE1 - tape buffer
., f3d0  98       tya
., f3d1  85 a6    sta $a6       ;BUFPNT - pointer to tape
                                buffer
., f3d3  18       clc
., f3d4  60       rts
```

62421 SEND SA

This routine exits if there is no secondary address or file name given. The I/O status word, ST is reset, and the serial device commanded to LISTEN. A check is made for a possible ?DEVICE NOT PRESENT error. Finally, the file name is sent to the device.

```
., f3d5  a5 b9    lda $b9       ;SA - current secondary
                                address
., f3d7  30 fa    bmi $f3d3     ;RTS
., f3d9  a4 b7    ldy $b7       ;FNLEN - length of current
                                file name
., f3db  f0 f6    beq $f3d3     ;RTS
., f3dd  a9 00    lda #$00
., f3df  85 90    sta $90       ;STATUS - I/O status word
., f3e1  a5 ba    lda $ba       ;FA - current device number
., f3e3  20 0c ed jsr $ed0c     ;send LISTEN to serial bus
., f3e6  a5 b9    lda $b9       ;SA
., f3e8  09 f0    ora #$f0
., f3ea  20 b9 ed jsr $edb9     ;send LISTEN SA
., f3ed  a5 90    lda $90       ;STATUS
., f3ef  10 05    bpl $f3f6
., f3f1  68       pla
```

```
., f3f2  68        pla
., f3f3  4c 07 f7  jmp $f707     ;I/O error #5 - device not
                                 present
., f3f6  a5 b7     lda $b7       ;FNLEN
., f3f8  f0 0c     beq $f406     ;
., f3fa  a0 00     ldy #$00
., f3fc  b1 bb     lda ($bb),y   ;FNADR - pointer to file name
., f3fe  20 dd ed  jsr $eddd     ;send serial deferred
., f401  c8        iny
., f402  c4 b7     cpy $b7       ;FNLEN
., f404  d0 f6     bne $f3fc     ;send next character in name
., f406  4c 54 f6  jmp $f654     ;
```

62473 OPEN RS-232

The RS-232 port is set to its start values, setting the interrupt mask and data direction register on the CIA. The bit count is computed and the bit time is set up from the timing tables. Note that a different table is used for PAL and NSTC models because of the difference in the clock cycle time (0.98 MHz PAL compared to 1.02 MHz NTSC). Finally the input and output buffers are initialised. See section on RS-232.

```
., f409  20 83 f4  jsr $f483     ;set RS-232 I/O port defaults
., f40c  8c 97 02  sty $0297     ;RSSTAT - 6551 status
                                 register image
., f40f  c4 b7     cpy $b7       ;FNLEN - length of file name
., f411  f0 0a     beq $f41d
., f413  b1 bb     lda ($bb),y   ;FNADR - get file name
., f415  99 93 02  sta $0293,y   ;set 6551 register images
., f418  c8        iny
., f419  c0 04     cpy #$04      ;max 4 characters in
                                 'filename'
., f41b  d0 f2     bne $f40f
., f41d  20 4a ef  jsr $ef4a     ;compute bit count
., f420  8e 98 02  stx $0298     ;BITNUM - # bits left to send
., f423  ad 93 02  lda $0293     ;M51CTR - 6551 control
                                 register image
., f426  29 0f     and #$0f      ;test baud rate given
., f428  f0 1c     beq $f446     ;user rate specified
., f42a  0a        asl
., f42b  aa        tax
., f42c  ad a6 02  lda $02a6     ;PAL/NTSC flag
., f42f  d0 09     bne $f43a     ;PAL
., f431  bc c1 fe  ldy $fec1,x   ;RS-232 timing table - NTSC
., f434  bd c0 fe  lda $fec0,x
., f437  4c 40 f4  jmp $f440     ;set up BPS time from table
., f43a  bc eb e4  ldy $e4eb,x   ;RS-232 timing table - PAL
```

```
., f43d  bd ea e4 lda $e4ea,x
., f440  8c 96 02 sty $0296      ;M51AJB - non standard BPS
                                 time
., f443  8d 95 02 sta $0295
., f446  ad 95 02 lda $0295
., f449  0a       asl
., f44a  20 2e ff jsr $ff2e      ;set baud rate
., f44d  ad 94 02 lda $0294      ;M51CDR - 6551 command
                                 register image
., f450  4a       lsr
., f451  90 09    bcc $f45c
., f453  ad 01 dd lda $dd01      ;RS-232 I/O port
., f456  0a       asl
., f457  b0 03    bcs $f45c
., f459  20 0d f0 jsr $f00d      ;no DSR error
., f45c  ad 9b 02 lda $029b      ;RIDBE - index to end of in
                                 buffer
., f45f  8d 9c 02 sta $029c      ;RIDBS - start page of in
                                 buffer
., f462  ad 9e 02 lda $029e      ;RODBE - index to end of out
                                 buffer
., f465  8d 9d 02 sta $029d      ;RODBS - start page of out
                                 buffer
., f468  20 27 fe jsr $fe27      ;read top of memory
., f46b  a5 f8    lda $f8        ;>RIBUF - in buffer pointer
., f46d  d0 05    bne $f474
., f46f  88       dey
., f470  84 f8    sty $f8        ;RIBUF
., f472  86 f7    stx $f7
., f474  a5 fa    lda $fa        ;>ROBUF - out buffer pointer
., f476  d0 05    bne $f47d
., f478  88       dey
., f479  84 fa    sty $fa
., f47b  86 f9    stx $f9        ;<ROBUF
., f47d  38       sec
., f47e  a9 f0    lda #$f0
., f480  4c 2d fe jmp $fe2d      ;set top of memory
., f483  a9 7f    lda #$7f       ;write interrupt mask to
., f485  8d 0d dd sta $dd0d      ;CIA I.C.R.
., f488  a9 06    lda #$06
., f48a  8d 03 dd sta $dd03      ;data direction register B
., f48d  8d 01 dd sta $dd01      ;set DSR & RTS
., f490  a9 04    lda #$04
., f492  0d 00 dd ora $dd00      ;RS-232 data out port
., f495  8d 00 dd sta $dd00
., f498  a0 00    ldy #$00
., f49a  8c a1 02 sty $02a1      ;ENABL - RS-232 enables
., f49d  60       rts
```

62622 LOAD: LOAD RAM

The KERNAL routine LOAD ($FFD5) is vectored here. If a relocated load is desired (eg. as for LOAD "FRED",8) then the start address is set in MEMUSS. The I/O status word, ST is reset and the device number is tested. If 0 or 3 (ie. keyboard or screen), then ?ILLEGAL DEVICE NUMBER error.

```
., f49e  86 c3     stx $c3      ;MEMUSS - relocated load addr
., f4a0  84 c4     sty $c4
., f4a2  6c 30 03  jmp ($0330)  ;vector ILOAD-points to $F4A5
., f4a5  85 93     sta $93      ;VERCK - load/verify flag
., f4a7  a9 00     lda #$00
., f4a9  85 90     sta $90      ;STATUS - I/O status word
., f4ab  a5 ba     lda $ba      ;FA - current device number
., f4ad  d0 03     bne $f4b2
., f4af  4c 13 f7  jmp $f713    ;I/O error #9 - illegal
                                device number
., f4b2  c9 03     cmp #$03     ;screen?
., f4b4  f0 f9     beq $f4af    ;yes - illegal device
., f4b6  90 7b     bcc $f533    ;load from tape
```

62648 LOAD FROM SERIAL BUS

A filename is assumed by the routine, and if not present, ?MISSING FILENAME is output. The message "SEARCHING" is printed and the filename sent with the TALK command and secondary address to the serial bus. If EOI occurs at this point, then ?FILE NOT FOUND. The message "LOADING" or "VERIFYING" is output and a loop entered, which receives a byte from the serial bus, checks the <STOP> key and either stores it in memory or compares it with memory, depending on the state of VERCK. Finally the bus is UNTALKed.

```
., f4b8  a4 b7     ldy $b7      ;FNLEN - length of filename
., f4ba  d0 03     bne $f4bf
., f4bc  4c 10 f7  jmp $f710    ;I/O error #8 - missing
                                filename
., f4bf  a6 b9     ldx $b9      ;SA - current secondary
                                address
., f4c1  20 af f5  jsr $f5af    ;print "SEARCHING"
., f4c4  a9 60     lda #$60
., f4c6  85 b9     sta $b9      ;SA
., f4c8  20 d5 f3  jsr $f3d5    ;send SA & filename
., f4cb  a5 ba     lda $ba      ;FA - current device number
., f4cd  20 09 ed  jsr $ed09    ;send TALK to serial bus
., f4d0  a5 b9     lda $b9      ;SA
., f4d2  20 c7 ed  jsr $edc7    ;send TALK SA
., f4d5  20 13 ee  jsr $ee13    ;receive from serial bus
```

```
., f4d8  85 ae     sta $ae       ;<EAL - load address
., f4da  a5 90     lda $90       ;STATUS - I/O status word
., f4dc  4a        lsr
., f4dd  4a        lsr
., f4de  b0 50     bcs $f530     ;EOI set - file not found
., f4e0  20 13 ee  jsr $ee13     ;receive from serial bus
., f4e3  85 af     sta $af       ;>EAL
., f4e5  8a        txa
., f4e6  d0 08     bne $f4f0
., f4e8  a5 c3     lda $c3       ;MEMUSS - relocated load
                                 address
., f4ea  85 ae     sta $ae       ;EAL
., f4ec  a5 c4     lda $c4
., f4ee  85 af     sta $af
., f4f0  20 d2 f5  jsr $f5d2     ;print "LOADING/VERIFYING"
., f4f3  a9 fd     lda #$fd
., f4f5  25 90     and $90       ;STATUS
., f4f7  85 90     sta $90
., f4f9  20 e1 ff  jsr $ffe1     ;STOP - scan stop key
., f4fc  d0 03     bne $f501     ;not pressed
., f4fe  4c 33 f6  jmp $f633
., f501  20 13 ee  jsr $ee13     ;receive from serial bus
., f504  aa        tax
., f505  a5 90     lda $90       ;STATUS
., f507  4a        lsr
., f508  4a        lsr
., f509  b0 e8     bcs $f4f3     ;EOI set
., f50b  8a        txa
., f50c  a4 93     ldy $93       ;VERCK
., f50e  f0 0c     beq $f51c     ;load - store byte
., f510  a0 00     ldy #$00
., f512  d1 ae     cmp ($ae),y   ;compare with byte of RAM
., f514  f0 08     beq $f51e
., f516  a9 10     lda #$10      ;flag mismatch
., f518  20 1c fe  jsr $fe1c     ;set status byte
., f51b  2c 91 ae  bit $ae91     ;mask - STA ($AE),Y - store
                                 byte in RAM
., f51e  e6 ae     inc $ae       ;EAL - next address
., f520  d0 02     bne $f524
., f522  e6 af     inc $af       ;>EAL
., f524  24 90     bit $90       ;STATUS
., f526  50 cb     bvc $f4f3     ;get next byte
., f528  20 ef ed  jsr $edef     ;send UNTALK to serial bus
., f52b  20 42 f6  jsr $f642
., f52e  90 79     bcc $f5a9     ;end
., f530  4c 04 f7  jmp $f704     ;I/O error #4 - file not
                                 found
```

62771 LOAD FROM TAPE

A test is made for load from RS-232, which generates ?ILLEGAL DEVICE NUMBER error. Otherwise the messages "PRESS PLAY" and then "SEARCHING" are printed. If a filename was specified, then the specific file header is searched for, otherwise the next header on the tape is searched for. If there is no header then ?FILE NOT FOUND error results. The start and end addresses are read in from the tape buffer and the program is then read from tape. On exit, (A/Y) points to the end address of the program.

```
., f533  4a        lsr
., f534  b0 03     bcs $f539
., f536  4c 13 f7  jmp $f713     ;I/O error #9 - illegal
                                 device number
., f539  20 d0 f7  jsr $f7d0     ;check tape stop
., f53c  b0 03     bcs $f541
., f53e  4c 13 f7  jmp $f713     ;I/O error #9 - illegal
                                 device number
., f541  20 17 f8  jsr $f817     ;print "PRESS PLAY"
., f544  b0 68     bcs $f5ae
., f546  20 af f5  jsr $f5af     ;print "SEARCHING"
., f549  a5 b7     lda $b7       ;FNLEN - length of filename
., f54b  f0 09     beq $f556
., f54d  20 ea f7  jsr $f7ea     ;find specific header
., f550  90 0b     bcc $f55d
., f552  f0 5a     beq $f5ae
., f554  b0 da     bcs $f530
., f556  20 2c f7  jsr $f72c     ;find any tape header
., f559  f0 53     beq $f5ae
., f55b  b0 d3     bcs $f530     ;I/O error #4 - file not
                                 found
., f55d  a5 90     lda $90       ;STATUS - I/O status word
., f55f  29 10     and #$10
., f561  38        sec
., f562  d0 4a     bne $f5ae     ;read error - RTS
., f564  e0 01     cpx #$01      ;secondary address =1?
., f566  f0 11     beq $f579
., f568  e0 03     cpx #$03
., f56a  d0 dd     bne $f549
., f56c  a0 01     ldy #$01
., f56e  b1 b2     lda ($b2),y   ;get character from tape
                                 buffer
., f570  85 c3     sta $c3       ;<MEMUSS - load address
., f572  c8        iny
., f573  b1 b2     lda ($b2),y
., f575  85 c4     sta $c4       ;>MEMUSS
., f577  b0 04     bcs $f57d
```

```
., f579  a5 b9     lda $b9        ;SA - current secondary
                                   address
., f57b  d0 ef     bne $f56c      ;set absolute load address
., f57d  a0 03     ldy #$03
., f57f  b1 b2     lda ($b2),y    ;get from tape buffer
., f581  a0 01     ldy #$01
., f583  f1 b2     sbc ($b2),y
., f585  aa        tax
., f586  a0 04     ldy #$04
., f588  b1 b2     lda ($b2),y
., f58a  a0 02     ldy #$02
., f58c  f1 b2     sbc ($b2),y
., f58e  a8        tay
., f58f  18        clc
., f590  8a        txa
., f591  65 c3     adc $c3        ;<MEMUSS
., f593  85 ae     sta $ae        ;<EAL - end of load address
., f595  98        tya
., f596  65 c4     adc $c4        ;>MEMUSS
., f598  85 af     sta $af        ;>EAL
., f59a  a5 c3     lda $c3
., f59c  85 c1     sta $c1        ;STAL - I/O start address
., f59e  a5 c4     lda $c4
., f5a0  85 c2     sta $c2        ;<STAL
., f5a2  20 d2 f5  jsr $f5d2      ;print "LOADING/VERIFYING"
., f5a5  20 4a f8  jsr $f84a      ;read tape
., f5a8  24 18     bit $18
., f5aa  a6 ae     ldx $ae        ;set (X/Y) = end address
., f5ac  a4 af     ldy $af
., f5ae  60        rts
```

62927 PRINT "SEARCHING..."

If MSGFLG indicates program mode then the message is not printed, otherwise the message "SEARCHING" is printed from the KERNAL I/O message table. If the length of the filename >0 then the message "FOR" is also printed and the routine drops through to print the filename.

```
., f5af  a5 9d     lda $9d        ;MSGFLG - direct or program
                                   mode?
., f5b1  10 1e     bpl $f5d1      ;program - don't print
., f5b3  a0 0c     ldy #$0c
., f5b5  20 2f f1  jsr $f12f      ;print "SEARCHING"
., f5b8  a5 b7     lda $b7        ;FNLEN - length of current
                                   filename
., f5ba  f0 15     beq $f5d1
., f5bc  a0 17     ldy #$17
., f5be  20 2f f1  jsr $f12f      ;print "FOR"
```

62913 PRINT FILENAME

The filename pointed to by FNADR, with length in FNLEN is printed via the KERNAL routine CHROUT.

```
., f5c1  a4 b7     ldy $b7       ;FNLEN - length of current
                                  filename
., f5c3  f0 0c     beq $f5d1
., f5c5  a0 00     ldy #$00
., f5c7  b1 bb     lda ($bb),y   ;FNADR - get character from
                                  name
., f5c9  20 d2 ff  jsr $ffd2     ;CHROUT - output character in
                                  (A)
., f5cc  c8        iny
., f5cd  c4 b7     cpy $b7       ;end of filename?
., f5cf  d0 f6     bne $f5c7     ;no - next character
., f5d1  60        rts
```

62930 PRINT "LOADING/VERIFYING"

The load/verify flag is checked, and the message to be output is flagged according to the result. This message is then printed from the KERNAL I/O messages table.

```
., f5d2  a0 49     ldy #$49      ;flag verify message
., f5d4  a5 93     lda $93       ;VERCK - load/verify flag
., f5d6  f0 02     beq $f5da
., f5d8  a0 59     ldy #$59      ;flag load message
., f5da  4c 2b f1  jmp $f12b     ;print message flagged by (Y)
```

62941 SAVE: SAVE RAM

The KERNAL routine SAVE ($FFD8) jumps to this routine. On entry, (X/Y) must hold the end address +1 of the area of memory to be saved. (A) holds the pointer to the start address of the block, held in zero-page. The current device number is checked to ensure that it is neither keyboard (0) or screen (3). Both of these result in ?ILLEGAL DEVICE NUMBER.

```
., f5dd  86 ae     stx $ae       ;EAL-end address of block +1
., f5df  84 af     sty $af
., f5e1  aa        tax
., f5e2  b5 00     lda $00,x
., f5e4  85 c1     sta $c1       ;STAL - start address of
                                  block
., f5e6  b5 01     lda $01,x
., f5e8  85 c2     sta $c2
```

```
., f5ea  6c 32 03 jmp ($0332)  ;vector ISAVE - points to
                               $F5ED
., f5ed  a5 ba    lda $ba      ;FA - current device number
., f5ef  d0 03    bne $f5f4
., f5f1  4c 13 f7 jmp $f713    ;I/O error #9 - illegal
                               device number
., f5f4  c9 03    cmp #$03     ;screen?
., f5f6  f0 f9    beq $f5f1    ;yes - illegal device
., f5f8  90 5f    bcc $f659    ;save to tape
```

62970 SAVE TO SERIAL BUS

A filename is assumed by the routine, or ?MISSING FILENAME error is called. The serial device is commanded to LISTEN, and the filename sent along with a secondary address of 1. The message "SAVING..." is printed, and a loop sends a byte to the serial bus and checks the <STOP> key until the whole specified block of memory has been saved. Note that the first two bytes to be sent are the start address of the block. Finally, the serial bus is UNLISTENed.

```
., f5fa  a9 61    lda #$61
., f5fc  85 b9    sta $b9      ;SA-set secondary address =1
., f5fe  a4 b7    ldy $b7      ;FNLEN - length of current
                               filename
., f600  d0 03    bne $f605
., f602  4c 10 f7 jmp $f710    ;I/O error #8 - missing
                               filename
., f605  20 d5 f3 jsr $f3d5    ;send SA & filename
., f608  20 8f f6 jsr $f68f    ;print "SAVING"
., f60b  a5 ba    lda $ba      ;FA - current device number
., f60d  20 0c ed jsr $ed0c    ;send 'LISTEN'
., f610  a5 b9    lda $b9      ;SA
., f612  20 b9 ed jsr $edb9    ;send LISTEN SA
., f615  a0 00    ldy #$00
., f617  20 8e fb jsr $fb8e    ;reset pointer
., f61a  a5 ac    lda $ac      ;SAL - holds start address
., f61c  20 dd ed jsr $eddd    ;send serial deferred
., f61f  a5 ad    lda $ad      ;>SAL
., f621  20 dd ed jsr $eddd    ;send serial deferred
., f624  20 d1 fc jsr $fcd1    ;check r/w pointer
., f627  b0 16    bcs $f63f
., f629  b1 ac    lda ($ac),y  ;get character from memory
., f62b  20 dd ed jsr $eddd    ;send serial deferred
., f62e  20 e1 ff jsr $ffe1    ;STOP - test stop key
., f631  d0 07    bne $f63a
., f633  20 42 f6 jsr $f642
., f636  a9 00    lda #$00     ;flag ?BREAK
., f638  38       sec
```

```
., f639  60       rts
., f63a  20 db fc jsr $fcdb     ;bump r/w pointer
., f63d  d0 e5    bne $f624     ;save next byte
., f63f  20 fe ed jsr $edfe     ;send 'UNLISTEN'
., f642  24 b9    bit $b9       ;SA
., f644  30 11    bmi $f657
., f646  a5 ba    lda $ba       ;FA
., f648  20 0c ed jsr $ed0c     ;send 'LISTEN'
., f64b  a5 b9    lda $b9       ;SA
., f64d  29 ef    and #$ef
., f64f  09 e0    ora #$e0
., f651  20 b9 ed jsr $edb9     ;send LISTEN SA
., f654  20 fe ed jsr $edfe     ;send 'UNLISTEN'
., f657  18       clc
., f658  60       rts
```

63065 SAVE TO TAPE

If the device number is 2 (ie RS-232), then ?ILLEGAL DEVICE NUMBER error. The tape buffer is set up and the messages "PRESS RECORD" and "SAVING..." are printed. The tape header is written and the block of memory saved. Finally, if the secondary address was given as 2, the End-Of-Tape header is written.

```
., f659  4a       lsr
., f65a  b0 03    bcs $f65f
., f65c  4c 13 f7 jmp $f713     ;I/O error #9 - illegal
                                device number
., f65f  20 d0 f7 jsr $f7d0     ;get tape buffer address
., f662  90 8d    bcc $f5f1     ;I/O error #9 - illegal
                                device number
., f664  20 38 f8 jsr $f838     ;print "PRESS RECORD"
., f667  b0 25    bcs $f68e     ;RTS
., f669  20 8f f6 jsr $f68f     ;print "SAVING"
., f66c  a2 03    ldx #$03
., f66e  a5 b9    lda $b9       ;SA - current secondary
                                address
., f670  29 01    and #$01      ;1?
., f672  d0 02    bne $f676
., f674  a2 01    ldx #$01
., f676  8a       txa
., f677  20 6a f7 jsr $f76a     ;write tape header
., f67a  b0 12    bcs $f68e
., f67c  20 67 f8 jsr $f867     ;write to tape
., f67f  b0 0d    bcs $f68e
., f681  a5 b9    lda $b9       ;SA
., f683  29 02    and #$02
., f685  f0 06    beq $f68d
```

```
.., f687  a9 05     lda #$05
.., f689  20 6a f7  jsr $f76a      ;write end-of-tape header
.., f68c  24 18     bit $18        ;mask - CLC
.., f68e  60        rts
```

63119 PRINT "SAVING"

MSGFLG is checked, and if direct mode is on, then the message "SAVING" is flagged and printed from the KERNAL I/O messages table.

```
.., f68f  a5 9d     lda $9d
.., f691  10 fb     bpl $f68e
.., f693  a0 51     ldy #$51
.., f695  20 2f f1  jsr $f12f
.., f698  4c c1 f5  jmp $f5c1
```

63131 UDTIM: BUMP CLOCK

The KERNAL routine UDTIM ($FFEA) jumps to this routine. The three byte jiffy clock in RAM is incremented. If it has reached #4F 1A 01, then it is reset to zero. This number represents 5184001 jiffies (each jiffy = 1/60 sec.) or 24 hours + 1 jiffy. Finally, the next routine is used to log the CIA key reading.

```
.., f69b  a2 00     ldx #$00
.., f69d  e6 a2     inc $a2        ;low byte of jiffy clock
.., f69f  d0 06     bne $f6a7
.., f6a1  e6 a1     inc $a1        ;mid byte of jiffy clock
.., f6a3  d0 02     bne $f6a7
.., f6a5  e6 a0     inc $a0        ;high byte of jiffy clock
.., f6a7  38        sec
.., f6a8  a5 a2     lda $a2        ;TIME reached 24 hours yet?
.., f6aa  e9 01     sbc #$01
.., f6ac  a5 a1     lda $a1
.., f6ae  e9 1a     sbc #$1a
.., f6b0  a5 a0     lda $a0
.., f6b2  e9 4f     sbc #$4f
.., f6b4  90 06     bcc $f6bc      ;no
.., f6b6  86 a0     stx $a0        ;reset TIME to zero
.., f6b8  86 a1     stx $a1
.., f6ba  86 a2     stx $a2
```

63164 LOG CIA KEY READING

This routine tests the keyboard for either <STOP> or <RVS> pressed. If so, the keypress is stored in STKEY.

```
., f6bc   ad 01 dc lda $dc01      ;keyboard read register
., f6bf   cd 01 dc cmp $dc01
., f6c2   d0 f8    bne $f6bc      ;wait for value to settle
., f6c4   aa       tax
., f6c5   30 13    bmi $f6da
., f6c7   a2 bd    ldx #$bd
., f6c9   8e 00 dc stx $dc00      ;keyboard write register
., f6cc   ae 01 dc ldx $dc01      ;keyboard read register
., f6cf   ec 01 dc cpx $dc01
., f6d2   d0 f8    bne $f6cc      ;wait for value to settle
., f6d4   8d 00 dc sta $dc00
., f6d7   e8       inx
., f6d8   d0 02    bne $f6dc
., f6da   85 91    sta $91        ;STKEY - flag STOP/RVS key
., f6dc   60       rts
```

63197 RDTIM: GET TIME

The KERNAL routine RDTIM ($FFDE) jumps to this routine. The three byte jiffy clock is read into (A/X/Y) in the format high/mid/low. The routine exits, setting the time to its existing value in the next routine. The clock resolution is 1/60 second. SEI is included since part of the IRQ routine is to update the clock.

```
., f6dd   78       sei
., f6de   a5 a2    lda $a2        ;TIME - real time jiffy clock
., f6e0   a6 a1    ldx $a1
., f6e2   a4 a0    ldy $a0
```

63204 SETTIM: SET TIME

The KERNAL routine SETTIM ($FFDB) jumps to this routine. On entry, (A/X/Y) must hold the value to be stored in the clock. The format is high/mid/low byte. The clock resolution is 1/60 second. SEI is included since part of the IRQ routine is to update the clock.

```
., f6e4   78       sei
., f6e5   85 a2    sta $a2        ;TIME - real time jiffy clock
., f6e7   86 a1    stx $a1
., f6e9   84 a0    sty $a0
., f6eb   58       cli
., f6ec   60       rts
```

63213 STOP: CHECK <STOP> KEY

The KERNAL routine STOP ($FFE1) is vectored here. If STKEY =#7F, then <STOP> was pressed and logged whilst the jiffy

clock was being updated, so all I/O channels are closed and the keyboard buffer reset.

```
., f6ed  a5 91     lda $91      ;STKEY - STOP/RVS key flag
., f6ef  c9 7f     cmp #$7f     ;<STOP>?
., f6f1  d0 07     bne $f6fa    ;no
., f6f3  08        php
., f6f4  20 cc ff  jsr $ffcc    ;CLRCHN - close I/O channels
., f6f7  85 c6     sta $c6      ;NDX - # characters in
                                keyboard buffer
., f6f9  28        plp
., f6fa  60        rts
```

63227 OUTPUT KERNAL ERROR MESSAGES

The error message to be output is flagged into (A) depending on the entry point. I/O channels are closed, and then if KERNAL messages are enabled, "I/O ERROR #" is printed along with the error number.

```
., f6fb  a9 01     lda #$01     ;error 1 - too many files
., f6fd  2c a9 02  bit $02a9    ;mask error 2 - file open
., f700  2c a9 03  bit $03a9    ;mask error 3 - file not open
., f703  2c a9 04  bit $04a9    ;mask error 4 - file not
                                found
., f706  2c a9 05  bit $05a9    ;mask error 5 - device not
                                present
., f709  2c a9 06  bit $06a9    ;mask error 6 - not input
                                file
., f70c  2c a9 07  bit $07a9    ;mask error 7 - not output
                                file
., f70f  2c a9 08  bit $08a9    ;mask error 8 - missing
                                filename
., f712  2c a9 09  bit $09a9    ;mask error 7 - illegal
                                device number
., f715  48        pha
., f716  20 cc ff  jsr $ffcc    ;CLRCHN - close I/O channels
., f719  a0 00     ldy #$00
., f71b  24 9d     bit $9d      ;MSGFLG - KERNAL messages
                                enabled?
., f71d  50 0a     bvc $f729    ;no
., f71f  20 2f f1  jsr $f12f    ;print "I/O ERROR #"
., f722  68        pla
., f723  48        pha
., f724  09 30     ora #$30     ;convert (A) to ASCII number
., f726  20 d2 ff  jsr $ffd2    ;CHROUT - output character in
                                (A)
., f729  68        pla
., f72a  38        sec
```

```
., f72b  60        rts
```

63277 FIND ANY TAPE HEADER

The next block on tape is read into the cassette buffer. The first character is read from the buffer and compared against the valid file types (see below). Assuming the type to be valid and not end-of-tape, then MSGFLG is checked to determine direct or program mode. In direct mode, "FOUND", then the filename, is output. A loop is entered which waits several seconds for either <SPACE> or <CBM> to be pressed. On exit, (A) holds #00 if no header found or end-of-tape, and #01 for a valid header. The header format is as follows:

```
BYTE      CONTENTS                 VALID FILE TYPES:
=====================              =======================
0         file type flag           1,3  program or RAM image
1-2       start address            4    data
3-4       end address              5    end-of-tape
5-21      filename

., f72c  a5 93     lda $93         ;VERCK - load/verify flag
., f72e  48        pha
., f72f  20 41 f8  jsr $f841       ;initiate tape read
., f732  68        pla
., f733  85 93     sta $93         ;VERCK
., f735  b0 32     bcs $f769
., f737  a0 00     ldy #$00
., f739  b1 b2     lda ($b2),y     ;get 1st character from
                                   buffer
., f73b  c9 05     cmp #$05        ;end-of-tape?
., f73d  f0 2a     beq $f769       ;yes - RTS
., f73f  c9 01     cmp #$01        ;program?
., f741  f0 08     beq $f74b
., f743  c9 03     cmp #$03        ;program?
., f745  f0 04     beq $f74b
., f747  c9 04     cmp #$04        ;data file?
., f749  d0 e1     bne $f72c
., f74b  aa        tax
., f74c  24 9d     bit $9d         ;MSGFLG - program/direct
                                   mode?
., f74e  10 17     bpl $f767       ;program - don't print
., f750  a0 63     ldy #$63
., f752  20 2f f1  jsr $f12f       ;print message "FOUND"
., f755  a0 05     ldy #$05
., f757  b1 b2     lda ($b2),y     ;get character in filename
., f759  20 d2 ff  jsr $ffd2       ;CHROUT - output character in
                                   (A)
```

```
., f75c  c8        iny
., f75d  c0 15     cpy #$15
., f75f  d0 f6     bne $f757     ;get next character in
                                 filename
., f761  a5 a1     lda $a1       ;TIME - mid byte of jiffy
                                 clock
., f763  20 e0 e4  jsr $e4e0     ;delay or keypress
., f766  ea        nop
., f767  18        clc
., f768  88        dey
., f769  60        rts
```

63338 WRITE TAPE HEADER

STAL and EAL are pushed onto the stack. The cassette buffer is then cleared by filling it with spaces (#20). The header type code (see notes for previous routine), start address and end address are then written to the buffer. If a filename was specified then this is also written to the buffer. Finally the start/end pointers to the cassette buffer are set up and the contents of the buffer written to tape along with a 10 second leader tone (50+ cycles of short pulses - see $FBA6). EAL and STAL are pulled from the stack.

```
., f76a  85 9e     sta $9e       ;PTR1 - holds type of header
                                 flag
., f76c  20 d0 f7  jsr $f7d0     ;get buffer address
., f76f  90 5e     bcc $f7cf
., f771  a5 c2     lda $c2       ;push STAL - I/O start
                                 address
., f773  48        pha
., f774  a5 c1     lda $c1
., f776  48        pha
., f777  a5 af     lda $af       ;push EAL - tape end address
., f779  48        pha
., f77a  a5 ae     lda $ae
., f77c  48        pha
., f77d  a0 bf     ldy #$bf
., f77f  a9 20     lda #$20      ;ASCII space
., f781  91 b2     sta ($b2),y   ;fill cassette buffer with
                                 spaces
., f783  88        dey
., f784  d0 fb     bne $f781
., f786  a5 9e     lda $9e       ;PTR1 - tape pass 1 error log
., f788  91 b2     sta ($b2),y   ;write header type to buffer
., f78a  c8        iny
., f78b  a5 c1     lda $c1       ;STAL I/O start address
., f78d  91 b2     sta ($b2),y   ;write start address to
                                 buffer
```

```
., f78f  c8        iny
., f790  a5 c2     lda $c2
., f792  91 b2     sta ($b2),y
., f794  c8        iny
., f795  a5 ae     lda $ae        ;EAL - tape end address
., f797  91 b2     sta ($b2),y    ;write end address to buffer
., f799  c8        iny
., f79a  a5 af     lda $af
., f79c  91 b2     sta ($b2),y
., f79e  c8        iny
., f79f  84 9f     sty $9f        ;PTR2 - holds write to buffer
                                  pointer
., f7a1  a0 00     ldy #$00
., f7a3  84 9e     sty $9e        ;PTR1
., f7a5  a4 9e     ldy $9e
., f7a7  c4 b7     cpy $b7        ;FNLEN - length of filename
., f7a9  f0 0c     beq $f7b7      ;no name specified
., f7ab  b1 bb     lda ($bb),y    ;get character from filename
., f7ad  a4 9f     ldy $9f        ;PTR2
., f7af  91 b2     sta ($b2),y    ;store character in cassette
                                  buffer
., f7b1  e6 9e     inc $9e
., f7b3  e6 9f     inc $9f
., f7b5  d0 ee     bne $f7a5      ;write next character in name
., f7b7  20 d7 f7  jsr $f7d7      ;set buffer start/end
                                  pointers
., f7ba  a9 69     lda #$69
., f7bc  85 ab     sta $ab        ;RIPRTY - cassette short
                                  count
., f7be  20 6b f8  jsr $f86b      ;write buffer to tape
., f7c1  a8        tay
., f7c2  68        pla
., f7c3  85 ae     sta $ae        ;pull EAL
., f7c5  68        pla
., f7c6  85 af     sta $af
., f7c8  68        pla
., f7c9  85 c1     sta $c1        ;pull STAL
., f7cb  68        pla
., f7cc  85 c2     sta $c2
., f7ce  98        tya
., f7cf  60        rts
```

63440 GET BUFFER ADDRESS

This routine places the start address of the cassette buffer in (X/Y). It then compares the start page of the buffer with page 2.

```
., f7d0  a6 b2     ldx $b2        ;TAPE1 - start of tape buffer
```

```
., f7d2  a4 b3    ldy $b3
., f7d4  c0 02    cpy #$02
., f7d6  60       rts
```

63447 SET BUFFER START/END POINTERS

The start address of the buffer ($033C) is placed in STAL and the end address ($03FB) in EAL.

```
., f7d7  20 d0 f7 jsr $f7d0     ;get buffer address
., f7da  8a       txa
., f7db  85 c1    sta $c1       ;<STAL - I/O start address
., f7dd  18       clc
., f7de  69 c0    adc #$c0
., f7e0  85 ae    sta $ae       ;EAL - tape end address
., f7e2  98       tya
., f7e3  85 c2    sta $c2       ;>STAL
., f7e5  69 00    adc #$00
., f7e7  85 af    sta $af       ;>EAL
., f7e9  60       rts
```

63466 FIND A SPECIFIC TAPE HEADER

The next header on the tape is searched for. Once it is found, the name in the header is compared with the specific filename given (the name starts at byte 5 in the buffer). If the names do not match, then the next header is sought, and so on, until either end-of-tape or the tape runs out. If a match is found then on exit, (Y) holds the length of the filename.

```
., f7ea  20 2c f7 jsr $f72c     ;find next tape header
., f7ed  b0 1d    bcs $f80c     ;none found - RTS
., f7ef  a0 05    ldy #$05
., f7f1  84 9f    sty $9f       ;PTR2 - points to filename in
                                buffer
., f7f3  a0 00    ldy #$00
., f7f5  84 9e    sty $9e       ;PTR1 - holds length of
                                filename
., f7f7  c4 b7    cpy $b7       ;FNLEN - length of filename
., f7f9  f0 10    beq $f80b     ;end of filename - RTS
., f7fb  b1 bb    lda ($bb),y   ;get character from specified
                                name
., f7fd  a4 9f    ldy $9f       ;PTR2
., f7ff  d1 b2    cmp ($b2),y   ;compare with name in buffer
., f801  d0 e7    bne $f7ea     ;different - find next header
., f803  e6 9e    inc $9e       ;PTR1
., f805  e6 9f    inc $9f       ;PTR2
., f807  a4 9e    ldy $9e
```

```
., f809  d0 ec     bne $f7f7    ;check next character in name
., f80b  18        clc
., f80c  60        rts
```

63501 BUMP TAPE POINTER

The pointer to the next free byte in the tape buffer is incremented and then checked to see if it has reached the end of the buffer.

```
., f80d  20 d0 f7 jsr $f7d0    ;get buffer address
., f810  e6 a6     inc $a6      ;BUFPNT - pointer to tape
                                buffer
., f812  a4 a6     ldy $a6
., f814  c0 c0     cpy #$c0     ;end of buffer?
., f816  60        rts
```

63511 PRINT "PRESS PLAY"

The tape status is checked, and if <play> is not pressed on the cassette deck then the message "PRESS PLAY ON TAPE" is printed. A loop waits until the <play> key is pressed and then prints the message "OK".

```
., f817  20 2e f8 jsr $f82e    ;check tape status
., f81a  f0 1a     beq $f836    ;key already pressed - RTS
., f81c  a0 1b     ldy #$1b
., f81e  20 2f f1 jsr $f12f    ;print "PRESS PLAY ON TAPE"
., f821  20 d0 f8 jsr $f8d0    ;get buffer address
., f824  20 2e f8 jsr $f82e    ;check tape status
., f827  d0 f8     bne $f821    ;key not pressed - wait
., f829  a0 6a     ldy #$6a
., f82b  4c 2f f1 jmp $f12f    ;print message "OK"
```

63534 CHECK TAPE STATUS

The 6510 onboard I/O port is checked to see if <play> is pressed on the cassette deck (ie bit 4 =1). If so, then Z is set, otherwise it is cleared.

```
., f82e  a9 10     lda #$10
., f830  24 01     bit $01      ;R6510 - microprocessor I/O
                                port
., f832  d0 02     bne $f836    ;<play> pressed
., f834  24 01     bit $01      ;R6510
., f836  18        clc
., f837  60        rts
```

63544 PRINT "PRESS RECORD"

The status of the cassette deck is checked, and if no keys are pressed then the message "PRESS RECORD & PLAY ON TAPE" is flagged. The message is then processed by the 'press play' routine.

```
., f838  20 2e f8 jsr $f82e    ;check tape status
., f83b  f0 f9    beq $f836    ;key pressed - RTS
., f83d  a0 2e    ldy #$2e     ;flag "PRESS RECORD & PLAY ON
                                TAPE"
., f83f  d0 dd    bne $f81e    ;print message & wait for key
```

63553 INITIATE TAPE READ

STATUS is reset and VERCK flagged to load. STAL and EAL are set to point to the start and end of the cassette buffer. These last two pointers can be preset to any load address if the routine is entered at $F84A. A series of temporary values are set to zero and the read I.C.R. mask and IRQ vector flagged. Finally, the common tape code is entered.

```
., f841  a9 00    lda #$00
., f843  85 90    sta $90      ;STATUS - I/O status word
., f845  85 93    sta $93      ;VERCK - load/verify flag
., f847  20 d7 f7 jsr $f7d7    ;set buffer start/end
                                pointers
., f84a  20 17 f8 jsr $f817    ;print "PRESS PLAY"
., f84d  b0 1f    bcs $f86e    ;end
., f84f  78       sei
., f850  a9 00    lda #$00
., f852  85 aa    sta $aa      ;RIDATA - sync countdown
., f854  85 b4    sta $b4      ;BITTS - byte sync flag
., f856  85 b0    sta $b0      ;<CMP0
., f858  85 9e    sta $9e      ;PTR1 - tape pass 1 error log
., f85a  85 9f    sta $9f      ;PTR2 - tape pass 2 error log
., f85c  85 9c    sta $9c      ;DPSW - tape dipole switch
., f85e  a9 90    lda #$90     ;interrupt mask byte
., f860  a2 0e    ldx #$0e     ;IRQ vector offset
., f862  d0 11    bne $f875    ;do common tape code
```

63588 INITIATE TAPE WRITE

The start and end pointers to the cassette buffer are set up, and the message "PRESS RECORD..." is printed. The correct I.C.R. mask for write is flagged, as is the tape write IRQ vector. Finally, the common tape code is executed.

```
., f864  20 d7 f7 jsr $f7d7    ;set buffer start/end
                                pointers
```

```
., f867  a9 14    lda #$14
., f869  85 ab    sta $ab       ;RIPRTY - cassette short
                                count
., f86b  20 38 f8 jsr $f838     ;print "PRESS RECORD"
., f86e  b0 6c    bcs $f8dc     ;RTS
., f870  78       sei
., f871  a9 82    lda #$82      ;interrupt mask byte
., f873  a2 08    ldx #$08      ;IRQ vector offset
```

63605 COMMON TAPE CODE

On entry, (X) holds the offset for the tape I/O IRQ vector table and (A) the CIA interrupt mask byte. The mask is written and the CIA control registers are set up. The screen is blanked and the normal IRQ vector is stored and replaced by the tape IRQ vector ($FC6A for write and $F92C for read). These new interrupts will only come into force once the SEI mask is cleared. The cassette motor is switched on and there is a delay loop of 1/3 second to allow the motor to pick up speed. Finally the interrupt is enabled and a loop entered which waits for the normal IRQ vector to be restored. <STOP> is tested for and the clock updated by this loop.

```
., f875  a0 7f    ldy #$7f
., f877  8c 0d dc sty $dc0d     ;initialise CIA I.C.R.
., f87a  8d 0d dc sta $dc0d     ;write given I.C.R. mask
., f87d  ad 0e dc lda $dc0e     ;set C.R. A
., f880  09 19    ora #$19
., f882  8d 0f dc sta $dc0f     ;set C.R. B
., f885  29 91    and #$91
., f887  8d a2 02 sta $02a2     ;TOD sense during tape I/O
., f88a  20 a4 f0 jsr $f0a4     ;check serial bus idle
., f88d  ad 11 d0 lda $d011     ;VIC II control register
., f890  29 ef    and #$ef
., f892  8d 11 d0 sta $d011     ;blank screen to border
                                colour
., f895  ad 14 03 lda $0314     ;CINV - hardware IRQ vector
., f898  8d 9f 02 sta $029f     ;IRQTMP - store for normal
                                IRQ vector
., f89b  ad 15 03 lda $0315
., f89e  8d a0 02 sta $02a0
., f8a1  20 bd fc jsr $fcbd     ;set tape IRQ vector
., f8a4  a9 02    lda #$02
., f8a6  85 be    sta $be       ;FSBLK - read/write block
                                count
., f8a8  20 97 fb jsr $fb97     ;new character setup
., f8ab  a5 01    lda $01       ;R6510 - onboard I/O port
```

```
., f8ad  29 1f     and #$1f
., f8af  85 01     sta $01       ;switch on cassette motor
., f8b1  85 c0     sta $c0       ;CAS1 - tape motor interlock
., f8b3  a2 ff     ldx #$ff
., f8b5  a0 ff     ldy #$ff
., f8b7  88        dey
., f8b8  d0 fd     bne $f8b7     ;delay part 1
., f8ba  ca        dex
., f8bb  d0 f8     bne $f8b5     ;delay part 2 - total delay =
                                  1/3 sec
., f8bd  58        cli
., f8be  ad a0 02 lda $02a0      ;>IRQTMP
., f8c1  cd 15 03 cmp $0315      ;>CINV
., f8c4  18        clc
., f8c5  f0 15     beq $f8dc     ;normal IRQ restored - end
., f8c7  20 d0 f8 jsr $f8d0      ;check tape stop
., f8ca  20 bc f6 jsr $f6bc      ;log CIA key reading
., f8cd  4c be f8 jmp $f8be      ;compare IRQ vectors
```

63696 CHECK TAPE STOP

This routine checks to see if <STOP> has been pressed and logged. If it has, then the normal IRQ vector is restored and >IRQTMP set to zero.

```
., f8d0  20 e1 ff jsr $ffe1      ;STOP - scan <STOP> key &
                                  update clock
., f8d3  18        clc
., f8d4  d0 0b     bne $f8e1     ;key not pressed
., f8d6  20 93 fc jsr $fc93      ;restore normal IRQ vector
., f8d9  38        sec
., f8da  68        pla
., f8db  68        pla
., f8dc  a9 00     lda #$00
., f8de  8d a0 02 sta $02a0      ;>IRQTMP - temp store for IRQ
                                  vector
., f8e1  60        rts
```

63714 SET READ TIMING

This routine sets CIA timer A to a new value, synchronised with CIA timer B. The value set into timer A is derived from the tape timing constant CMP0 and (X).

```
., f8e2  86 b1     stx $b1       ;CMP0 - holds dipole time
., f8e4  a5 b0     lda $b0       ;tape timing constant
., f8e6  0a        asl
., f8e7  0a        asl
., f8e8  18        clc
```

```
., f8e9  65 b0     adc $b0       ;CMP0
., f8eb  18        clc
., f8ec  65 b1     adc $b1
., f8ee  85 b1     sta $b1
., f8f0  a9 00     lda #$00
., f8f2  24 b0     bit $b0
., f8f4  30 01     bmi $f8f7
., f8f6  2a        rol
., f8f7  06 b1     asl $b1
., f8f9  2a        rol
., f8fa  06 b1     asl $b1
., f8fc  2a        rol
., f8fd  aa        tax
., f8fe  ad 06 dc  lda $dc06     ;timer B low byte
., f901  c9 16     cmp #$16
., f903  90 f9     bcc $f8fe
., f905  65 b1     adc $b1       ;CMP0
., f907  8d 04 dc  sta $dc04     ;timer A low byte
., f90a  8a        txa
., f90b  6d 07 dc  adc $dc07     ;timer B high byte
., f90e  8d 05 dc  sta $dc05     ;timer A high byte
., f911  ad a2 02  lda $02a2     ;TOD sense during tape I/O
., f914  8d 0e dc  sta $dc0e     ;CIA C.R. A
., f917  8d a4 02  sta $02a4     ;temp D1IRQ indicator
., f91a  ad 0d dc  lda $dc0d     ;CIA I.C.R.
., f91d  29 10     and #$10      ;tape read IRQ?
., f91f  f0 09     beq $f92a
., f921  a9 f9     lda #$f9
., f923  48        pha
., f924  a9 2a     lda #$2a
., f926  48        pha
., f927  4c 43 ff  jmp $ff43     ;do IRQ routine
., f92a  58        cli
., f92b  60        rts
```

63788 READ TAPE BITS

This is the interrupt routine for the vector offset #0E. Although it is a long routine, its function can be described quite simply. Before the byte is read in, there is a delay timed with the CIA timers. Timer B is then set to 65 milliseconds and the tape read. The byte is read in bit by bit, each bit being rotated into MYCH. Once the read is completed, DPSW is set and the interrupt exited.

```
., f92c  ae 07 dc  ldx $dc07     ;timer B high byte
., f92f  a0 ff     ldy #$ff
., f931  98        tya
., f932  ed 06 dc  sbc $dc06     ;timer A high byte
```

```
., f935  ec 07 dc cpx $dc07
., f938  d0 f2    bne $f92c
., f93a  86 b1    stx $b1       ;>CMP0 - tape timing constant
., f93c  aa       tax
., f93d  8c 06 dc sty $dc06     ;timer A high
., f940  8c 07 dc sty $dc07     ;timer B high
., f943  a9 19    lda #$19
., f945  8d 0f dc sta $dc0f     ;CIA C.R. B - set & start
                                timer B
., f948  ad 0d dc lda $dc0d     ;CIA I.C.R.
., f94b  8d a3 02 sta $02a3     ;temp store
., f94e  98       tya
., f94f  e5 b1    sbc $b1       ;>CMP0
., f951  86 b1    stx $b1
., f953  4a       lsr
., f954  66 b1    ror $b1       ;>CMP0
., f956  4a       lsr
., f957  66 b1    ror $b1
., f959  a5 b0    lda $b0       ;<CMP0
., f95b  18       clc
., f95c  69 3c    adc #$3c
., f95e  c5 b1    cmp $b1
., f960  b0 4a    bcs $f9ac
., f962  a6 9c    ldx $9c       ;DPSW - byte received flag
., f964  f0 03    beq $f969     ;byte not received
., f966  4c 60 fa jmp $fa60     ;store characters in RAM
                                buffer
., f969  a6 a3    ldx $a3       ;temp data area
., f96b  30 1b    bmi $f988
., f96d  a2 00    ldx #$00
., f96f  69 30    adc #$30
., f971  65 b0    adc $b0       ;<CMP0
., f973  c5 b1    cmp $b1
., f975  b0 1c    bcs $f993
., f977  e8       inx
., f978  69 26    adc #$26
., f97a  65 b0    adc $b0       ;<CMP0
., f97c  c5 b1    cmp $b1
., f97e  b0 17    bcs $f997
., f980  69 2c    adc #$2c
., f982  65 b0    adc $b0       ;CMP0
., f984  c5 b1    cmp $b1
., f986  90 03    bcc $f98b
., f988  4c 10 fa jmp $fa10
., f98b  a5 b4    lda $b4       ;BITTS - byte sync flag
., f98d  f0 1d    beq $f9ac     ;no sync error
., f98f  85 a8    sta $a8       ;BITC1 - flag read error
., f991  d0 19    bne $f9ac
., f993  e6 a9    inc $a9       ;RINONE - count of zeros
```

```
., f995  b0 02     bcs $f999
., f997  c6 a9     dec $a9       ;RINONE
., f999  38        sec
., f99a  e9 13     sbc #$13
., f99c  e5 b1     sbc $b1       ;>CMP0
., f99e  65 92     adc $92       ;SVXT - tape timing constant
., f9a0  85 92     sta $92
., f9a2  a5 a4     lda $a4       ;temp data area
., f9a4  49 01     eor #$01
., f9a6  85 a4     sta $a4
., f9a8  f0 2b     beq $f9d5
., f9aa  86 d7     stx $d7       ;temp data area
., f9ac  a5 b4     lda $b4       ;BITTS - byte sync flag
., f9ae  f0 22     beq $f9d2     ;no error - exit interrupt
., f9b0  ad a3 02  lda $02a3     ;temp store for cassette read
., f9b3  29 01     and #$01
., f9b5  d0 05     bne $f9bc
., f9b7  ad a4 02  lda $02a4     ;temp D1IRQ indicator
., f9ba  d0 16     bne $f9d2
., f9bc  a9 00     lda #$00
., f9be  85 a4     sta $a4       ;tape cycle count
., f9c0  8d a4 02  sta $02a4     ;temp D1IRQ indicator
., f9c3  a5 a3     lda $a3       ;temp store for cassette read
., f9c5  10 30     bpl $f9f7
., f9c7  30 bf     bmi $f988
., f9c9  a2 a6     ldx #$a6      ;read timing value
., f9cb  20 e2 f8  jsr $f8e2     ;set read timing for next
                                 dipole
., f9ce  a5 9b     lda $9b       ;PRTY - tape character parity
., f9d0  d0 b9     bne $f98b
., f9d2  4c bc fe  jmp $febc     ;exit interrupt
., f9d5  a5 92     lda $92       ;SVXT - tape timing constant
., f9d7  f0 07     beq $f9e0
., f9d9  30 03     bmi $f9de
., f9db  c6 b0     dec $b0       ;<CMP0
., f9dd  2c e6 b0  bit $b0e6     ;mask - INC $B0
., f9e0  a9 00     lda #$00
., f9e2  85 92     sta $92       ;SVXT
., f9e4  e4 d7     cpx $d7       ;current dipole bit value
., f9e6  d0 0f     bne $f9f7
., f9e8  8a        txa
., f9e9  d0 a0     bne $f98b
., f9eb  a5 a9     lda $a9       ;RINONE - counts zeros
., f9ed  30 bd     bmi $f9ac
., f9ef  c9 10     cmp #$10
., f9f1  90 b9     bcc $f9ac
., f9f3  85 96     sta $96       ;SYNO - cassette sync number
., f9f5  b0 b5     bcs $f9ac
., f9f7  8a        txa
```

```
., f9f8  45 9b     eor $9b       ;PRTY - tape parity bit
., f9fa  85 9b     sta $9b
., f9fc  a5 b4     lda $b4       ;BITTS - byte sync flag
., f9fe  f0 d2     beq $f9d2     ;no sync error - exit
                                 interrupt
., fa00  c6 a3     dec $a3       ;serial bit count
., fa02  30 c5     bmi $f9c9     ;set read timing for next
                                 dipole
., fa04  46 d7     lsr $d7       ;latest dipole bit value
., fa06  66 bf     ror $bf       ;MYCH - serial word buffer
., fa08  a2 da     ldx #$da      ;read timing value
., fa0a  20 e2 f8  jsr $f8e2     ;set read timing for next
                                 dipole
., fa0d  4c bc fe  jmp $febc     ;exit interrupt
., fa10  a5 96     lda $96       ;SYNO - cassette sync no
., fa12  f0 04     beq $fa18
., fa14  a5 b4     lda $b4       ;BITTS - byte sync flag
., fa16  f0 07     beq $fa1f
., fa18  a5 a3     lda $a3       ;serial bit count
., fa1a  30 03     bmi $fa1f
., fa1c  4c 97 f9  jmp $f997     ;read next bit
., fa1f  46 b1     lsr $b1       ;CMPO
., fa21  a9 93     lda #$93
., fa23  38        sec
., fa24  e5 b1     sbc $b1       ;CMPO
., fa26  65 b0     adc $b0
., fa28  0a        asl
., fa29  aa        tax
., fa2a  20 e2 f8  jsr $f8e2     ;set read timing
., fa2d  e6 9c     inc $9c       ;DPSW - byte received flag
., fa2f  a5 b4     lda $b4       ;BITTS
., fa31  d0 11     bne $fa44
., fa33  a5 96     lda $96       ;SYNO - block sync flag
., fa35  f0 26     beq $fa5d     ;end of block - exit
                                 interrupt
., fa37  85 a8     sta $a8       ;BITC1 - flag read error
., fa39  a9 00     lda #$00
., fa3b  85 96     sta $96       ;SYNO - block sync flag
., fa3d  a9 81     lda #$81
., fa3f  8d 0d dc  sta $dc0d     ;CIA I.C.R.
., fa42  85 b4     sta $b4       ;BITTS
., fa44  a5 96     lda $96       ;SYNO
., fa46  85 b5     sta $b5       ;NXTBIT - tape EOT flag
., fa48  f0 09     beq $fa53
., fa4a  a9 00     lda #$00
., fa4c  85 b4     sta $b4       ;BITTS - byte sync flag
., fa4e  a9 01     lda #$01
., fa50  8d 0d dc  sta $dc0d     ;CIA I.C.R.
., fa53  a5 bf     lda $bf       ;MYCH
```

```
.. fa55  85 bd     sta $bd      ;ROPRTY - recieve input
                                character
.. fa57  a5 a8     lda $a8      ;BITC1 - read error flag
.. fa59  05 a9     ora $a9      ;RINONE - count zeros
.. fa5b  85 b6     sta $b6      ;RODATA - combined errors
                                flag
.. fa5d  4c bc fe  jmp $febc    ;exit interrupt
```

64096 STORE TAPE CHARACTERS IN RAM

The character read from the cassette player is stored in RAM or verified against RAM, depending on the state of VERCK. Tests are also made for the following error conditions: LONG BLOCK (ST = #08), SHORT BLOCK (ST = #04), UNRECOVERABLE READ ERROR (ST = #10) and CHECKSUM ERROR (ST= 20). Where an error occurs, its position is flagged into the tape error log at the low end of the stack, and also in PTR1. The errors are then rechecked on pass 2 of the data. Finally the r/w pointer is bumped and the interrupt exited.

```
.., fa60  20 97 fb  jsr $fb97    ;new character setup
.., fa63  85 9c     sta $9c      ;DPSW - byte recieved flag
.., fa65  a2 da     ldx #$da
.., fa67  20 e2 f8  jsr $f8e2    ;set read timing
.., fa6a  a5 be     lda $be      ;FSBLK - cassette r/w block
                                 count
.., fa6c  f0 02     beq $fa70
.., fa6e  85 a7     sta $a7      ;INBIT - tape short count
.., fa70  a9 0f     lda #$0f
.., fa72  24 aa     bit $aa      ;RIDATA - function mode
.., fa74  10 17     bpl $fa8d
.., fa76  a5 b5     lda $b5      ;NXTBIT - tape EOT flag
.., fa78  d0 0c     bne $fa86
.., fa7a  a6 be     ldx $be      ;FSBLK - block indicator
.., fa7c  ca        dex
.., fa7d  d0 0b     bne $fa8a    ;exit interrupt
.., fa7f  a9 08     lda #$08     ;flag long block error
.., fa81  20 1c fe  jsr $fe1c    ;set STATUS
.., fa84  d0 04     bne $fa8a    ;exit interrupt
.., fa86  a9 00     lda #$00
.., fa88  85 aa     sta $aa      ;RIDATA - sync countdown
.., fa8a  4c bc fe  jmp $febc    ;exit interrupt
.., fa8d  70 31     bvs $fac0    ;load byte
.., fa8f  d0 18     bne $faa9
.., fa91  a5 b5     lda $b5      ;NXTBIT
.., fa93  d0 f5     bne $fa8a    ;exit interrupt
.., fa95  a5 b6     lda $b6      ;RODATA - combined error
                                 values
.., fa97  d0 f1     bne $fa8a    ;error - exit interrupt
```

```
., fa99  a5 a7     lda $a7       ;INBIT
., fa9b  4a        lsr
., fa9c  a5 bd     lda $bd       ;ROPRTY - receive input
                                 character
., fa9e  30 03     bmi $faa3
., faa0  90 18     bcc $faba
., faa2  18        clc
., faa3  b0 15     bcs $faba
., faa5  29 0f     and #$0f
., faa7  85 aa     sta $aa       ;RIDATA - sync countdown
., faa9  c6 aa     dec $aa
., faab  d0 dd     bne $fa8a     ;exit interrupt
., faad  a9 40     lda #$40
., faaf  85 aa     sta $aa       ;RIDATA
., fab1  20 8e fb  jsr $fb8e     ;reset pointer
., fab4  a9 00     lda #$00
., fab6  85 ab     sta $ab       ;RIPRTY - cassette short
                                 count
., fab8  f0 d0     beq $fa8a     ;exit interrupt
., faba  a9 80     lda #$80
., fabc  85 aa     sta $aa       ;RIDATA
., fabe  d0 ca     bne $fa8a     ;exit interrupt
., fac0  a5 b5     lda $b5       ;NXTBIT
., fac2  f0 0a     beq $face
., fac4  a9 04     lda #$04      ;flag short block error
., fac6  20 1c fe  jsr $fe1c     ;set STATUS
., fac9  a9 00     lda #$00
., facb  4c 4a fb  jmp $fb4a
., face  20 d1 fc  jsr $fcd1     ;check r/w pointer
., fad1  90 03     bcc $fad6
., fad3  4c 48 fb  jmp $fb48
., fad6  a6 a7     ldx $a7       ;INBIT - tape short count
., fad8  ca        dex
., fad9  f0 2d     beq $fb08
., fadb  a5 93     lda $93       ;VERCK - load/verify flag
., fadd  f0 0c     beq $faeb
., fadf  a0 00     ldy #$00
., fae1  a5 bd     lda $bd       ;ROPRTY - receive input
                                 character
., fae3  d1 ac     cmp ($ac),y   ;compare byte with RAM
., fae5  f0 04     beq $faeb     ;byte matches
., fae7  a9 01     lda #$01
., fae9  85 b6     sta $b6       ;RODATA - combined error
                                 value
., faeb  a5 b6     lda $b6
., faed  f0 4b     beq $fb3a
., faef  a2 3d     ldx #$3d
., faf1  e4 9e     cpx $9e       ;PTR1 - tape pass 1 error log
., faf3  90 3e     bcc $fb33     ;unrecoverable read error
```

```
.,  faf5   a6 9e      ldx $9e       ;PTR1
.,  faf7   a5 ad      lda $ad       ;>SAL - tape buffer pointer
.,  faf9   9d 01 01 sta $0101,x   ;<BAD - tape input error log
.,  fafc   a5 ac      lda $ac       ;<SAL
.,  fafe   9d 00 01 sta $0100,x   ;<BAD
.,  fb01   e8         inx
.,  fb02   e8         inx
.,  fb03   86 9e      stx $9e       ;PTR1
.,  fb05   4c 3a fb jmp $fb3a
.,  fb08   a6 9f      ldx $9f       ;PTR2 - tape pass 2 error log
.,  fb0a   e4 9e      cpx $9e       ;PTR1
.,  fb0c   f0 35      beq $fb43     ;bump pointer
.,  fb0e   a5 ac      lda $ac       ;<SAL
.,  fb10   dd 00 01 cmp $0100,x   ;<BAD
.,  fb13   d0 2e      bne $fb43     ;bump pointer
.,  fb15   a5 ad      lda $ad       ;>SAL
.,  fb17   dd 01 01 cmp $0101,x   ;>BAD
.,  fb1a   d0 27      bne $fb43     ;bump pointer
.,  fb1c   e6 9f      inc $9f       ;PTR2
.,  fb1e   e6 9f      inc $9f
.,  fb20   a5 93      lda $93       ;VERCK
.,  fb22   f0 0b      beq $fb2f
.,  fb24   a5 bd      lda $bd       ;ROPRTY
.,  fb26   a0 00      ldy #$00
.,  fb28   d1 ac      cmp ($ac),y   ;compare byte with RAM
.,  fb2a   f0 17      beq $fb43     ;match - bump pointer
.,  fb2c   c8         iny
.,  fb2d   84 b6      sty $b6       ;RODATA
.,  fb2f   a5 b6      lda $b6
.,  fb31   f0 07      beq $fb3a
.,  fb33   a9 10      lda #$10      ;flag unrecoverable read
                                    error
.,  fb35   20 1c fe jsr $fe1c     ;set STATUS
.,  fb38   d0 09      bne $fb43     ;bump pointer
.,  fb3a   a5 93      lda $93       ;VERCK
.,  fb3c   d0 05      bne $fb43
.,  fb3e   a8         tay
.,  fb3f   a5 bd      lda $bd       ;ROPRTY - receive input
                                    character
.,  fb41   91 ac      sta ($ac),y   ;store character in RAM
.,  fb43   20 db fc jsr $fcdb     ;bump r/w pointer
.,  fb46   d0 43      bne $fb8b     ;exit interrupt
.,  fb48   a9 80      lda #$80      ;flag - ignore bytes until
                                    RINONE set
.,  fb4a   85 aa      sta $aa       ;RIDATA - function mode
.,  fb4c   78         sei
.,  fb4d   a2 01      ldx #$01
.,  fb4f   8e 0d dc stx $dc0d     ;CIA I.C.R.
.,  fb52   ae 0d dc ldx $dc0d
```

```
.. fb55  a6 be     ldx $be       ;FSBLK - current block #
.. fb57  ca        dex
.. fb58  30 02     bmi $fb5c
.. fb5a  86 be     stx $be       ;FSBLK
.. fb5c  c6 a7     dec $a7       ;INBIT - tape short count
.. fb5e  f0 08     beq $fb68
.. fb60  a5 9e     lda $9e       ;PTR1
.. fb62  d0 27     bne $fb8b     ;exit interrupt
.. fb64  85 be     sta $be       ;FSBLK
.. fb66  f0 23     beq $fb8b     ;exit interrupt
.. fb68  20 93 fc  jsr $fc93     ;restore normal IRQ vector
.. fb6b  20 8e fb  jsr $fb8e     ;reset pointer
.. fb6e  a0 00     ldy #$00
.. fb70  84 ab     sty $ab       ;RIPRTY - cassette short
                                 count
.. fb72  b1 ac     lda ($ac),y   ;get stored character from
                                 RAM
.. fb74  45 ab     eor $ab       ;RIPRTY
.. fb76  85 ab     sta $ab
.. fb78  20 db fc  jsr $fcdb     ;bump r/w pointer
.. fb7b  20 d1 fc  jsr $fcd1     ;check r/w pointer
.. fb7e  90 f2     bcc $fb72
.. fb80  a5 ab     lda $ab       ;RIPRTY
.. fb82  45 bd     eor $bd       ;ROPRTY - receive input
                                 character
.. fb84  f0 05     beq $fb8b     ;exit interrupt
.. fb86  a9 20     lda #$20      ;flag checksum error
.. fb88  20 1c fe  jsr $fe1c     ;set STATUS
.. fb8b  4c bc fe  jmp $febc     ;exit interrupt
```

64398 RESET TAPE POINTER

This routine sets the two byte buffer pointer to the start address of the load. Thus a program would be loaded directly into its RAM space, rather than via the cassette buffer.

```
.. fb8e  a5 c2     lda $c2       ;>STAL - I/O start address
.. fb90  85 ad     sta $ad       ;>SAL - tape buffer pointer
.. fb92  a5 c1     lda $c1
.. fb94  85 ac     sta $ac
.. fb96  60        rts
```

64407 NEW CHARACTER SETUP

This routine sets the serial bit counter to 8, and zeros four temporary pointers.

```
.. fb97  a9 08     lda #$08
```

```
., fb99  85 a3     sta $a3       ;bit counter for tape I/O
., fb9b  a9 00     lda #$00
., fb9d  85 a4     sta $a4       ;half dipole marker
., fb9f  85 a8     sta $a8       ;BITC1 - error flag
., fba1  85 9b     sta $9b       ;PRTY - tape character parity
., fba3  85 a9     sta $a9       ;RINONE - count zeros
., fba5  60        rts
```

64422 SEND TONE TO TAPE

This routine loads CIA timer B with either #0060 or #00B0, depending on the state of bit 0 of ROPRTY. The next routine uses a later entry point to load the timer with #0110. The data output line to the cassette unit is then toggled. The value in the timer causes a particular tone to be written onto the tape.

```
., fba6  a5 bd     lda $bd       ;ROPRTY - receive input
                                 character
., fba8  4a        lsr
., fba9  a9 60     lda #$60      ;flag #0060 for timer
., fbab  90 02     bcc $fbaf
., fbad  a9 b0     lda #$b0      ;flag #00B0 for timer
., fbaf  a2 00     ldx #$00
., fbb1  8d 06 dc sta $dc06      ;timer B low byte
., fbb4  8e 07 dc stx $dc07      ;timer B high byte
., fbb7  ad 0d dc lda $dc0d      ;clear CIA I.C.R.
., fbba  a9 19     lda #$19
., fbbc  8d 0f dc sta $dc0f      ;CIA control register B
., fbbf  a5 01     lda $01       ;R6510 - 6510 onboard I/O
                                 port
., fbc1  49 08     eor #$08      ;toggle data output line
., fbc3  85 01     sta $01
., fbc5  29 08     and #$08
., fbc7  60        rts
```

64456 WRITE DATA TO TAPE

This routine is the IRQ entry for the IRQ vector offset #0A. The data is written by using the previous routine to send a tone to the tape. Different signals are written to the tape for a 0 and 1. A parity bit and inter-byte marker are also written to the tape, and a two-second inter record gap is written after each 192 byte block of data. Data is also divided into blocks of eight bytes, with a ninth byte being computed and written as a checksum.

```
., fbc8  38        sec
., fbc9  66 b6     ror $b6       ;RODATA - combined error
```

```
                                  value
.,  fbcb  30 3c     bmi $fc09     ;exit interrupt
.,  fbcd  a5 a8     lda $a8       ;BITC1 - errors flag
.,  fbcf  d0 12     bne $fbe3
.,  fbd1  a9 10     lda #$10      ;flag #0110 for timer
.,  fbd3  a2 01     ldx #$01
.,  fbd5  20 b1 fb  jsr $fbb1     ;send long transition to tape
.,  fbd8  d0 2f     bne $fc09     ;exit interrupt
.,  fbda  e6 a8     inc $a8       ;BITC1
.,  fbdc  a5 b6     lda $b6       ;RODATA - combined error
                                  value
.,  fbde  10 29     bpl $fc09     ;exit interrupt
.,  fbe0  4c 57 fc  jmp $fc57     ;write tape block
.,  fbe3  a5 a9     lda $a9       ;RINONE - count zeros
.,  fbe5  d0 09     bne $fbf0
.,  fbe7  20 ad fb  jsr $fbad     ;send medium transition to
                                  tape
.,  fbea  d0 1d     bne $fc09     ;exit interrupt
.,  fbec  e6 a9     inc $a9       ;RINONE
.,  fbee  d0 19     bne $fc09     ;exit interrupt
.,  fbf0  20 a6 fb  jsr $fba6     ;send short transition to
                                  tape
.,  fbf3  d0 14     bne $fc09     ;exit interrupt
.,  fbf5  a5 a4     lda $a4       ;half cycle count
.,  fbf7  49 01     eor #$01
.,  fbf9  85 a4     sta $a4
.,  fbfb  f0 0f     beq $fc0c
.,  fbfd  a5 bd     lda $bd       ;ROPRTY - receive input
                                  character
.,  fbff  49 01     eor #$01
.,  fc01  85 bd     sta $bd       ;ROPRTY
.,  fc03  29 01     and #$01
.,  fc05  45 9b     eor $9b       ;PRTY - tape character parity
.,  fc07  85 9b     sta $9b
.,  fc09  4c bc fe  jmp $febc     ;exit interrupt
.,  fc0c  46 bd     lsr $bd       ;ROPRTY
.,  fc0e  c6 a3     dec $a3       ;serial bit count
.,  fc10  a5 a3     lda $a3
.,  fc12  f0 3a     beq $fc4e
.,  fc14  10 f3     bpl $fc09     ;exit interrupt
.,  fc16  20 97 fb  jsr $fb97     ;new character setup
.,  fc19  58        cli
.,  fc1a  a5 a5     lda $a5       ;CNTDN - cassette sync
                                  countdown
.,  fc1c  f0 12     beq $fc30
.,  fc1e  a2 00     ldx #$00
.,  fc20  86 d7     stx $d7       ;most recent dipole bit value
.,  fc22  c6 a5     dec $a5       ;CNTDN
.,  fc24  a6 be     ldx $be       ;FSBLK - cassette r/w block
```

```
                                        count
., fc26  e0 02     cpx #$02
., fc28  d0 02     bne $fc2c
., fc2a  09 80     ora #$80
., fc2c  85 bd     sta $bd         ;ROPRTY
., fc2e  d0 d9     bne $fc09       ;exit interrupt
., fc30  20 d1 fc  jsr $fcd1       ;check r/w pointer
., fc33  90 0a     bcc $fc3f
., fc35  d0 91     bne $fbc8       ;write data to tape
., fc37  e6 ad     inc $ad         ;>SAL - tape buffer pointer
., fc39  a5 d7     lda $d7         ;most recent dipole bit value
., fc3b  85 bd     sta $bd         ;ROPRTY
., fc3d  b0 ca     bcs $fc09       ;exit interrupt
., fc3f  a0 00     ldy #$00
., fc41  b1 ac     lda ($ac),y     ;get character from buffer
., fc43  85 bd     sta $bd         ;ROPRTY
., fc45  45 d7     eor $d7         ;most recent dipole bit value
., fc47  85 d7     sta $d7
., fc49  20 db fc  jsr $fcdb       ;bump r/w pointer
., fc4c  d0 bb     bne $fc09       ;exit interrupt
., fc4e  a5 9b     lda $9b         ;PRTY - tape character parity
., fc50  49 01     eor #$01
., fc52  85 bd     sta $bd         ;ROPRTY
., fc54  4c bc fe  jmp $febc       ;exit interrupt
```

64599 WRITE TAPE LEADER

This routine writes a block of data onto the tape. It first checks to see if there are blocks left to send; if not, then the tape motor is switched off. The CIA timer is loaded with #0078 and the corresponding tone sent to the tape. After a delay of #50 (set in INBIT and decremented each IRQ), the IRQ vector offset is set to #0A and the new vector set up. This will use the WRITE BITS routine to send data to tape in every interrupt.

```
., fc57  c6 be     dec $be         ;FSBLK - cassette r/w block
                                    count
., fc59  d0 03     bne $fc5e
., fc5b  20 ca fc  jsr $fcca       ;kill tape motor
., fc5e  a9 50     lda #$50
., fc60  85 a7     sta $a7         ;INBIT - temp delay counter
., fc62  a2 08     ldx #$08
., fc64  78        sei
., fc65  20 bd fc  jsr $fcbd       ;set IRQ vector #08
., fc68  d0 ea     bne $fc54       ;exit interrupt
., fc6a  a9 78     lda #$78        ;timer value = #0078
., fc6c  20 af fb  jsr $fbaf       ;send transition to tape
., fc6f  d0 e3     bne $fc54       ;exit interrupt
```

```
., fc71  c6 a7     dec $a7       ;INBIT - tape leader short
                                 count
., fc73  d0 df     bne $fc54     ;exit interrupt
., fc75  20 97 fb  jsr $fb97     ;new character setup
., fc78  c6 ab     dec $ab       ;RIPRTY - cassette short
                                 count
., fc7a  10 d8     bpl $fc54     ;exit interrupt
., fc7c  a2 0a     ldx #$0a
., fc7e  20 bd fc  jsr $fcbd     ;set IRQ vector #0A
., fc81  58        cli
., fc82  e6 ab     inc $ab       ;RIPRTY
., fc84  a5 be     lda $be       ;FSBLK
., fc86  f0 30     beq $fcb8     ;set IRQ vector
., fc88  20 8e fb  jsr $fb8e
., fc8b  a2 09     ldx #$09
., fc8d  86 a5     stx $a5       ;CNTDN - cassette sync
                                 countdown
., fc8f  86 b6     stx $b6       ;RODATA
., fc91  d0 83     bne $fc16     ;exit interrupt
```

64659 RESTORE NORMAL IRQ

This routine turns the screen back on, switches off the tape motor, restores normal operation of the CIA and replaces the hardware IRQ vector from its temporary store.

```
., fc93  08        php
., fc94  78        sei
., fc95  ad 11 d0  lda $d011     ;VIC control register
., fc98  09 10     ora #$10      ;switch screen on
., fc9a  8d 11 d0  sta $d011
., fc9d  20 ca fc  jsr $fcca     ;kill tape motor
., fca0  a9 7f     lda #$7f
., fca2  8d 0d dc  sta $dc0d     ;CIA I.C.R.
., fca5  20 dd fd  jsr $fddd     ;enable timer
., fca8  ad a0 02  lda $02a0     ;IRQTMP - holds normal vector
., fcab  f0 09     beq $fcb6     ;vector already restored
., fcad  8d 15 03  sta $0315     ;CINV - hardware IRQ vector
., fcb0  ad 9f 02  lda $029f
., fcb3  8d 14 03  sta $0314
., fcb6  28        plp
., fcb7  60        rts
```

64696 SET IRQ VECTOR

This routine first restores the normal IRQ vector, then sets a new vector from the table at $FD94 according to the offset in (X).

```
., fcb8  20 93 fc  jsr $fc93     ;restore normal IRQ
```

```
.,  fcbb  f0 97     beq $fc54       ;exit interrupt
.,  fcbd  bd 93 fd  lda $fd93,x     ;get new vector from table
.,  fcc0  8d 14 03  sta $0314       ;CINV - hardware IRQ vector
.,  fcc3  bd 94 fd  lda $fd94,x
.,  fcc6  8d 15 03  sta $0315
.,  fcc9  60        rts
```

64714 KILL TAPE MOTOR

This routine turns off the cassette motor via the 6510 I/O port.

```
.,  fcca  a5 01     lda $01         ;R6510 - onboard I/O port
.,  fccc  09 20     ora #$20        ;switch off tape motor
.,  fcce  85 01     sta $01
.,  fcd0  60        rts
```

64721 CHECK R/W POINTER

If the buffer pointer has reached the EOT address then Z is set to 1.

```
.,  fcd1  38        sec
.,  fcd2  a5 ac     lda $ac         ;SAL - tape buffer pointer
.,  fcd4  e5 ae     sbc $ae         ;EAL - tape end address
.,  fcd6  a5 ad     lda $ad
.,  fcd8  e5 af     sbc $af
.,  fcda  60        rts
```

64731 BUMP R/W POINTER

This routine is used to increment the tape buffer pointer.

```
.,  fcdb  e6 ac     inc $ac         ;<SAL - tape buffer pointer
.,  fcdd  d0 02     bne $fce1
.,  fcdf  e6 ad     inc $ad         ;>SAL
.,  fce1  60        rts
```

64738 POWER RESET ENTRY POINT

The system hardware reset vector ($FFFC) points here. This is the first routine executed by the machine when it is switched on. Firstly it sets the stack pointer to #FF, ie. the top of the stack. The interrupt flag is set and decimal mode cleared. Next a check is made for autostart code on any external ROM plugged into the cartridge port. If the autostart code is present, then an indirect jump is made to the cartridge cold start vector at $8000. The I/O chips are initialised and system constants (including the IRQ vector)

set up. Finally the interrupt mask is cleared and an indirect jump made to $A000 to cold start BASIC. Note that there is an NE555 timer attached to the power reset line to ensure that reset is pulled low for a minimum of 6 clock cycles after power is first switched on. This forces the processor to start executing the reset routine.

```
., fce2  a2 ff     ldx #$ff
., fce4  78        sei
., fce5  9a        txs           ;set stack pointer = #FF
., fce6  d8        cld
., fce7  20 02 fd  jsr $fd02     ;check 8-ROM
., fcea  d0 03     bne $fcef     ;no autostart code
., fcec  6c 00 80  jmp ($8000)   ;cold start 8-ROM
., fcef  8e 16 d0  stx $d016     ;VIC control register
., fcf2  20 a3 fd  jsr $fda3     ;initialise I/O
., fcf5  20 50 fd  jsr $fd50     ;initialise system constants
., fcf8  20 15 fd  jsr $fd15     ;KERNAL reset
., fcfb  20 5b ff  jsr $ff5b     ;setup for PAL/NSTC
., fcfe  58        cli
., fcff  6c 00 a0  jmp ($a000)   ;cold start BASIC
```

64770 CHECK FOR 8-ROM

This routine checks memory from $8000 to $8008 for external ROM autostart code. If found, then the routine exits with Z=1. The autostart code is as follows: a two byte cold restart vector, a two-byte warm restart vector, the message "CBM" in CBM ASCII, but with bit 7 of each character set, and the ASCII numerals "80".

```
., fd02  a2 05     ldx #$05
., fd04  bd 0f fd  lda $fd0f,x   ;8-ROM mask
., fd07  dd 03 80  cmp $8003,x   ;compare with actual 8-ROM
                                  code
., fd0a  d0 03     bne $fd0f
., fd0c  ca        dex
., fd0d  d0 f5     bne $fd04
., fd0f  60        rts
```

64786 8-ROM MASK

This is a table of 5 bytes holding the CBM ASCII message "CBM80". Note that CBM has bit 7 set and 80 has bit 7 clear. It is used in conjunction with the previous routine to check for autostart code at the beginning of a plug in ROM cartridge.

```
.:fd10 c3 c2 cd 38 30
```

64789 RESTOR: KERNAL RESET

The KERNAL routine RESTOR ($FF8A) jumps to this routine. It resets the KERNAL RAM vectors at $0304 onwards from the table at $FD30. The routine drops through to perform VECTOR.

```
., fd15  a2 30     ldx #$30
., fd17  a0 fd     ldy #$fd      ;$FD30 = table of KERNAL
                                 reset vectors
., fd19  18        clc
```

64794 VECTOR: KERNAL MOVE

The KERNAL routine VECTOR ($FF8D) jumps here to read or set the vectors depending on the state of carry. In this case, the vector table address must be in (X/Y).

```
., fd1a  86 c3     stx $c3       ;MEMUSS - temp for moving
                                 vectors
., fd1c  84 c4     sty $c4
., fd1e  a0 1f     ldy #$1f      ;pointer to end of table
., fd20  b9 14 03  lda $0314,y
., fd23  b0 02     bcs $fd27
., fd25  b1 c3     lda ($c3),y   ;get KERNAL reset vector
., fd27  91 c3     sta ($c3),y   ;replace it in 'ROM'
., fd29  99 14 03  sta $0314,y   ;store vector in RAM
., fd2c  88        dey
., fd2d  10 f1     bpl $fd20
., fd2f  60        rts
```

64816 KERNAL RESET VECTORS

This is a table of vectors for page 3. They are used to point to the KERNAL I/O routines and by altering them, provide a start point for user routines.

```
.:fd30 31 ea 66 fe 47 fe 4a f3
.:fd38 91 f2 0e f2 50 f2 33 f3
.:fd40 57 f1 ca f1 ed f6 3e f1
.:fd48 2f f3 66 fe a5 f4 ed f5
```

64848 RAMTAS: INITIALISE SYSTEM CONSTANTS

The KERNAL routine RAMTAS ($FF87) jumps to this routine. It starts by writing zeros into pages 0, 2 and 3 of memory. The pointer to the start of the cassette buffer is set up. A test is performed on RAM to determine where it ends and

ROM begins. This is done by writing a value, reading it and then comparing it with the value originally written. Note that the original RAM contents are preserved. Top of memory is then set by this test. Finally, bottom of memory and top of screen pointers are set.

```
., fd50  a9 00     lda #$00
., fd52  a8        tay
., fd53  99 02 00  sta $0002,y  ;clear page 0
., fd56  99 00 02  sta $0200,y  ;clear page 2
., fd59  99 00 03  sta $0300,y  ;clear page 3
., fd5c  c8        iny
., fd5d  d0 f4     bne $fd53
., fd5f  a2 3c     ldx #$3c
., fd61  a0 03     ldy #$03
., fd63  86 b2     stx $b2      ;TAPE1 - start of tape buffer
., fd65  84 b3     sty $b3
., fd67  a8        tay
., fd68  a9 03     lda #$03
., fd6a  85 c2     sta $c2      ;>STAL - I/O start address
., fd6c  e6 c2     inc $c2
., fd6e  b1 c1     lda ($c1),y
., fd70  aa        tax
., fd71  a9 55     lda #$55
., fd73  91 c1     sta ($c1),y
., fd75  d1 c1     cmp ($c1),y
., fd77  d0 0f     bne $fd88
., fd79  2a        rol
., fd7a  91 c1     sta ($c1),y
., fd7c  d1 c1     cmp ($c1),y
., fd7e  d0 08     bne $fd88
., fd80  8a        txa
., fd81  91 c1     sta ($c1),y
., fd83  c8        iny
., fd84  d0 e8     bne $fd6e
., fd86  f0 e4     beq $fd6c
., fd88  98        tya
., fd89  aa        tax
., fd8a  a4 c2     ldy $c2      ;>STAL
., fd8c  18        clc
., fd8d  20 2d fe  jsr $fe2d    ;set top of memory
., fd90  a9 08     lda #$08
., fd92  8d 82 02  sta $0282    ;>MEMSTR - pointer to bottom
                                 of memory
., fd95  a9 04     lda #$04
., fd97  8d 88 02  sta $0288    ;HIBASE - page holding start
                                 of screen
., fd9a  60        rts
```

64923 TABLE OF TAPE I/O IRQ VECTORS

This table holds the vectors for the four IRQ routines that are used in tape I/O. The vectors are:- $FC6A - tape write part 1, $FCBD - tape write part 2, $EA31 - normal keyscan IRQ, and $F92C - tape read.

```
.:fd9b 6a fc cd fb 31 ea 2c f9
```

64931 IOINIT: INITIALISE I/O

The KERNAL routine IOINT ($FF84) jumps to this routine. It sets the initial values of the Interrupt Control Register, Control Registers and Data Direction Registers for both CIAs. The SID filter/volume register is also set, and the 6510 onboard I/O port is initialised.

```
.. fda3  a9 7f     lda #$7f
.. fda5  8d 0d dc sta $dc0d     ;CIA #1 I.C.R.
.. fda8  8d 0d dd sta $dd0d     ;CIA #2 I.C.R.
.. fdab  8d 00 dc sta $dc00     ;CIA #1 data port A
.. fdae  a9 08     lda #$08
.. fdb0  8d 0e dc sta $dc0e     ;CIA #1 C.R. A
.. fdb3  8d 0e dd sta $dd0e     ;CIA #2 C.R. A
.. fdb6  8d 0f dc sta $dc0f     ;CIA #1 C.R. B
.. fdb9  8d 0f dd sta $dd0f     ;CIA #2 C.R. B
.. fdbc  a2 00     ldx #$00
.. fdbe  8e 03 dc stx $dc03     ;CIA #1 DDRB
.. fdc1  8e 03 dd stx $dd03     ;CIA #2 DDRB
.. fdc4  8e 18 d4 stx $d418     ;SID filter / volume register
.. fdc7  ca        dex
.. fdc8  8e 02 dc stx $dc02     ;CIA #1 DDRA
.. fdcb  a9 07     lda #$07
.. fdcd  8d 00 dd sta $dd00     ;CIA #2 data port A
.. fdd0  a9 3f     lda #$3f
.. fdd2  8d 02 dd sta $dd02     ;CIA #2 DDRA
.. fdd5  a9 e7     lda #$e7
.. fdd7  85 01     sta $01      ;R6510 - onboard I/O port
.. fdd9  a9 2f     lda #$2f
.. fddb  85 00     sta $00      ;D6510 - onboard DDR
```

64989 ENABLE TIMER

This routine loads and starts CIA#1 timer A with a value according to the PAL/NTSC flag. PAL sets the timer to #4025 and NTSC sets the timer to #4295. This is due to different system clock rates being used in PAL and NTSC systems.

```
.. fddd  ad a6 02 lda $02a6     ;PAL/NTSC flag
```

```
.. fde0  f0 0a     beq $fdec      ;NTSC
.. fde2  a9 25     lda #$25
.. fde4  8d 04 dc  sta $dc04      ;timer A low byte
.. fde7  a9 40     lda #$40
.. fde9  4c f3 fd  jmp $fdf3      ;set timer A high byte
.. fdec  a9 95     lda #$95
.. fdee  8d 04 dc  sta $dc04      ;timer A low byte
.. fdf1  a9 42     lda #$42
.. fdf3  8d 05 dc  sta $dc05      ;timer A high byte
.. fdf6  4c 6e ff  jmp $ff6e      ;load and start timer
```

65017 SETNAM: SAVE FILENAME DATA

The KERNAL routine SETNAM ($FFBD) jumps to this routine. On entry, (A) must hold the length of the filename and (X/Y) its start address.

```
.. fdf9  85 b7     sta $b7        ;FNLEN - length of current
                                   filename
.. fdfb  86 bb     stx $bb        ;FNADDR - pointer to current
                                   filename
.. fdfd  84 bc     sty $bc
.. fdff  60        rts
```

65024 SETLFS: SAVE FILE DETAILS

The KERNAL routine SETLFS ($FFBA) jumps to this routine. On entry, (A) must hold the logical file number, (X) the device number and (Y) the secondary address. These values are stored in their respective zero page locations.

```
.. fe00  85 b8     sta $b8        ;LA - current logical file
                                   number
.. fe02  86 ba     stx $ba        ;FA - current device number
.. fe04  84 b9     sty $b9        ;SA - current secondary
                                   address
.. fe06  60        rts
```

65031 READST: GET STATUS

The KERNAL routine READST ($FFB7) jumps to this routine. The current value in the I/O status word, ST is returned in (A) If the current device number is 2 (ie. RS-232) then the value of RSSTAT is returned.

```
.. fe07  a5 ba     lda $ba        ;FA - current device number
.. fe09  c9 02     cmp #$02       ;RS-232?
.. fe0b  d0 0d     bne $fe1a      ;no - read STATUS
.. fe0d  ad 97 02  lda $0297      ;RSSTAT - 6551 status
```

```
                                        register image
.. fe10  48        pha
.. fe11  a9 00     lda #$00
.. fe13  8d 97 02 sta $0297     ;RSSTAT
.. fe16  68        pla
.. fe17  60        rts
```

65048 SETMSG: FLAG STATUS

The KERNAL routine SETMSG ($FF90) jumps to this routine. The value in (A) is stored in MSGFLG, then the I/O status word, ST is placed in (A). If the routine is entered at $FE1C, then ST will be set to the value held in (A).

```
.. fe18  85 9d     sta $9d      ;MSGFLG - control KERNAL
                                messages
.. fe1a  a5 90     lda $90      ;STATUS - I/O status word
.. fe1c  05 90     ora $90
.. fe1e  85 90     sta $90
.. fe20  60        rts
```

65057 SETTMO: SET TIMEOUT

The KERNAL routine SETTMO ($FFA2) jumps to this routine. The value held in (A) is stored in the IEEE timeout flag.

```
.. fe21  8d 85 02 sta $0285     ;TIMOUT - IEEE timeout flag
.. fe24  60        rts
```

65061 MEMTOP: READ/SET TOP OF MEMORY

The KERNAL routine MEMTOP ($FFA9) jumps to this routine. If carry is set on entry, then the top of memory address is loaded into (X/Y). If carry is clear. then top of memory is set to the address held in (X/Y).

```
.. fe25  90 06bcc $fe2d         ;set top of memory
.. fe27  ae 83 02 ldx $0283     ;MEMSIZ - top of memory
                                pointer
.. fe2a  ac 84 02 ldy $0284
.. fe2d  8e 83 02 stx $0283
.. fe30  8c 84 02 sty $0284
.. fe33  60        rts
```

65076 MEMBOT: READ/SET BOTTOM OF MEMORY

The KERNAL routine MEMBOT ($FF9C) jumps to this routine. If carry is set on entry, then the bottom of memory address is loaded into (X/Y). If carry is clear, then bottom of memory

is set to the address held in (X/Y).

```
., fe34  90 06    bcc $fe3c     ;set bottom of memory
., fe36  ae 81 02 ldx $0281     ;MEMSTR - bottom of memory
                                pointer
., fe39  ac 82 02 ldy $0282
., fe3c  8e 81 02 stx $0281
., fe3f  8c 82 02 sty $0282
., fe42  60       rts
```

65091 NMI ENTRY POINT

This routine is executed every time a hardware Non-Maskable Interrupt occurs (eg. from RS-232). All 6510 internal registers are preserved on the stack, and if a ROM cartridge with autostart is present, it is warm started, otherwise the following warm start routine is called.

```
., fe43  78       sei
., fe44  6c 18 03 jmp ($0318)   ;vector NMINV - points to $FE47
., fe47  48       pha
., fe48  8a       txa
., fe49  48       pha
., fe4a  98       tya
., fe4b  48       pha
., fe4c  a9 7f    lda #$7f
., fe4e  8d 0d dd sta $dd0d     ;CIA I.C.R. (NMI)
., fe51  ac 0d dd ldy $dd0d
., fe54  30 1c    bmi $fe72     ;RS-232 NMI - service RS-232
., fe56  20 02 fd jsr $fd02     ;check 8-ROM
., fe59  d0 03    bne $fe5e
., fe5b  6c 02 80 jmp ($8002)   ;warm start 8-ROM
., fe5e  20 bc f6 jsr $f6bc     ;log CIA key reading
., fe61  20 e1 ff jsr $ffe1     ;STOP - scan stop key
., fe64  d0 0c    bne $fe72     ;warm start without resets
```

65126 WARM START BASIC

This routine is called from the NMI service routine and is vectored through $A002. If <stop> was pressed then KERNAL vectors are reset and I/O vectors initialised. If there are RS-232 bits to send then the next in line is sent. The NMI RS-232 out/in routines are executed dependant on the state of ENABL. Finally the 6510 registers are restored and the interrupt exited.

```
., fe66  20 15 fd jsr $fd15     ;KERNAL reset
., fe69  20 a3 fd jsr $fda3     ;initialise I/O
., fe6c  20 18 e5 jsr $e518     ;initialise I/O
., fe6f  6c 02 a0 jmp ($a002)   ;warm start BASIC
```

```
.. fe72  98        tya
.. fe73  2d a1 02  and $02a1    ;ENABL - RS-232 enables
.. fe76  aa        tax
.. fe77  29 01     and #$01
.. fe79  f0 28     beq $fea3
.. fe7b  ad 00 dd  lda $dd00    ;read RS-232 port
.. fe7e  29 fb     and #$fb
.. fe80  05 b5     ora $b5      ;NXTBIT - next RS-232 bit to
                                send
.. fe82  8d 00 dd  sta $dd00    ;write bit to RS-232 port
.. fe85  ad a1 02  lda $02a1    ;ENABL
.. fe88  8d 0d dd  sta $dd0d    ;CIA I.C.R. (NMI)
.. fe8b  8a        txa
.. fe8c  29 12     and #$12
.. fe8e  f0 0d     beq $fe9d    ;RS-232 send
.. fe90  29 02     and #$02
.. fe92  f0 06     beq $fe9a
.. fe94  20 d6 fe  jsr $fed6
.. fe97  4c 9d fe  jmp $fe9d    ;RS-232 send
.. fe9a  20 07 ff  jsr $ff07    ;NMI RS-232 out
.. fe9d  20 bb ee  jsr $eebb    ;RS-232 send
.. fea0  4c b6 fe  jmp $feb6    ;reset IRQ and exit
.. fea3  8a        txa
.. fea4  29 02     and #$02
.. fea6  f0 06     beq $feae
.. fea8  20 d6 fe  jsr $fed6    ;NMI RS-232 in
.. feab  4c b6 fe  jmp $feb6    ;reset IRQ and exit
.. feae  8a        txa
.. feaf  29 10     and #$10
.. feb1  f0 03     beq $feb6
.. feb3  20 07 ff  jsr $ff07    ;NMI RS-232 out
.. feb6  ad a1 02  lda $02a1    ;ENABL
.. feb9  8d 0d dd  sta $dd0d    ;CIA I.C.R (NMI)
.. febc  68        pla
.. febd  a8        tay
.. febe  68        pla
.. febf  aa        tax
.. fec0  68        pla
.. fec1  40        rti
```

65128 RS-232 TIMING TABLE - NTSC

This is the RS-232 baud rate timing prescaler table for use with NTSC machines. Each of the 10 entries in the table corresponds to one of the fixed RS-232 baud rates, starting with the lowest (50 baud), and finishing with the highest (2400 baud). Because of the difference in clock frequency between PAL and NTSC machines, there is a separate PAL timing table at $E4EC.

```
.:fec2 c1 27 3e 1a c5 11 74 0e
.:feca ed 0c 45 06 f0 02 46 01
.:fed2 b8 00 71 00
```

65238 NMI RS-232 IN

This routine inputs a bit from the RS-232 port and sets up the baud rate timing for the next bit. The RS-232 receive routine is then called.

```
., fed6  ad 01 dd lda $dd01     ;RS-232 I/O port
., fed9  29 01     and #$01
., fedb  85 a7     sta $a7      ;INBIT - input bit buffer
., fedd  ad 06 dd lda $dd06     ;timer B low
., fee0  e9 1c     sbc #$1c
., fee2  6d 99 02 adc $0299     ;<BAUDOF - baud rate
., fee5  8d 06 dd sta $dd06     ;timer B low
., fee8  ad 07 dd lda $dd07     ;timer B high
., feeb  6d 9a 02 adc $029a     ;>BAUDOF
., feee  8d 07 dd sta $dd07     ;timer B high
., fef1  a9 11     lda #$11
., fef3  8d 0f dd sta $dd0f     ;CIA C.R. B
., fef6  ad a1 02 lda $02a1     ;ENABL - RS-232 enables
., fef9  8d 0d dd sta $dd0d     ;CIA I.C.R. (NMI)
., fefc  a9 ff     lda #$ff
., fefe  8d 06 dd sta $dd06     ;timer B low
., ff01  8d 07 dd sta $dd07     ;timer B high
., ff04  4c 59 ef jmp $ef59     ;RS-232 receive
```

65287 NMI RS-232 OUT

This routine sets up the baud rate for sending the bits out, and adjusts the number of bits remaining to send.

```
., ff07  ad 95 02 lda $0295     ;M51AJB - non standard BPS time
., ff0a  8d 06 dd sta $dd06     ;timer B low
., ff0d  ad 96 02 lda $0296
., ff10  8d 07 dd sta $dd07     ;timer B high
., ff13  a9 11     lda #$11
., ff15  8d 0f dd sta $dd0f     ;CIA C.R. B
., ff18  a9 12     lda #$12
., ff1a  4d a1 02 eor $02a1     ;ENABL - RS-232 enables
., ff1d  8d a1 02 sta $02a1
., ff20  a9 ff     lda #$ff
., ff22  8d 06 dd sta $dd06     ;timer B low
., ff25  8d 07 dd sta $dd07     ;timer B high
., ff28  ae 98 02 ldx $0298     ;BITNUM - # bits still to
                                 send
., ff2b  86 a8     stx $a8      ;BITC1 - RS-232 in bit count
., ff2d  60        rts
., ff2e  aa        tax
```

```
.. ff2f  ad 96 02 lda $0296     ;M51AJB - non standard BPS time
.. ff32  2a       rol
.. ff33  a8       tay
.. ff34  8a       txa
.. ff35  69 c8    adc #$c8
.. ff37  8d 99 02 sta $0299     ;BAUDOF - baud rate
.. ff3a  98       tya
.. ff3b  69 00    adc #$00
.. ff3d  8d 9a 02 sta $029a
.. ff40  60       rts
```

65347 FAKE IRQ

```
.. ff41  ea       nop
.. ff42  ea       nop
.. ff43  08       php
.. ff44  68       pla
.. ff45  29 ef    and #$ef
.. ff47  48       pha
```

65352 IRQ ENTRY

This is the routine pointed to by the hardware IRQ vector at $FFFE. its main function is to distinguish between a hardware IRQ and a software BRK. Each type of interrupt is processed by its own routine.

```
.. ff48  48       pha           ;save processor registers
.. ff49  8a       txa
.. ff4a  48       pha
.. ff4b  98       tya
.. ff4c  48       pha
.. ff4d  ba       tsx
.. ff4e  bd 04 01 lda $0104,x
.. ff51  29 10    and #$10
.. ff53  f0 03    beq $ff58
.. ff55  6c 16 03 jmp ($0316)  ;vector CBINV - points to $FE66
.. ff58  6c 14 03 jmp ($0314)  ;vector CINV - points to $EA31
```

65371 CINT: INITIALISE SCREEN EDITOR

The KERNAL routine CINT ($FF81) jumps to this routine. It sets up the VIC II chip for normal operation. This section of code here is actually a patch to the original CINT routine, located at $E518. This patch serves to test whether the machine is built for NTSC or PAL use. The raster compare register is set to 311, and an interrupt awaited. because of the different number of scan lines on NTSC monitors, the interrupt will only occur if the machine operates the PAL system.

```
.. ff5b  20 18 e5 jsr $e518    ;initialise I/O
```

```
.. ff5e  ad 12 d0 lda $d012    ;VIC raster register
.. ff61  d0 fb    bne $ff5e    ;wait for top of screen
.. ff63  ad 19 d0 lda $d019    ;VIC interrupt flag register
.. ff66  29 01    and #$01
.. ff68  8d a6 02 sta $02a6    ;PAL/NTSC flag
.. ff6b  4c dd fd jmp $fddd    ;enable timer
.. ff6e  a9 81    lda #$81
.. ff70  8d 0d dc sta $dc0d    ;CIA I.C.R. (IRQ)
.. ff73  ad 0e dc lda $dc0e    ;CIA C.R. A
.. ff76  29 80    and #$80
.. ff78  09 11    ora #$11
.. ff7a  8d 0e dc sta $dc0e    ;CIA C.R. A
.. ff7d  4c 8e ee jmp $ee8e    ;serial clock off
```

65408 KERNAL VERSION ID

This is an I.D. byte, used to identify the version of the KERNAL ROM. The original 64 had a KERNAL I.D. of #AA.

```
.:ff80 00
```

65409 KERNAL JUMP TABLE

This is a table of jump vectors to I/O routines. No matter what Commodore machine, or where the routine is in ROM, the jump vector is always at the same location in this table.

```
.. ff81  4c 5b ff jmp $ff5b    ;CINT   - initialise screen
                               editor
.. ff84  4c a3 fd jmp $fda3    ;IOINT  - initialise
                               input/output
.. ff87  4c 50 fd jmp $fd50    ;RAMTAS - initialise RAM,
                               tape, screen
.. ff8a  4c 15 fd jmp $fd15    ;RESTOR - restore default I/O
                               vectors
.. ff8d  4c 1a fd jmp $fd1a    ;VECTOR - read/set vectored I/O
.. ff90  4c 18 fe jmp $fe18    ;SETMSG - control KERNAL
                               messages
.. ff93  4c b9 ed jmp $edb9    ;SECOND - send SA after
                               LISTEN
.. ff96  4c c7 ed jmp $edc7    ;TKSA   - send SA after TALK
.. ff99  4c 25 fe jmp $fe25    ;MEMTOP - read/set top of
                               memory
.. ff9c  4c 34 fe jmp $fe34    ;MEMBOT - read/set bottom of
                               memory
.. ff9f  4c 87 ea jmp $ea87    ;SCNKEY - scan keyboard
.. ffa2  4c 21 fe jmp $fe21    ;SETTMO - set IEEE timeout
.. ffa5  4c 13 ee jmp $ee13    ;ACPTR  - input byte from
                               serial bus
.. ffa8  4c dd ed jmp $eddd    ;CIOUT  - output byte to
                               serial bus
```

```
.., ffab  4c ef ed jmp $edef      ;UNTALK - command serial bus
                                  UNTALK
.., ffae  4c fe ed jmp $edfe      ;UNLSN  - command serial bus
                                  UNLISTEN
.., ffb1  4c 0c ed jmp $ed0c      ;LISTEN - command serial bus
                                  LISTEN
.., ffb4  4c 09 ed jmp $ed09      ;TALK   - command serial bus
                                  TALK
.., ffb7  4c 07 fe jmp $fe07      ;READST - read I/O status word
.., ffba  4c 00 fe jmp $fe00      ;SETLFS - set logical file
                                  parameters
.., ffbd  4c f9 fd jmp $fdf9      ;SETNAM - set filename
.., ffc0  6c 1a 03 jmp ($031a)    ;OPEN   - open logical file
.., ffc3  6c 1c 03 jmp ($031c)    ;CLOSE  - close logical file
.., ffc6  6c 1e 03 jmp ($031e)    ;CHKIN  - open channel for input
.., ffc9  6c 20 03 jmp ($0320)    ;CHKOUT - open channel for
                                  output
.., ffcc  6c 22 03 jmp ($0322)    ;CLRCHN - close all I/O
                                  channels
.., ffcf  6c 24 03 jmp ($0324)    ;CHRIN  - I/P character from
                                  channel
.., ffd2  6c 26 03 jmp ($0326)    ;CHROUT - O/P character to
                                  channel
.., ffd5  4c 9e f4 jmp $f49e      ;LOAD   - load RAM from device
.., ffd8  4c dd f5 jmp $f5dd      ;SAVE   - save RAM to device
.., ffdb  4c e4 f6 jmp $f6e4      ;SETTIM - set real-time clock
.., ffde  4c dd f6 jmp $f6dd      ;RDTIM  - read real-time clock
.., ffe1  6c 28 03 jmp ($0328)    ;STOP   - scan <stop> key
.., ffe4  6c 2a 03 jmp ($032a)    ;GETIN  - get from keyboard
                                  buffer
.., ffe7  6c 2c 03 jmp ($032c)    ;CLALL  - close all channels
                                  and files
.., ffea  4c 9b f6 jmp $f69b      ;UDTIM  - increment real-time
                                  clock
.., ffed  4c 05 e5 jmp $e505      ;SCREEN - return screen
                                  organisation
.., fff0  4c 0a e5 jmp $e50a      ;PLOT   - read/set cursor X/Y
                                  position
.., fff3  4c 00 e5 jmp $e500      ;IOBASE - return I/O base
                                  address
```

65530 SYSTEM HARDWARE VECTORS

This table contains the vectors for system reset, IRQ and NMI. They point to ROM routines which contain an indirect jump to RAM, so that user interrupt routines etc. can be written.

```
.:fff4 00 e5 52 52 42 59 43 fe
.:fffc e2 fc 48 ff
```

SECTION 4.
KERNAL GUIDE

A GUIDE TO THE KERNAL

When the operating system of a computer has been changed, most machine language programs written for that machine will no longer work since the machine language program will need to access specific locations in memory which may have been moved. This problem happens not only between models in a range but also in different 'issues' of the same model.

On Commodore computers, the most important Operating System routines have been written so that they always start at the same location no matter what version or model of the machine. This first instruction is always a JMP to wherever the actual code has been written. This group of JMP instructions is called the KERNAL JUMP TABLE. The routines within the table are as follows (in alphabetic order):

NAME	ADDRESS	DESCRIPTION
ACPTR	$FFA5	Input byte from serial port
CHKIN	$FFC6	Open channel for input
CHKOUT	$FFC9	Open channel for output
CHRIN	$FFCF	Input character from channel
CHROUT	$FFD2	Output character to channel
CIOUT	$FFA8	Output byte to serial bus
CINT	$FF81	Initialise screen editor
CLALL	$FFE7	Close all channels and files
CLOSE	$FFC3	Close specified file
CLRCHN	$FFCC	Close all channels
GETIN	$FFE4	Get character from file
IOBASE	$FFF3	Return base I/O address
IOINIT	$FF84	Initialise I/O
LISTEN	$FFB1	Send LISTEN to serial bus
LOAD	$FFD5	Load file into RAM
MEMBOT	$FF9C	Read/Set bottom of memory
MEMTOP	$FF99	Read/Set top of memory
OPEN	$FFC0	Open a logical file
PLOT	$FFF0	Read/Set cursor position
RAMTAS	$FF87	Initialise RAM, tape buffer & screen
RDTIM	$FFDE	Read TI jiffy clock
READST	$FFB7	Read I/O status word
RESTOR	$FF8A	Restore default I/O vectors
SAVE	$FFD8	Save memory to a device
SCNKEY	$FF9F	Scan keyboard
SCREEN	$FFED	Return screen dimensions
SECOND	$FF93	Send secondary address after LISTEN
SETLFS	$FFBA	Set up file parameters

SETMSG	$FF90	Control KERNAL messages
SETNAM	$FFBD	Set file name
SETTIM	$FFDB	Set TI jiffy clock
SETTMO	$FFA2	Set timeout on serial bus
STOP	$FFE1	Scan <STOP> key
TALK	$FFB4	Send TALK to serial bus
TKSA	$FF96	Send secondary address after TALK
UDTIM	$FFEA	Increment TI jiffy clock
UNLSN	$FFAE	Send UNLISTEN to serial bus
UNTLK	$FFAB	Send UNTALK to serial bus
VECTOR	$FF8D	Read/Set vectored I/O

A NOTE ON KERNAL ERROR HANDLING

During the use of the KERNAL I/O routines it is possible to create an error condition (for example, OPENing a write file to the keyboard). If this occurs in BASIC, this fact is recognised and the program is halted with a descriptive error message, but this does not happen when using the KERNAL routines. However, errors can be easily detected and dealt with.

For all KERNAL routines in which it is possible to generate an error, the carry flag in the Processor Status Word is used as a flag. If an error has occurred, the flag is set, and, if not, the flag is clear.

Many possible errors can be found by examining the I/O status word, ST (See the comments on READST for details of this). Other errors are flagged into (A). These errors follow the same sequence as the BASIC error codes and are summarised as follows:

I/O ERROR #0 ... <STOP> KEY WAS PRESSED
I/O ERROR #1 ... TOO MANY FILES
I/O ERROR #2 ... FILE OPEN
I/O ERROR #3 ... FILE NOT OPEN
I/O ERROR #4 ... FILE NOT FOUND
I/O ERROR #5 ... DEVICE NOT PRESENT
I/O ERROR #6 ... NOT INPUT FILE
I/O ERROR #7 ... NOT OUTPUT FILE
I/O ERROR #8 ... MISSING FILENAME
I/O ERROR #9 ... ILLEGAL DEVICE NUMBER

Additionally, it is possible to have the "I/O ERROR #..." message printed on the screen when the error occurs by calling the KERNAL SETMSG routine with (A) = #40 or #C0. Note that this routine also controls the printing of "SEARCHING", "FOUND" etc messages.

ACPTR ($FFA5)

Purpose:	Get a byte from the serial bus into (A).
Registers used:	(A)
Registers affected:	(A), (X)
Stack used:	13
Preparation:	TALK, (TKSA)
Errors:	#0, See READST

This routine is used to get a byte of data from a device on the serial bus. No information needs to be passed to this routine when it is called, and the byte of data is returned in (A). The TALK routine must be called first to command the device to send the data. The optional secondary address can also be sent by using the TKSA routine. Any errors encountered will be flagged into the I/O status word, ST.

Use: 1. Command device to TALK (using TALK, TKSA)
2. JSR ACPTR
3. Process data in (A)
4. Command device to UNTALK

CHKIN ($FFC6)

Purpose:	Open a channel for input
Registers used:	(X)
Registers affected:	(A), (X)
Stack used:	0
Preparation:	(SETLFS, SETNAM, OPEN)
Errors:	#0, #3, #5, #6.

Any logical file that is already OPEN can be defined as an input channel by this routine. The routine will automatically abort and return an error code if the device on the channel is not an input device. If the device is on the serial bus, then TALK and TKSA are sent automatically onto the bus.
If input is to be from the keyboard and there are no other channels open, then CHRIN or GETIN can be called without having to use either OPEN or CHKIN.

Use: 1. OPEN the logical file (with SETLFS, SETNAM and OPEN)
2. LDX #$ logical file number
3. JSR CHKIN
4. Input the data (using CHRIN or GETIN)
5. CLOSE the logical file

CHKOUT ($FFC9)

Purpose: Open a channel for output
Registers used: (X)
Registers affected: (A), (X)
Stack used: 4+
Preparation: (SETLFS, SETNAM, OPEN)
Errors: #0, #3, #5, #7

Any logical file that is already OPEN can be defined as an output channel by this routine. The routine will automatically abort and return an error code if the device on the channel is not an output device. If the device is on the serial bus, then LISTEN and SECOND are sent automatically onto the bus.
If the output is to be to the screen and there are no other channels open, then CHROUT can be called without having to use either OPEN or CHKOUT.

Use: 1. OPEN the logical file (with SETLFS, SETNAM and OPEN)
2. LDX #$ logical file number
3. JSR CHKOUT
4. Output data to the device (with CHROUT)
5. CLOSE the logical file

CHRIN ($FFCF)

Purpose: Input a byte from current input channel
Registers used: (A)
Registers affected: (A), (X)
Stack used: 7+
Preparation: (OPEN, CHKIN)
Errors: #0, see READST

This routine gets a byte from the input channel. No data needs to be passed to the routine when it is called. The input data is returned in (A). Note that the channel remains open after the call is made. A check must be made for the terminator to the data being input (carriage return for the keyboard, and the EOI flag set for other devices). The EOI signal is sent when the last byte in the file is sent, and is found via the READST routine.
Keyboard entry is handled somewhat differently. If there are no other channels open, then the OPEN and CHKIN calls are not needed. On the first call to CHRIN, the cursor is turned on, and all keys pressed are echoed on the screen until RETURN is pressed, when the logical screen line is transferred to the input bufer. The first character in the

buffer is then returned in (A). Subsequent calls to CHRIN will read characters from the input buffer, until it is empty.

Use: 1. OPEN file and set input device
     2. JSR CHRIN
     3. Process data byte
     4. JSR READST
     5. AND #$40 (Check for EOI)
     6. BEQ step 2 (Not EOI so get next byte)
     7. CLOSE file

CHROUT ($FFD2)

Purpose: Output a byte to the current channel
Registers used: (A)
Registers affected: (A)
Stack used: 8+
Preparation: (OPEN, CHKOUT)
Errors: #0, See READST

This routine sends a byte of data to the output channel. If this is the screen, and there are no other channels open, then the OPEN and CHKOUT routines are not needed. When this routine is called, the character in (A) is sent to the open channel. Note that the channel remains open after the call has been made.
Data is sent to the serial bus in the following way. When CHROUT is first called, the character is merely stored in the serial buffer. Subsequent calls to CHROUT cause the character in the buffer to be sent, and the new character stored in the buffer. The final character in the buffer is not sent until the channel is closed, when the EOI signal is sent with the last byte, and the device is UNLISTENED. This routine sends data to all open channels on the bus, so some care must be taken.

Use: 1. OPEN the logical file and channel
     2. LDA #$ data to be output
     3. JSR CHROUT
     4. Repeat from step 2 as necessary
     5. CLOSE the channel and logical file

CIOUT ($FFA8)

Purpose: Send a byte to the serial bus
Registers used: (A)

Registers affected:
Stack used: 5
Preparation: LISTEN, SECOND
Errors: #0, See READST

This routine is used to send a byte of data to a device on the serial bus. The byte of data to be sent must be placed in (A) before calling the routine. Any errors are flagged by the carry bit in the status register, and can be found by reading the I/O status word or (A)=0. The LISTEN routine must be called first to command the device to receive the data. The optional secondary address can also be sent by using the SECOND routine. Note that the byte to be sent is buffered. ie. the byte given is stored in the buffer, and the brevious byte from the buffer is sent to the bus. This is so that the last byte of the message can have the EOI handshake superimposed on it when the device is UNLISTENED.

Use: 1. Command the device to LISTEN (using LISTEN, SECOND)
2. LDA #$ data to be output
3. JSR CIOUT
4. Repeat from step 2 as needed
5. Command device to UNLISTEN

CINT ($FF81)

Purpose: Initialise screen editor
Registers used:
Registers affected: (A), (X), (Y)
Stack used: 4
Preparation:
Errors:

The VIC II chip is set up and the screen editor initialised by this routine. Any external ROM cartridge used should call this routine.

Use: 1. JSR CINT

CLALL ($FFE7)

Purpose: Close all files and channels
Registers used:
Registers affected: (A), (X)
Stack used: 11
Preparation:
Errors:

This routine effectively aborts all I/O operations and restores the system defaults. All of the entries in the file table are deleted and the CLRCHN routine is used to send UNTALK and UNLISTEN to all devices on the serial bus and restore keyboard and screen as the default input and output devices. The use for this routine is limited by the fact that the devices involved are not informed that the files have been closed.

Use: 1. JSR CLALL

CLOSE ($FFC3)

Purpose: Close a logical file
Registers used: (A)
Registers affected: (A), (X), (Y)
Stack used: 2+
Preparation:
Errors: #0, #3, See READST

This routine closes a specified logical file that was opened using the kernal OPEN routine. Unlike the CLALL routine, the file to the device is closed 'properly' as well as

UNLISTEN or UNTALK being sent. The RS-232 receive and transmit buffers are de-allocated if an RS-232 file is being closed.

Use: 1. LDA #$ logical file number
     2. JSR CLOSE

CLRCHN ($FFCC)

Purpose: Clear all I/O channels
Registers used:
Registers affected: (A), (X)
Stack used: 9
Preparation:
Errors:

This routine closes down all active I/O channels (NOT the logical files), and restores the default input device (keyboard) and output device (screen). If the channel(s) to be closed are on the serial bus, then UNTALK or UNLISTEN are sent to the bus. Note that all open channels on the bus will receive any data sent. Thus if, say the printer were commanded to LISTEN and the disk drive to TALK, a disk file

could be printed directly.
Note that this routine is called automatically by CLALL.

Use: 1. JSR CLRCHN

GETIN ($FFE4)

Purpose: Get a byte from the keyboard buffer
Registers used: (A)
Registers affected: (A), (X), (Y)
Stack used: 7+
Preparation: (CHKIN, OPEN)
Errors: See READST

This routine gets a byte from the keyboard buffer or RS-232 channel. No data needs to be passed to the routine when it is called. The input data is returned in (A). Note that the channel remains open after the call is made. A check must be made for the terminator to the data being input. Keyboard entry is handled in this manner: if there are no other channels open, then the OPEN and CHKIN calls are not needed. The routine will return the first character in the keyboard buffer in (A). This character is placed in the buffer by the IRQ routine (which calls SCNKEY) and is not echoed to the screen. Once the keyboard buffer is full (10 characters), then all further keypresses are ignored until a character has been removed from the buffer. For channels to devices other than RS-232 or the keyboard, use CHRIN or ACPTR routines.

Use: 1. OPEN file and set input device
2. JSR GETIN
3. Process data byte
4. JSR READST
5. BEQ step 2
6. CLOSE file

IOBASE ($FFF3)

Purpose: Get base address of I/O devices
Registers used: (X), (Y)
Registers affected: (X), (Y)
Stack used: 2
Preparation:
Errors:

The (X/Y) registers are set to the base address of the I/O

chips in the format (X) = low, (Y) = high. By addressing I/O registers as an offset from this address, compatibility will be maintained with any future version of the Commodore 64.

Use: 1. JSR IOBASE

IOINIT ($FF84)

Purpose: Initialise I/O devices
Registers used:
Registers affected: (A), (X), (Y)
Stack used:
Preparation:
Errors:

This routine initialises all I/O devices and routines. It should be called by an external ROM cartridge.

Use: 1. JSR IOINIT

LISTEN ($FFB1)

Purpose: Command a serial device to LISTEN
Registers used: (A)
Registers affected: (A)
Stack used:
Preparation:
Errors: See READST

This routine commands a specified device on the serial bus to LISTEN. The device number must be placed in (A) before entry. The device number is converted into a LISTEN address by ORing it with #20, and then sending it to the bus as a command 'under ATN'. The device will then receive data via the CIOUT routine.

Use: 1. LDA # device number
2. JSR LISTEN

LOAD ($FFD5)

Purpose: Load / Verify RAM from a device
Registers used: (A), (X), (Y)
Registers affected: (A), (X), (Y)
Stack used:

Preparation: SETLFS, SETNAM
Errors: #0, #4, #5, #8, #9, See READST

This routine is used to load or verify RAM from a device. (A) must hold #00 to load, and #01 to verify. It is not possible to load from the keyboard, RS-232 or screen. If the input device is given a secondary address of #00, then the header of the file is ignored, and the program is loaded at the relocated address given in (A/Y). If the secondary address is #01, then the program is loaded at the absolute address specified in the file header. On exit, the end address of the load is held in (X/Y).

Use: 1. Call SETLFS to set the file and device parameters
(Note: SA = 0 for relocated load, and 1 for absolute load)
2. Call SETNAM to specify the filename
3. LDA #$ (00 for LOAD, 01 for VERIFY)
4. LDX #$ Load address low (For relocated load only)
5. LDY #$ Load address high (For relocated load only)
6. JSR LOAD
7. STX VARTAB
8. STY VARTAB +1

MEMBOT ($FF9C)

Purpose: Read / set bottom of memory
Registers used: (X), (Y)
Registers affected: (X), (Y)
Stack used:
Preparation:
Errors:

This routine reads or sets the bottom of memory according to the state of the carry flag. If it is set, then the address of the lowest available byte of RAM is returned in (X/Y). If carry is clear, then the pointer to the beginning of RAM is set to the address held in (X/Y).

Use: 1. SEC (This reads MEMBOT)
2. JSR MEMBOT
3. Process (X) and (Y) registers

OR 4. LDX #$ address low byte
5. LDY #$ address high byte
6. CLC (This writes MEMBOT)
7. JSR MEMBOT

MEMTOP ($FFF9)

Purpose: Read / set top of memory
Registers used: (X), (Y)
Registers affected: (X), (Y)
Stack used: 2
Preparation:
Errors:

This routine reads or sets the top of memory according to the state of the carry flag. If it is set, then the address of the highest available byte of RAM is returned in (X/Y). If carry is clear, then the pointer to the top of RAM is set to the address held in (X/Y).

Use: 1. SEC (This reads MEMTOP)
     2. JSR MEMTOP
     3. Process (X) and (Y) registers

OR   4. LDX #$ address low byte
     5. LDY #$ address high byte
     6. CLC (This writes MEMTOP)
     7. JSR MEMTOP

OPEN ($FFC0)

Purpose: Open a logical file
Registers used:
Registers affected: (A), (X), (Y)
Stack used:
Preparation: SETLFS, SETNAM
Errors: #1, #2, #4, #5, #6, See READST

This routine is used to open a logical file, which can then be used by other KERNAL I/O routines. The arguments required to open the file (file number, device, secondary address and filename) must be set up using the SETLFS and SETNAM routines before OPEN is called. As a result, OPEN needs no parameters to be passed to it.

Special conditions apply to the RS-232 port. Firstly, two 256 byte FIFO (First In First Out) buffers are set up at the top of memory. The 512 bytes required are automatically allocated, and if there is insufficient space, the buffers will overwrite (and hence destroy) the end of any program or data present. No error message is printed, so care must be taken.

Secondly, there can be only one RS-232 file open at any time, since the buffer pointers would be reset by further OPEN commands.
Thirdly, a filename can be specified, containing up to 4 characters, representing the baud rate, parity, word length etc to be used. Se the section on the RS-232 port for more detail.

Use: 1 Call SETLFS to set file parameters
2. Call SETNAM to set filename
3. JSR OPEN
4 Perform other KERNAL I/O routines

PLOT ($FFF0)

Purpose: Read / set cursor position
Registers used: (X), (Y)
Registers affected: (A), (X), (Y)
Stack used: 2
Preparation:
Errors:

This routine reads or sets the current cursor position according to the state of the carry flag. If it is set,

then the X-Y coordinates of the cursor are returned in (X/Y). If carry is clear, then the X-Y coordinates of the cursor are set to the values held in (X/Y).

Use: 1. SEC (To read the cursor)
2. JSR PLOT
3. (X) and (Y) hold the cursor X-Y position

OR 4. LDX #$ cursor X position
5. LDY #$ cursor Y position
6. CLC (To set the cursor)
7. JSR PLOT

RAMTAS ($FF87)

Purpose: Perform RAM test and initialise RAM
Registers used: (A), (X), (Y)
Registers affected: (A), (X), (Y)
Stack used: 2
Preparation:

Errors:

This routine tests RAM, and sets its top and bottom pointers according to the result. Pages 0, 2 and 3 are cleared, the screen base is set to $0400, and the cassette buffer is allocated. The routine is normally called by an external ROM cartridge as part of the initialisation process.

Use: 1. JSR RAMTAS

RDTIM ($FFDE)

Purpose: Read system jiffy clock
Registers used: (A), (X), (Y)
Registers affected: (A), (X), (Y)
Stack used: 2
Preparation:
Errors:

This routine reads the system real-time software jiffy clock. The result is returned as three bytes, the most significant in (A), the next most significant in (X), and the least significant in (Y). The time is given in jiffies, each jiffy being 1/60 second.

Use: 1. JSR RDTIM

READST ($FFB7)

Purpose: Read I/O status word
Registers used: (A)
Registers affected: (A)
Stack used: 2
Preparation:
Errors:

This routine returns the current value of the I/O status word, ST in (A). This word gives information about the status of the last I/O action performed. The significance of each bit is shown below. Note that the result of RS-232 communications is obtained from RSSTAT rather than STATUS, although it is still read through this routine.

BIT	TAPE READ	TAPE LOAD/VERIFY	SERIAL READ/WRITE	RS-232
0			WRITE TIME OUT	PARITY ERROR
1			READ TIME OUT	FRAMING ERROR
2	SHORT BLOCK	SHORT BLOCK		IN BUFFER OVERRUN
3	LONG BLOCK	LONG BLOCK		IN BUFFER EMPTY
4	READ ERROR	ANY MISMATCH		NO CTS
5	BAD CHECKSUM	BAD CHECKSUM		
6	END-OF-FILE (EOF)		END-OR-IDENTIFY	NO DSR
7	END-OF-TAPE (EOT)	END-OF-TAPE (EOT)	DEVICE NOT PRESENT	BREAK DETECT

Use: 1. JSR READST
     2. decode information in (A)

RESTOR ($FF8A)

Purpose: Restore default system and interrupt vectors
Registers used:
Registers affected: (A), (X), (Y)
Stack used: 2
Preparation:
Errors:

This routine restores the default values of all BASIC and KERNAL vectors and interrupts.

Use: 1. JSR RESTOR

SAVE ($FFD8)

Purpose: Save memory to a device
Registers used: (A), (X), (Y)
Registers affected: (A), (X), (Y)
Stack used:
Preparation: SETLFS, SETNAM
Errors: #5, #8, #9, See READST

This routine saves memory to a specified device (not the keyboard, RS-232 or screen). (A) must point to a two byte area in zero page that holds the start address of the save. (X/Y) must point to the end address of the area of memory to

be saved. The file parameters must be set up before entry (using SETLFS and SETNAM), although the filename is optional when saving to cassette.

Use: 1. Call SETLFS (to set file parameters)
2. Call SETNAM (to set filename)
3. Set start address of save into zero page (TXTTAB ($2B-$2C) is often used)
4. LDA #$ zero page offset (to, say, <TXTTAB)
5. LDX #$ end address low byte
6. LDY #$ end address high byte
7. JSR SAVE

SCNKEY ($FF9F)

Purpose: Scan the keyboard
Registers used:
Registers affected: (A), (X), (Y)
Stack used: 5
Preparation: IOINIT
Errors:

This routine scans the keyboard and places any pressed keys (except for CTRL, SHIFT, CBM, STOP and RESTORE) in the keyboard buffer. This routine is called by the normal IRQ routine, and need not be called by the user, unless the normal IRQ service routine is bypassed.

Use: 1. JSR SCNKEY

SCREEN ($FFED)

Purpose: Return screen format
Registers used: (X), (Y)
Registers affected: (X), (Y)
Stack used: 2
Preparation: IOINIT
Errors:

This routine returns the number of rows and columns on the screen. Rows are held in (Y), and columns in (X).

Use: 1. JSR SCREEN

SECOND ($FF93)

Purpose: Send secondary address after LISTEN

Registers used: (A)
Registers affected: (A)
Stack used: 8
Preparation: LISTEN
Errors: See READST

This routine sends a secondary address to a device on the serial bus after it has been commanded to LISTEN. The secondary address must be placed in (A) before calling the routine.

Use: 1. Command device to LISTEN
2. LDA #$ Secondary address
3. JSR SECOND

SETLFS ($FFBA)

Purpose: Set up a logical file
Registers used: (A), (X), (Y)
Registers affected:
Stack used: 2
Preparation:
Errors:

This routine sets up the logical file number, device number and secondary address for the OPEN, LOAD and SAVE kernal routines. The logical file number is used for the table of active logical files and can be any value (#01 - #FF). Device numbers refer to CBM peripheral devices and can range from #00 to #1F. Device numbers greater than 3 are all on the serial bus. A table of commonly used devices is given below:

NUMBER	DEVICE
0	Keyboard
1	Cassette
2	RS-232 port
3	Screen
4	Printer
5	Printer (optional)
6	Plotter
8	Disk drive

A secondary address can be sent to the device during the

initial ATN handshake. For information on the use of secondary addresses, see the manuals to the devices concerned. If no secondary address is to be sent, then (Y) should be set to #FF.

Use: 1. LDA #$ Logical file number
     2. LDX #$ Device number
     3. LDY #$ Secondary address (or #FF)
     4. JSR SETLFS

SETMSG ($FF90)

Purpose: Control kernal messages
Registers used: (A)
Registers affected: (A)
Stack used: 2
Preparation:
Errors:

This routine controls the output of kernal control and error messages. Bit 6 controls the output of control messages (eg. 'SEARCHING FOR...'), and bit 7 controls the output of error messages (I/O ERROR #...'). When the bit is set, the messages are enabled, when it is clear, the messages are disabled. Note that messages of the type 'PRESS PLAY... cannot be disabled using this routine.

Use: 1. LDA #$ Control byte (#00, 40, #80, #C0)
     2. JSR SETMSG

SETNAM ($FFBD)

Purpose: Set up file name
Registers used: (A), (X), (Y)
Registers affected: (A), (X), (Y)
Stack used:
Preparation:
Errors:

This routine sets up a file name for use with the kernal OPEN, LOAD and SAVE routines. (A) is loaded with the length of the file name, and (X/Y) with its start address (format: low/high). This address can be anywhere in system memory. If no file name is required for cassette I/O then (A) should be set to zero. Note that in this case, the values in (X) and (Y) are unimportant.

Use: 1. LDA #$ File name
     2. LDX #$ Start address of name low
     3. LDY #$ Start address of name high
     4. JSR SETNAM

SETTIM ($FFDB)

Purpose: Set the system jiffy clock
Registers used: (A), (X), (Y)
Registers affected:
Stack used: 2
Preparation:
Errors:

This routine is used to set the real-time jiffy clock. The clock is three bytes long, and is set with (A) holding the MSB, then (X), and then (Y) holding the LSB. The clock is automatically incremented by an IRQ request and resets to zero after 5,184,000 jiffies (24 hours).

Use: 1. LDA #$ MSB of time (jiffies)
     2. LDX #$ Next most significant byte (jiffies)
     3. LDY #$ LSB of time (jiffies)
     4. JSR SETTIM

SETTMO ($FFA2)

Purpose: Set serial bus timeout
Registers used: (A)
Registers affected:
Stack used: 2
Preparation:
Errors:

This routine is used to enable and disable timeouts on the serial bus. If bit 7 of (A) is clear on entry, then the timeout will be enabled, and the Commodore will wait 64 milliseconds for a device to respond on the serial bus before flagging ?DEVICE NOT PRESENT or I/O ERROR #5. Setting bit 7 of (A) will disable the timeout. Note that timeouts are also used to communicate ?FILE NOT FOUND during a BASIC OPEN command.

Use: 1. LDA #$ Timeout flag (#00 to enable, #80 to disable)
     2. JSR SETTMO

STOP ($FFE1)

Purpose: Check if <STOP> was pressed
Registers used: (A)
Registers affected: (A), (X)
Stack used:
Preparation: (UDTIM)
Errors:

If <STOP> was pressed during the last call of UDTIM (normally during IRQ servicing), then the Z flag will be set on exit. Additionally, the input and output channels will be reset to their initial values. If <STOP> was not pressed, then Z remains clear and (A) will hold the value of the last row of the keyboard scan. This enables certain other keys to be checked for.

Use: 1. Call UDTIM if IRQ is disabled
2. JSR STOP
3. BEQ <STOP> pressed

TALK ($FFB4)

Purpose: Command serial device to TALK
Registers used: (A)
Registers affected: (A)
Stack used: 8
Preparation:
Errors: See READST

This routine commands the device specified in (A) (where the number can be from 4 to 31), to TALK. This is done by ORing the device number with #40 and sending this byte to the serial bus 'under ATN'.

Use: 1. LDA #$ Device number
2. JSR TALK

TKSA ($FF96)

Purpose: Send secondary address after TALK
Registers used: (A)
Registers affected: (A)
Stack used: 8
Preparation: TALK
Errors: See READST

This routine sends a secondary address to the serial bus after the TALK command. On entry, (A) must hold the secondary address to be sent to the device 'under ATN'. For an explanation of the effect of a particular secondary address, see the manual for the device concerned.

Use: 1. LDA #$ Secondary address
     2. JSR TKSA

UDTIM ($FFEA)

Purpose: Update system jiffy clock
Registers used:
Registers affected: (A), (X)
Stack used: 2
Preparation:
Errors:

This routine updates the system real time jiffy clock and scans the keyboard. It is normally called by the IRQ service routine 60 times each second. However, if the IRQ is disabled, then this routine must be called by the user and then STOP called if the clock and <STOP> key are to remain enabled.

Use: 1. JSR UDTIM

UNLSN ($FFAE)

Purpose: Command serial bus to UNLISTEN
Registers used:
Registers affected: (A)
Stack used: 8
Preparation:
Errors: See READST

This routine commands all devices on the serial bus to UNLISTEN, ie. to stop receiving data. Devices commanded to TALK are not affected by this routine. No parameters are needed.

Use: 1. JSR UNLSN

UNTLK ($FFAB)

Purpose: Command serial bus to UNTALK

Registers used:
Registers affected: (A)
Stack used: 8
Preparation:
Errors: See READST

This routine commands all devices on the serial bus to UNTALK, ie. to stop transmitting data. Devices commanded to LISTEN are not affected by this routine. No parameters need to be passed to the routine.

Use: 1. JSR UNTLK

VECTOR ($FF8D)

Purpose: Read / set RAM vectors
Registers used: (X), (Y)
Registers affected: (A), (X), (Y)
Stack used: 2
Preparation:
Errors:

This routine manages the system jump vectors stored in RAM from $0314 to $0333. An address must be specified in (X/Y). If the routine is entered with carry set, it will copy the RAM vectors to the new location pointed to by (X/Y). If carry is clear, then the block of memory pointed to by (X/Y) is copied to the RAM vector area. This can be used to, say copy the vectors into RAM, alter those desired, then copy the vectors back again.

Use: 1. Set (X/Y) to address for storing vectors
2. SEC (or CLC to copy vectors back)
3. JSR VECTOR

SECTION 5.
I/O GUIDE

THE CASSETTE PORT

The Commodore 64 has one dedicated external cassette deck for use in storing programs and data. The cassette port consists of six lines; MOTOR, SENSE, WRITE, READ. +5V and GND. The six lines can be divided into three functions: control, power and data.

The two power lines (+5V and GND) are used to drive the printed circuit board within the cassette unit and to provide a general power supply.

The MOTOR control line is used to directly drive the cassette motor. This line is connected to P5 of the 6510 onboard I/O port. The 5V signal from the port is boosted to an unregulated 9V, 500mA signal by a series of cascaded transistor amplifiers.

The SENSE control line is an input to P4 of the 6510 onboard I/O port, and is active low. It is used to detect whether the play, fast forward or rewind button has been pressed. The detection is performed using a switch inside the cassette deck. (Note that it is unable to differentiate between which of the buttons has been pressed.)

The two data lines are used to WRITE and READ data from the cassette deck. The WRITE line is connected to P3 of the 6510 onboard I/O port, and the READ line (normally pulled high) is connected to the FLAG input of 6526 CIA#1. The operation of the data lines is controlled entirely by software. Some amplification and pulse shaping circuits are required within the cassette deck itself to give the correct signal to the record head, and to amplify the signal from the playback head to give a 5V pulse train.

All Commodore I/O devices are assigned a 'device number'. The Cassette deck is assigned as device number 1. Logical files can be opened to this device and data read or written. Programs or blocks of memory can also be saved to it and loaded from it. Each open file must have a unique logical file number. This is to distinguish between other files which may be open. However, only one file may be opened to the cassette deck at any time. In addition to a device number, a logical file can specify a secondary or command address. This has the effect of sending a command to the device concerned. The available secondary addresses (and hence commands) for the cassette deck are detailed below for both data and program files.

SECONDARY ADDRESS	DATA FILE	PROGRAM FILE
0	Read from tape	Load/save at start of BASIC ($0801)
1	Write to tape	Load/save at absolute address
2	Write & force EOT	

(EOT = End Of Tape header)

Program (binary) files are created using the SAVE command. They are normally read using the LOAD command, but it is possible to read a binary file using an OPEN....GET# sequence. A binary file is normally used to store programs. This is because it stores the binary value of each consecutive memory location between specified start and end addresses.

The start address is normally $0801 (start of BASIC text) and the end address is the end of the program. However, any block of memory can be specified and saved using the KERNAL save routine ($FFD8). This enables the saving (and loading) of machine language programs, which are often address dependant.

ASCII files are created using the OPEN and PRINT# commands. They are read by using the OPEN and GET# or INPUT# commands. ASCII files are normally used to store strings of variables but programs can be stored in this way if required. Storing variables on tape as they were written would require the tape motor to be turned on and off repeatedly for a few bytes at a time, or leaving long gaps of unused (wasted) tape between each item. This problem is overcome on the Commodore 64 by use of a 192 byte buffer. Data is written to the buffer as required, and is only transmitted to the tape once the buffer is full or the file is closed. Similarly, 192 bytes of data are recieved from the tape into the buffer, and this data is removed from the buffer as it is required by the program. Once the buffer is empty then the process is repeated. There is a two second gap left between each 192 byte block on the tape to allow the motor time to achieve correct speed.

The Commodore 64 employs extensive error checking in the use of cassette tape. This provides for the greater reliability of Commodore tape systems over those employed by most other home computers. The error checking operates on two levels:

1) Data is split into chunks of 8 bytes. These are added together and the low byte of the result is written to tape as byte 9. This byte is known as a checksum. When the tape is read, the eight bytes are added together and the result is compared with the checksum digit. If they are different, then an error has occurred.

2) The second level of error checking involves recording each block of 192 bytes twice (or the whole program twice for binary files). Any errors that were detected and flagged by the checksum digit can be corrected with the pass 2 data. The two blocks are verified against each other, detecting those few errors not found by the checksum.

Each byte of data is recorded as a series of audio pulses. There are 8 data bits, one parity bit and a byte marker recorded per byte. The audio pulses are square waves with a mark:space ratio of 1:1. There are three different width pulses used. Long pulses have a frequency of 1.488 KHz, medium pulses are 1.953 KHz and short pulses are 2.840 KHz.

Data and parity bits are recorded as logic 0 or 1. '0' is defined as one short pulse followed by one medium pulse. '1' is defined as one medium pulse followed by one short pulse. Both of these values have the same overall "bit time" of 864 uS. A word marker indicates the start of a new byte and consists of one long pulse followed by one medium pulse. The gap between the pass 1 and pass 2 data is indicated by one long pulse and at least fifty short pulses.

At the start of recording, a 10 second leader is written consisting of short pulses. This enables the operating system to synchronise the read timing to the tape timing, enabling a wide variation of tape speeds to be read sucessfully. The two second inter-block gap is also written as a series of short pulses.

The synchronisation of tape and read timing allows for variations in tape motor speed of up to 20 percent. Unfortunately, the Direct Memory Access (DMA) operations required by the VIC II chip to generate the screen display can occasionally push the timing beyond its limits. To overcome this, the VIC II chip is disabled and the screen blanked during cassette operations.

Immediately after the leader, a 192 byte file header is written. This contains details of the file type, its filename and the start and end addresses (these point to the start and end of the cassette buffer in an ASCII file). An

additional header can be written at the end of the file indicating that End Of Tape has been reached.

When an error has occurred in a tape read/write operation, a flag is set in the I/O status word, ST. A different bit is set in ST according to which error has occurred. The significance of the bits within the I/O status word is detailed in the section on the KERNAL routine READST.

THE RS-232 PORT

The Commodore 64 contains one RS-232 style serial communications port for interfacing with printers, modems etc. The term RS-232 refers to an industry standard protocol for serial communications. Recognised variations to this standard are referred to by adding an extra letter to the name. eg. RS-232C. These variations usually refer to the voltage and polarity of the data signal.

Normally the RS-232C port transmits data using a V24 protocol ('0' = +12V, '1' = -12V). The Commodore 64 however only provides a 5 volt signal direct from one of the 6526 CIAs. An interface card may be needed to boost the signal to the required voltages and to provide the 25 way 'D' connector required by RS-232 devices.

There are two transmission modes available on the Commodore 64. The simplest of these is the '3 line' interface (Data in, Data out, GND). The other, more complex mode is the 'X line' interface which incorporates full handshaking. The pin out of the RS-232 port on the Commodore 64 is shown below:

PIN	6526	DESCRIPTION	EIA	ABV	I/O	MODES
C	PB0	RECEIVED DATA	(BB)	Sin	IN	1 2
D	PB1	REQUEST TO SEND	(CA)	RTS	OUT	* 2
E	PB2	DATA TERMINAL READY	(CD)	DTR	OUT	* 2
F	PB3	RING INDICATOR	(CE)	RI	IN	3
H	PB4	RECEIVED LINE SIGNAL	(CF)	DCD	IN	2
J	PB5	UNASSIGNED	( )	XXX	IN	3
K	PB6	CLEAR TO SEND	(CB)	CTS	IN	2
L	PB7	DATA SET READY	(CC)	DSR	IN	2
B	FLAG2	RECEIVED DATA	(BB)	Sin	IN	1 2
M	PA2	TRANSMITTED DATA	(BA)	Sout	OUT	1 2
A	GND	PROTECTIVE GROUND	(AA)	GND		1 2
N	GND	SIGNAL GROUND	(AB)	GND		1 2 3

MODES
1: 3-line interface
2: X-line interface
3: User available only
*: Held high during 3-line mode

The RS-232 port is implemented in a rather interesting way. Software emulation is used to simulate the operation of the 6551 Asynchronous Communications Interface Adaptor (ACIA).

Like other I/O chips, the 6551 functions through the use of memory mapped internal registers. These registers are duplicated in RAM for this emulation. These locations in RAM are called PSEUDO REGISTERS or REGISTER IMAGES. The three most important are the Control, Command and Status registers.
The Control register sets the speed of data transmission and reception (baud rate), the number of bits in each character of data (word length) and the number of stop bits attached to each word.

The Command register controls the mode of bus operation (3 line or X line), duplex mode and parity options.

The Status register returns any errors that may have occurred in data transmission or reception. With the hardware device in operation, these errors would have generated an Interrupt ReQuest (IRQ).

The layout and functions of these register images are shown in table 1

The RS-232 port has been assigned as I/O device 2. A logical file can be opened to it and data transmitted and recieved using the PRINT# and GET# commands; (INPUT# is not recommended for use with RS-232 since it causes the computer to hang while waiting for a carriage return character). When a file is opened to the port, input and an output buffers are set up. Each buffer is 256 bytes long, and they are located in the top 512 bytes of RAM. Care must be taken in opening an RS-232 file, since these buffers will overwrite any data or program which is stored in this area without returning an error condition.

Since there is provision for only one set of buffers, there can only be one file open to the RS-232 port. An attempt to open additional files to the port will cause the buffer pointers to be reset, and any data in the buffers will be lost.

A four character 'file name' should be specified in the OPEN command to set up the control, command and user baud rate register images. The user baud rate is not implemented by the software emulation of the system and need not be specified in the command. Typical syntax of the OPEN

command is:

OPEN 2,2,"<Control word><Command word><opt baud low><opt baud high>"

It must be noted that no error checking is carried out on the 'file name', and using a non-implemented baud rate will cause a system malfunction. High baud rates (above 300 baud) are not recommended for use from BASIC due to its inherent slowness.

The RS-232 buffers are organised on a First In First Out (FIFO) basis. Data is read from and written to the buffer by the user (rather than directly to the port) using PRINT# and GET# commands. The data is transmitted from the buffer to the port, and read from the port into the buffer under Non Maskable Interrupt (NMI) control. This allows accurate implementation of the baud rate via crystal controlled timers, and saves BASIC having to wait for data input before processing. Should the input bufer overflow (due say, to reading data at a high baud rate from BASIC), an error condition is set in the RS-232 status register, and all characters received whilst the error persists will be lost.

Note that neither the cassette unit nor serial bus should be used during RS-232 operations, as this would cause interrupt conflicts to occur (both cassette and serial bus operations involve IRQ interrupts).

THE SERIAL BUS

The Commodore 64 has a six line serial I/O bus for communicating with the Commodore range of serial peripherals - printers, plotters, disk drives etc. Although it is a serial port, it has an advantage over, say, RS-232 in that it can communicate with more than one device at a time.

In essence, the serial port is a cut down version of the IEEE-488 port found on the PET range of business computers. Both ports obey a strict protocol between devices which are listening and talking on the bus. The six lines employed by the serial bus are as follows:

1. Serial Service Request (SRQ) in
2. GND
3. Serial Attention (ATN) in/out
4. Serial CLK in/out
5. Serial DATA in/out
6. RESET

There are three classes of device which can be attached to the IEEE bus (serial or parallel). These are:

1. CONTROLLER (one only - CBM 64)
2. TALKER (Device transmitting to the bus - one at a time)
3. LISTENER (Device recieving from the bus - any number)

Note that the Commodore 64 is the only legal controller of the bus. It can also act as a listener or talker. All devices on the bus can be listening, but only one device can talk at any time. The controller is responsible for commanding each device to TALK or LISTEN. Similarly, it is responsible for commanding the device to stop talking (UNTALK) or to stop listening (UNLISTEN).

Commands are differentiated from normal data on the bus by use of the ATN (ATteNtion) line. When this line is pulled low, data is sent 'under attention' and is interpreted as a command. Such commands incorporate device numbers, secondary addresses and file names as well as instructions to TALK, LISTEN, UNTALK and UNLISTEN.

The CLK line is used for handshaking on the port and for indicating when data is valid.

The IEEE serial port does not have its own unique device number as do other peripherals. Instead, it is a means of accessing a range of individually numbered peripherals which are all attached to the one bus. The device numbers allocated to peripherals on the serial bus can range from 4 to 30. Any file open to a device within this range is automatically directed onto the serial bus.

The controller communicates with a device on the serial bus in the following way,

Firstly, the device number (or primary address) is sent to the bus 'under attention'. The controller will then wait for up to 65 milliseconds for the device to respond. If it does not respond within the required time, then a response timeout occurs and the controller returns with a ?DEVICE NOT PRESENT ERROR.

Next, a 'TALK - ATteNtion or 'LISTEN - ATteNtion' sequence is sent, commanding the device to TALK or LISTEN. The filename, if specified, is also sent in this sequence. If a 65 millisecond timeout occurs at this stage, then the controller responds with ?FILE NOT FOUND.

Data transfer can now proceed with ATN set at 1 (ie. not 'under ATteNtion). The final byte in the message being sent on the bus contains an End-Or-Identify (EOI) handshake. This informs the controller that it is time to UNTALK or UNLISTEN the device. (For more information see the Commodore 64 Programmer's Reference Guide)

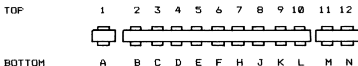
Secondary addresses have great importance on the serial bus, as they are used to command devices to enter various modes of operation. The effect of a secondary address is unique to the peripheral being addressed. Therefore the function of any particular secondary address on any particular device must be obtained from the relevant manuals of the device concerned, as it is not within the scope of this book.

Files are opened to serial devices using the OPEN command, and data is transferred via PRINT#, CMD....PRINT, GET# and INPUT# commands. It is also possible to SAVE blocks of memory to some of these devices. A total of up to ten files may be open to serial bus devices at any one time.

THE USER PORT

The user port is a general purpose 8-bit parallel I/O port. It is situated at the back of the Commodore 64, next to the cassette port. Physically, it consists of a flat 24 way, .15 pitch, male, edge connector.

In addition to the 8 data lines and two handshaking lines available to the user, this port contains control lines from other I/O ports on the Commodore 64. A pin description of the user port is shown below:



USER PORT LOOKING IN

PIN	DESCRIPTION	NOTES
1	GROUND	
2	+5V	100 mA MAX
3	RESET	Active low. Causes CBM 64 to COLD START. All pointers are reset but memory is not cleared.
4	CNT 1	Serial port counter from CIA#1
5	SP1	Serial port from CIA#1
6	CNT2	Serial port counter from CIA#2
7	SP2	Serial port from CIA#1
8	PC2	Handshaking line from CIA#2
9	SERIAL ATN	Serial bus ATN line
10	9V AC + PHASE	Connected directly to CBM 64 transformer. 50 mA MAX
11	9V AC − PHASE	
12	GROUND	
A	GROUND	
B	FLAG2	Negative going edge sensitive interrupt input. Sets FLAG interrupt bit in Interrupt Control Register.
C	PB0	User I/O port, RS-232 Sin
D	PB1	User I/O port, RS-232 DTR
E	PB2	User I/O port, RS-232 RI
F	PB3	User I/O port, RS-232 DCD
H	PB4	User I/O port
J	PB5	User I/O port, RS-232 CTS
K	PB6	User I/O port, RS-232 DSR
L	PB7	User I/O port, RS-232 Sout
M	PA2	User I/O port
N	GROUND	

The user port is derived from one of the two 6526 Complex Interface Adaptors (CIA) used by the Commodore 64. These 6526s control the keyboard, joysticks, light pen, serial bus, RS-232 port and serial port. The RS-232 port and the user port share the same data lines, and so cannot be used simultaneously. Exact usage of the two CIAs is shown in the I/O memory map in the next section.

Two handshaking lines are provided on the 6526: FLAG and PC.

PC is an output and will go low for one clock cycle after data is written to port B. It can thus be used to signal that data is available.

FLAG is an input and is sensitive to negative going signals. It sets the FLAG bit in the interrupt register, and hence can be used to signal that data is available from an external device.

The 6526 has 16 internal registers, listed below:

REG.	NAME	FUNCTION
0	PRA	Peripheral register A
1	PRB	Peripheral register B
2	DDRA	Data Direction Register A
3	DDRB	Data Direction Register B
4	TA LO	Timer A low register
5	TA HI	Timer A high register
6	TB LO	Timer B low register
7	TB HI	Timer B high register
8	TOD 10THS	10ths of seconds register
9	TOD SEC	Seconds register
10	TOD MIN	Minutes register
11	TOD HR	Hours and AM/PM register
12	SDR	Serial Data Register
13	ICR	Interrupt Control Register
14	CRA	Control Register A
15	CRB	Control Register B

CIA REGISTER FUNCTION DESCRIPTIONS

0: PERIPHERAL REGISTER A

This register reads or writes data to the outside world data port A. Its contents reflect the value held on pins PA0-PA7 of the chip. The direction of flow of data through each bit of the port is controlled by register 2.

1: PERIPHERAL REGISTER B

This register reads or writes data to the outside world data port B. Its contents reflect the value held on pins PB0-PB7

of the chip. The direction of flow of data through each bit of the port is controlled by register 3.

2: DATA DIRECTION REGISTER A

This register determines which bits of PRA will be inputs or outputs. If a bit in this register is set to 0, then the corresponding bit in PRA is an input. Similarly, if a bit in this register is set to 1, its corresponding bit is an output. This register is set to all zeros (ie inputs) on power-up.

3: DATA DIRECTION REGISTER B

This register determines which bits of PRB will be inputs or outputs. If a bit in this register is set to 0, then the corresponding bit in PRB is an input. Similarly, if a bit in this register is set to 1, its corresponding bit is an output. This register is set to all zeros (ie inputs) on power-up.

4,5: TIMER A

These two registers taken together form a 16 bit value. Once a value is written into these registers, it is latched and remains until a new value is written. This means that the timer can be repeatedly used without having to rewrite the start value. The timer, once started counts down from its set value to zero. Its action is controlled via register 14. Reading the timer returns its current value regardless of the start value latched into it by the write operation.

6,7: TIMER B

These two registers taken together form a 16 bit value. Once a value is written into these registers, it is latched and remains until a new value is written. This means that the timer can be repeatedly used without having to rewrite the start value. The timer, once started counts down from its set value to zero. Its action is controlled via register 15. Reading the timer returns its current value regardless of the start value latched into it by the write operation.

8: 10THS OF SECOND

Writing to this register sets the value for 10ths of a second in the Time Of Day (TOD) clock, or the alarm, depending on the state of the ALARM bit in register 15. When this

register is read, 10ths of seconds value of the TOD clock is returned. This register only uses the low four bits, and is in BCD (Binary Coded Decimal) format.

9: SECONDS

Writing to this register sets the value for seconds in the Time Of Day (TOD) clock, or the alarm, depending on the state of the ALARM bit in register 15. When this register is read, the seconds value of the TOD clock is returned. This register is in BCD format.

10: MINUTES

Writing to this register sets the value for minutes in the Time Of Day (TOD) clock, or the alarm, depending on the state of the ALARM bit in register 15. When this register is read, the minutes value of the TOD clock is returned. This register is in BCD format.

11: HOURS and AM/PM

Writing to this register sets the value for hours in the Time Of Day (TOD) clock, or the alarm, depending on the state of the ALARM bit in register 15. When this register is read, the hours value of the TOD clock is returned. Bits 0-5 represent the hours value and bit 7 the AM/PM value (0=AM, 1=PM). All values are in BCD format.

12: SERIAL DATA REGISTER

Data written to this register is shifted out one bit at a time (MSB first), through a shift register onto the SP pin of the chip. The port operates in a synchronous manner, the timing pulses appearing on the CNT pin of the 6526. The baud rate for the serial port is generated by TIMER A, with a maximum possible rate of o2 divided by 4. After each byte has been sent, an interrupt bit is set in register 13. Should the processor write new data to the SDR before this interrupt occurs, then data will be sent continuously. Data can also be shifted into the SDR from the outside world. The process in this case is essentially the same as for output. The direction of data transfer is controlled by bit 6 of register 14.

13: INTERRUPT CONTROL REGISTER

This register can be used to cause the 6510 processor in the Commodore 64 to stop its current task and execute a program called the Interrupt Service Routine (see $EA31 in section 3). This register allows an Interrupt ReQuest (IRQ) to be generated when any one of five bits is set. The IRQ will only be sent to the 6510 however if a 1 is written into that bit. The function of each bit is shown below:

BIT 7 SET/CLR When set to 1, all other bits with 1 in them will be set When set to 0, all other bits with 1 in will be cleared BIT 4 FLAG When set, any -ve transition on FLAG pin causes IRQ BIT 3 SP When set, IRQ occurs after 8 bits of serial data have been shifted in or out
BIT 2 ALRM When set, IRQ occurs when TOD clock = value in ALARM
BIT 1 TB When set, IRQ occurs on TIMER B timeout
BIT 0 TA When set, IRQ occurs on TIMER A timeout
NOTE: CIA#1 is connected to the IRQ line, and CIA#2 to the NMI line.

14: CONTROL REGISTER A

This register is used to control TIMER A. Each bit has a separate function and is detailed below:

BIT 7 TODIN 1 = 50Hz signal required on the TOD pin for accurate timing. 0 = 60 Hz required.
BIT 6 SPMODE 1 = Data input to serial port. 0 = Data output from serial port. Timing is at CNT pin.
BIT 5 INMODE 1 = Timer driven by pulses on CNT pin. 0 = Timer driven by o2 system clock.
BIT 4 LOAD 1 = Latched value is forced into timer regardless of current state. Bit returns to 0 value.
BIT 3 RUNMODE 1 = Count from latched value to zero and stop. 0 = Count from latched value to zero and repeat.
BIT 2 OUTMODE If PBON is set, then 1 = Toggle PB6 on each timeout. 0 = Pulse PB6 high for 1 cycle each timeout.
BIT 1 PBON 1 = Direct timeout IRQ onto PB6 of user port. 0 = PB6 normal.
BIT 0 START 1 = Start timer. 0 = Stop timer.

15: CONTROL REGISTER B

This register is used to control timer B. Bits 0 - 4 are as CRA, except that bit 1 directs output to PB7 and not PB6. The function of bits 5 - 7 are shown below:

BIT 5,6 INMODE Set one of 4 possible sources to drive timer:

CRB6	CRB5	
0	0	Run timer on system o2 clock
0	1	Run timer on +ve CNT transitions
1	0	Count TIMER A timeouts
1	1	Count TIMER A timeouts while CNT = 1

BIT 7 ALARM 1 = Write to TOD sets alarm. 0 = Write to TOD sets clock.

6510 ONBOARD I/O PORT

$0000: 6510 DATA DIRECTION REGISTER — 0

This register determines which of the lines on the 6510 I/O port are inputs, and which are outputs. By setting a bit in this register to 1, the corresponding bit of the I/O port is defined as an output. By clearing a bit in this register to 0, the corresponding bit of the I/O port is defined as an input. The register is set to the value %00101111 during the system initialisation process.

$0001: 6510 I/O PORT — 1

This register is a 6-bit bi-directional I/O port. The direction of flow of data for each bit is determined by the corresponding bit of the data direction register ($0000). Although this is an 8-bit register, the two most significant bits do not appear on the 6510 pin out. The I/O port is allocated as follows:

BIT	FUNCTION
0	LORAM signal (0= switch out BASIC ROM)
1	HIRAM signal (0= switch out KERNAL ROM)
2	CHAREN signal (0=switch in charaacter ROM)
3	Cassette data output line
4	Cassette switch sense (1= switch closed)
5	Cassette motor control (0= on)

6566/7 VIC II CHIP

$D000: SPRITE 0, X POSITION — 53248

This register controls the X or horizontal position of sprite 0. There are 512 possible horizontal positions for the sprite, with the most significant bit of the register being held at $D010. Note that only positions 23 to 347 are visible on the screen, all other positions being behind the border.

$D001: SPRITE 0, Y POSITION — 53249

This register controls the Y or vertical position of sprite 0. There are 256 possible vertical positions for the sprite. Note that only positions 50 to 249 are visible on the screen, all other positions being behind the border.

$D002: SPRITE 1, X POSITION 53250

This register is the same as $D000, but for sprite 1.

$D003: SPRITE 1, Y POSITION 53251

This register is the same as $D000, but for sprite 1.

$D004: SPRITE 2, X POSITION 53252

This register is the same as $D000, but for sprite 2.

$D005: SPRITE 2, Y POSITION 53253

This register is the same as $D000, but for sprite 2.

$D006: SPRITE 3, X POSITION 53254

This register is the same as $D000, but for sprite 3.

$D007: SPRITE 3, Y POSITION 53255

This register is the same as $D000, but for sprite 3.

$D008: SPRITE 4, X POSITION 53256

This register is the same as $D000, but for sprite 4.

$D009: SPRITE 4, Y POSITION 53257

This register is the same as $D000, but for sprite 4.

$D00A: SPRITE 5, X POSITION 53258

This register is the same as $D000, but for sprite 5.

$D00B: SPRITE 5, Y POSITION 53259

This register is the same as $D000, but for sprite 5.

$D00C: SPRITE 6, X POSITION 53260

This register is the same as $D000, but for sprite 6.

$D00D: SPRITE 6, Y POSITION 63261

This register is the same as $D000, but for sprite 6.

$D00E: SPRITE 7, X POSITION 53262

This register is the same as $D000, but for sprite 7.

$D00F: SPRITE 7, Y POSITION 53263

This register is the same as $D000, but for sprite 7.

$D010: SPRITES 0-7 MSB OF X POSITION 53264

This register contains the most significant bits of the 9-bit registers controlling the horizontal position of sprites 0-7. Bit 0 of this register is for sprite 0, bit 1 for sprite 1 etc. When a bit is zero, the corresponding sprite will be displayed in horizontal positions 0 to 255 (ie. on the left of the screen). When a bit is set to 1, the corresponding sprite will be displayed in horizontal positions 256 to 511 (ie. on the right of the screen).

$D011: VIC CONTROL REGISTER 1 53265

BIT 7: RASTER COMPARE REGISTER MSB

This is the most significant bit of the raster compare register ($D012). A full explanation of its function is given in the description for that register.

BIT 6: EXTENTED COLOUR TEXT MODE

This mode is enabled by setting bit 6 to 1. Extended colour mode enables an individual background colour to be selected for each individual character cell. The colour of the actual character is determined by colour RAM in the normal way, but the background colour is determined by a combination of the two most significant bits of the character screen code and the background colour registers as shown below:

MS BIT PAIR OF CHARACTER	BACKGROUND COLOUR DISPLAYED
00	Background #0 ($D021)
01	Background #1 ($D022)
10	Background #2 ($D023)
11	Background #3 ($D024)

Because of the limitations imposed by this mode, only 4 of the possible 16 background colours may be on the screen at

any one time. Also, because the two most significant bits of the character value are used to
determine the background colour, only the first 64 characters of the character set may be displayed. It is not recommended that this mode be used in conjunction with multicolour mode.

BIT 5: BIT MAP MODE

Bit map mode is enabled by setting bit 5 of $D011 to 1. This mode provides a screen resolution of 320 by 200 individually addressable pixels. Each pixel to be displayed is stored as part of a byte in an 8000 byte 'display base'. Each 8 consecutive bytes in the display base form a block equivalent to a character cell in normal text mode. The text video matrix is used in this mode to determine the screen colour. The high 4 bits determine the colour of any 'set' pixel, and the low 4 bits determine the colour of any 'clear' pixels within that block. Colour RAM is not used in this mode.

BIT 4: DISPLAY ENABLE

This bit is used to enable or disable the video screen display. During normal display operations, the screen is enabled, with this bit set to 1. When this bit is clear, the screen is blanked to the background colour held in $D020, and the VIC display operations are suspended. This means that the processor can run slightly faster, as the chip is no longer stealing o2 clock cycles for display purposes. Some o2 cycles may however be required if there are sprites enabled, even though they are not displayed. The screen is automatically blanked during cassette operations.

BIT 3: SELECT 24/25 ROW TEXT

By clearing this bit to 0, the height of the text screen window can be reduced from 25 to 24 rows. This facility is normally used in conjunction with smooth scrolling to allow data to be hidden behind the screen border and smoothly scrolled into view. The expansion of the border over the screen does not affect the data in the display matrix.

BITS 2-0: SMOOTH SCROLL VERTICAL DOT POSITION

These bits are used to scroll the screen smoothly up or down. The bits can hold a value between 0 and 7 to indicate the pixel position in a single character block to be scrolled. By reducing the height of the screen, new

information can be written to the hidden row and subsequently scrolled onto the screen.

$D012: RASTER READ/WRITE REGISTER 53266

This is a dual function 9-bit register (the MSB is bit 7 of $D011). Reading this register returns the current raster scan position on the screen (the raster is the dot of light that scans across the tv tube). The screen display window is from raster 51 to raster 251.
Writing to this register causes a latch to be set for an internal raster compare. When the raster scan reaches the latched value, the raster interrupt bit is set in $D019. This facility can be used, for example, to display more than 8 sprites on the screen, to produce a text screen window on a hires screen etc.. It is also used by the operating system to determine whether a PAL or NTSC system is being used.

$D013: LIGHT PEN LATCH, X POSITION 53267

This register can be used to read the horizontal position of a light pen on the screen. Horizontal resolution is 2 pixels. The light pen latch can only be triggered once per frame, so several readings may have to be taken and averged to form the final value.

$D014: LIGHT PEN LATCH, Y POSITION 53268

This register can be used to read the vertical position of a light pen on the screen. Vertical resolution is 1 pixel. The light pen latch can only be triggered once per frame, so several readings may have to be taken and averged to form the final value.

$D015: SPRITE DISPLAY ENABLE 53269

This register is used to enable each of the eight sprites. By setting a bit in this register, the corresponding sprite is enabled and can be displayed on the screen. Note that the displaying of sprites may cause the processor to slow down slightly, as the VIC II chip needs to make additional memory accesses.

$D016: VIC CONTROL REGISTER 2 53270

BITS 7-6: UNUSED

BIT 5: RESET BIT
This bit resets the VIC II chip and should always be set to 0.

BIT 4: ENABLE MULTICOLOUR MODE

Setting this bit will enable either multicolour character mode or multicolour bit map mode, dependant on the mode at entry.

In multicolour character mode, up to 4 colours can be displayed in each character cell, but with reuced horizontal resolution. The character cell is divided into 4 horizontal blocks, each 2 pixels wide. The patterns of the bit pairs within that character cell determine the colour of the blocks in the following manner:

BIT PAIR	COLOUR
00	Background #0 ($D021)
01	Background #1 ($D022)
10	Background #2 ($D023)
11	Determined by colour RAM

Bit 3 of any location in colour RAM is used to select multicolour mode for each character cell. Bit 3=1 sets multicolour mode, and bit 3=0 sets the normal high resolution character mode.

Multicolour bit map mode is similar to multicolour character mode in that 4 colours can be displayed in each character cell. The bit pattern, however, is held in the 8000 byte hires display base.

BIT 3: SELECT 38/40 COLUMN TEXT

By clearing this bit to 0, the width of the text screen window can be reduced from 40 to 38 columns. This facility is normally used in conjunction with smooth scrolling to allow data to be hidden behind the screen border and smoothly scrolled into view. The expansion of the border over the screen does not affect the data in the display matrix.

BITS 2-0: SMOOTH SCROLL HORIZONTAL DOT POSITION

These bits are used to scroll the screen smoothly right or left. The bits can hold a value between 0 and 7 to indicate the pixel position in a single character block to be scrolled. By reducing the width of the screen, new

information can be written to the hidden columns and subsequently scrolled onto the screen.

$D017: SPRITES 0-7 EXPAND, VERTICAL DIRECTION 53271

Setting a bit in this register causes the corresponding sprite to be expanded in the vertical (Y) direction. This expansion doubles the height of the sprite, but affords no increase in resolution.

$D018: VIC MEMORY CONTROL REGISTER 53272

BITS 4-7: VIDEO MATRIX BASE ADDRESS

These 4 bits determine the start location of the normal video matrix within the 16K memory bank currently selected for the VIC chip. There are 16 possible start locations for the screen, each occurring on a 1K boundary in memory ($0000, $0400, $0800 etc.). The default value of these bits is %0001, which places the screen at $0400 (in bank 0). Whenever this value is changed, the KERNAL must be informed by adjusting location $0228.

BITS 3-1: CHARACTER MATRIX DOT DATA BASE ADDRESS

These three bits determine the start location within the 16K VIC memory window (or bank) of the data used to make up the screen characters used for display. There are 8 possible start locations for the character data, each occurring on a 2K boundary in memory ($0000, $0800, $1000 etc.). The default value is %010, pointing to $1000 in bank 0. This location provides an image of the character ROM which is at $D000-$DFFF. There are also ROM images at value %010 in bank 2, and at value %011 in banks 0 and 2.

BIT 0: UNUSED

$D019: VIC INTERRUPT FLAG REGISTER 53273

BIT 7: IRQ LATCHED

This bit is set to 1 if any of the four interrupt sources have been triggered, and if the corresponding enable bit in register $D01A has been set. This represents an IRQ signal being sent to the processor.

BITS 6-4: UNUSED

BIT 3: LIGHT PEN

This bit is set to 1 on a negative transition of the light pen input. This can only happen once per video frame. In order for this to cause an IRQ, bit 3 of $D01A must be set by the programmer.

BIT 2: SPRITE TO SPRITE COLLISION

This bit is set to 1 on a collision between any two sprites. The bit is set only by the first such collision, and is unaffected by any further collisions. In order to determine which sprites have collided, the sprite collision register ($D01E) must be examined. In order for this to cause an IRQ, bit 2 of $D01A must be set by the programmer.

BIT 1: SPRITE TO BACKGROUND COLLISION

This bit is set to 1 on a collision between any sprite and screen data. The bit is set only by the first such collision, and is unaffected by any further collisions. In order to determine which sprites have collided, the sprite collision register ($D01F) must be examined. For collision purposes, multicolour data 01 is cosidered transparent, even though it is visible on the screen. In order for this to cause an IRQ, bit 1 of $D01A must be set by the programmer.

BIT 0: RASTER COMPARE

This bit is set to 1 when the screen raster count reaches the value written into the raster compare register ($D012). In order for this to cause an IRQ, bit 0 of $D01A must be set by the programmer.

$D01A: VIC INTERRUPT MASK REGISTER 53274

The bits in this register correspond with the bits in the Interrupt Flag register ($D019). In order to allow a set interrupt flag bit to cause a processor interrupt request (IRQ), the corresponding bit in this register must be set to 1. Once an interrupt flag is set, it can only be cleared by writing a 1 to the corresponding bit in this register.

$D01B: SPRITE - BACKGROUND DISPLAY PRIORITY 53275

This register determines whether a sprite or screen data will be displayed when both share the same screen co-ordinates. Setting a bit to 1 causes the corresponding sprite to be displayed, while clearing the bit causes the screen data to be displayed. The effect is that of passing the sprites in front of or behind screen data. Note that

sprite 0 will always be in front of sprite 1 etc.

$D01C: SPRITE MULTICOLOUR MODE 53276

Multicolour sprite mode can be enabled for a particular sprite by setting the corresponding bit in this register to 1. In this mode, the sprite horizontal resolution is halved to 12 pixels, whilst the vertical resolution remains the same at 21 pixels. The same number of bits are used to determine the sprite, but they are interpreted in pairs, each pair forming one pixel. Up to 4 colours can be displayed in the sprite (including transparent), but two of these must be shared with all the other multicolour sprites. The colour displayed for each bit pair is shown below:

BIT PAIR	COLOUR
00	Transparent
01	Multicolour #1 ($D025)
10	Sprite colour
11	Multicolour #2 ($D026)

The sprite colour is the colour that is used to display a high resolution sprite and is determined by registers $D027 - $D02E.

$D01D: SPRITES 0-7 EXPAND, HORIZONTAL DIRECTION 53277

Setting a bit in this register causes the corresponding sprite to be expanded in the horizontal (X) direction. This expansion doubles the width of the sprite, but affords no increase in resolution.

$D01E: SPRITE TO SPRITE COLLISION 53278

A bit is set to 1 in this register whenever the corresponding sprite has collided with another sprite. A 'collision' occurs whenever visible data from two sprites overlaps (with the exception of '01' data in multicolour sprites). Thus if sprites 0 and 4 were to collide, bits 0 and 4 of this register would be set. With collisions between 3 or more sprites, the position of each sprite should be examined in order to determine exactly which sprites are touching each other. ie. sprite 2 may collide with sprites 3 and 6, but sprite 6 is not touching sprite 3. This register should be used in conjunction with bit 2 of $D019.

$D01F: SPRITE TO SCREEN DATA COLLISION 53279

A bit is set to 1 in this register whenever the corresponding sprite has collided with foreground screen data. Thus if sprite 0 were to collide with an object on the screen, bit 0 of this register would be set. This register should be used in congunction with bit 1 of $D019.

$D020: BORDER COLOUR 53280

The value in this register (0-15) determines the colour of the border that surrounds the central display screen. When the screen is blanked using bit 4 of $D011, the whole screen becomes this colour. Note that only the lower four bits of this register have any meaning.

$D021: BACKGROUND COLOUR 0 53281

The value in this register (0-15) determines the colour of the central display screen. This is the colour seen in both the text and bitmap modes. Note that only the lower four bits of this register have any meaning.

$D022: BACKGROUND COLOUR 1 53282

The value in this register (0-15) determines the colour of '01' bit pair pixels in all multicolour modes. It is also the background colour for characters in extended colour mode having screen codes 64 -127. Note that only the lower four bits of this register have any meaning.

$D023: BACKGROUND COLOUR 2 53283

The value in this register (0-15) determines the colour of '10' bit pair pixels in all multicolour modes. It is also the background colour for characters in extended colour mode having screen codes 128 -191. Note that only the lower four bits of this register have any meaning.

$D024: BACKGROUND COLOUR 3 53284

The value in this register (0-15) determines the background colour for characters in extended colour mode having screen codes 192 - 255. Note that only the lower four bits of this register have any meaning.

$D025: SPRITE MULTICOLOUR REGISTER 0 53285

The value in this register (0-15) determines the colour displayed by a '01' bit pair in multicolour sprite graphics. Note that only the lower four bits of this register have

any meaning.

$D026: SPRITE MULTICOLOUR REGISTER 1 53286

The value in this register (0-15) determines the colour displayed by a '11' bit pair in multicolour sprite graphics. Note that only the lower four bits of this register have any meaning.

$D027: SPRITE 0 COLOUR 53287

The value in this register (0-15) determines the foreground colour of sprite 0. It also determines the '10' colour displayed by the sprite when in multicolour mode. Note that only the lower four bits of this register have any meaning.

$D028: SPRITE 1 COLOUR 53288

The value in this register (0-15) determines the foreground colour of sprite 1. It also determines the '10' colour displayed by the sprite when in multicolour mode. Note that only the lower four bits of this register have any meaning.

$D029: SPRITE 2 COLOUR 53289

The value in this register (0-15) determines the foreground colour of sprite 2. It also determines the '10' colour displayed by the sprite when in multicolour mode. Note that only the lower four bits of this register have any meaning.

$D02A: SPRITE 3 COLOUR 53290

The value in this register (0-15) determines the foreground colour of sprite 3. It also determines the '10' colour displayed by the sprite when in multicolour mode. Note that only the lower four bits of this register have any meaning.

$D02B: SPRITE 4 COLOUR 53291

The value in this register (0-15) determines the foreground colour of sprite 4. It also determines the '10' colour displayed by the sprite when in multicolour mode. Note that only the lower four bits of this register have any meaning.

$D02C: SPRITE 5 COLOUR 53292

The value in this register (0-15) determines the foreground colour of sprite 5. It also determines the '10' colour displayed by the sprite when in multicolour mode. Note that

only the lower four bits of this register have any meaning.

$D02D: SPRITE 6 COLOUR 53293

The value in this register (0-15) determines the foreground colour of sprite 6. It also determines the '10' colour displayed by the sprite when in multicolour mode. Note that only the lower four bits of this register have any meaning.

$D02E: SPRITE 7 COLOUR 53294

The value in this register (0-15) determines the foreground colour of sprite 7. It also determines the '10' colour displayed by the sprite when in multicolour mode. Note that only the lower four bits of this register have any meaning.

6581 SID CHIP

$D400: VOICE 1 FREQUENCY LOW BYTE 54272

This register is used in conjunction with the next register ($D401) to form a 16 bit number. This value is used to control the frequency output of voice 1. The frequency can be determined by the following formula:

FREQUENCY = REGISTER * o2 CLOCK / 1677216 Hz.

Where REGISTER is the value of the 16 bit number stored in $D400 and $D401, o2 CLOCK is the sysstem o2 clock frequency (1.02273 MHz for NTSC machines, and 0.98525 MHz for PAL machines).

As an approximation, a low value in the frequency registers gives a low note, and a high value gives a high note.

$D401: VOICE 1 FREQUENCY HIGH BYTE 54273

This is the high order byte of the 16 bit frequency register for voice 1. The value of the 16 bit number can be calculated from BASIC using the following formula:

REGISTER = (HIGH BYTE * 256) + LOW BYTE

$D402: VOICE 1 PULSE WIDTH LOW BYTE 54274

When voice 1 is played using the pulse waveform, this register (and the following one) are used to form a 12 bit number. This value is used to control the pulse width (also

called duty cycle) for voice 1.

The duty cycle of the waveform indicates what proportion of the total cycle time will be spent in the 'high' state. The range of values is from 0%, ie. no time spent high, to 100%, ie. always high in 4096 steps.

The following formula can be used to calculate the proportion of the cycle spent in the high state:

PULSE WIDTH = REGISTER / 40.95 %

Note that the frequency or cycle time for the note is determined by the frequency control register ($D400-$D401).

$D403: VOICE 1 PULSE WIDTH HIGH NYBBLE 54275

Bits 0-3 of this register form the high order nybble of the 12 bit pulse width register for voice 1. The value of the 16 bit number can be calculated from BASIC using the following formula:

REGISTER = (HIGH NYBBLE * 256) + LOW BYTE

Note that the high order nybble (bits 4-7) are unused.

$D404: VOICE 1 CONTROL REGISTER 54276

BIT 7: SELECT RANDOM NOISE

Setting this bit to 1 enables the noise waveform for voice 1. This waveform produces a random noise output, centred around the voice 1 frequency. It is not recommended that any other waveform be selected whilst noise is enabled, since this can cause the oscillator to lock up.

BIT 6: SELECT PULSE WAVEFORM

Setting this bit to 1 enables the pulse waveform for voice 1. This waveform produces a rectangular pulse output, centred at the voice 1 frequency. The pulse width or duty cycle can be varied by using the pulse width register ($D402-$D403). Values can be made to vary from dc to a square wave. Selecting additional waveforms while pulse is enabled is not additive, but produces a logical ANDing of the result.

BIT 5: SELECT SAWTOOTH WAVEFORM

Setting this bit to 1 enables the sawtooth waveform for voice 1. Selecting additional waveforms while sawtooth is enabled is not additive, but produces a logical ANDing of the result.

BIT 4: SELECT TRIANGLE WAVEFORM

Setting this bit to 1 enables the triangle waveform for voice 1. Selecting additional waveforms while triangle is enabled is not additive, but produces a logical ANDing of the result.

BIT 3: TEST BIT

Setting this bit to 1 causes the output of voice 1 to be disabled. It is only enabled again when this bit is cleared. This is useful in generating very complex waveforms, software speech synthesis, synchronisation of the voice to external events etc. This bit must be set in order to reset voice 1 if it has locked up from, say selecting multiple waveforms.

BIT 2: RING MODULATE VOICE 1 WITH VOICE 3

Setting this bit to 1 causes the triangle waveform of voice 1 to be replaced with a ring modulated combination of voices 1 and 3. By varying the frequency of voice 1 with respect to voice 3, a range of non-harmonic overtones can be produced for special effects etc. Note that only the frequency register of voice 3 has any influence on ring modulation, and the voice 1 triangle waveform must be selected.

BIT 1: SYNCHRONISE VOICE 1 WITH VOICE 3 FREQUENCY

Setting this bit to 1 causes the fundamental frequency of voice 1 to be synchronised with the fundamental frequency of voice 3. By varying the frequency of voice 1 with respect to voice 3, a range of harmonics can be produced from voice 1. Note that only the frequency register of voice 3 has any influence on synchronisation.

BIT 0: GATE BIT

Setting this bit to 1 causes the sound output from voice 1 to be triggered (gated). This will start the attack/decay/sustain part of the cycle. Clearing this bit to 0 causes the release part of the cycle to be started.

$D405: VOICE 1 ENVELOPE 54277

BITS 7-4: ATTACK CYCLE DURATION

The value in these four bits determines the time taken for the note to reach its peak volume once the voice has been gated. This is known as the attack phase of the envelope cycle. A value of 0 here will cause the attack phase to last 2 milliseconds, while a value of 15 causes the phase to last 8 seconds.

BITS 3-0: DECAY CYCLE DURATION

The value in these four bits determines the time taken for the note to decay from its peak value to its sustain level. This is known as the decay phase of the envelope cycle. A value of 0 here will cause the attack phase to last 6 milliseconds, while a value of 15 causes the phase to last 24 seconds.

$D406: VOICE 1 ENVELOPE 54278

BITS 7-4: SUSTAIN VOLUME LEVEL

The value in these four bits determine the volume of the note during the sustain phase of the envelope cycle. The volume levels are the same as those of the master volume register ($D418). The sustain phase will continue until the gate bit of $D404 is cleared to zero.

BITS 3-0: RELEASE CYCLE DURATION

The value in these four bits determines the time taken for the note to decay from its sustain level to zero volume. This is known as the release phase of the envelope cycle. The release phase is only started once the gate bit of $D404 is cleared to zero. A value of 0 here will cause the attack phase to last 6 milliseconds, while a value of 15 causes the phase to last 24 seconds.

$D407: VOICE 2 FREQUENCY LOW BYTE 54279

This register is used in conjunction with the next register ($D408) to form a 16 bit number. This value is used to control the frequency output of voice 1. The frequency can be determined by the following formula:

FREQUENCY = REGISTER * o2 CLOCK / 1677216 Hz.

Where REGISTER is the value of the 16 bit number stored in $D407 and $D408, o2 CLOCK is the system o2 clock frequency (1.02273 MHz for NTSC machines, and 0.98525 MHz for PAL machines).

As an approximation, a low value in the frequency registers gives a low note, and a high value gives a high note.

$D408: VOICE 2 FREQUENCY HIGH BYTE 54280

This is the high order byte of the 16 bit frequency register for voice 2. The value of the 16 bit number can be calculated from BASIC using the following formula:

REGISTER = (HIGH BYTE * 256) + LOW BYTE

$D409: VOICE 2 PULSE WIDTH LOW BYTE 54281

When voice 2 is played using the pulse waveform, this register (and the following one) are used to form a 12 bit number. This value is used to control the pulse width (also called duty cycle) for voice 2.

The duty cycle of the waveform indicates what proportion of the total cycle time will be spent in the 'high' state. The range of values is from 0%, ie. no time spent high, to 100%, ie. always high in 4096 steps.

The following formula can be used to calculate the proportion of the cycle spent in the high state:

PULSE WIDTH = REGISTER / 40.95 %

Note that the frequency or cycle time for the note is determined by the frequency control register ($D407-$D408).

$D40A: VOICE 2 PULSE WIDTH HIGH NYBBLE 54282

Bits 0-3 of this register form the high order nybble of the 12 bit pulse width register for voice 2. The value of the 16 bit number can be calculated from BASIC using the following formula:

REGISTER = (HIGH NYBBLE * 256) + LOW BYTE

Note that the high order nybble (bits 4-7) are unused.

$D40B: VOICE 2 CONTROL REGISTER 54283

BIT 7: SELECT RANDOM NOISE

Setting this bit to 1 enables the noise waveform for voice 2. This waveform produces a random noise output, centred around the voice 2 frequency. It is not recommended that any other waveform be selected whilst noise is enabled, since this can cause the oscillator to lock up.

BIT 6: SELECT PULSE WAVEFORM

Setting this bit to 1 enables the pulse waveform for voice 2. This waveform produces a rectangular pulse output, centred at the voice 2 frequency. The pulse width or duty cycle can be varied by using the pulse width register ($D409-$D40A). Values can be made to vary from dc to a square wave. Selecting additional waveforms while pulse is enabled is not additive, but produces a logical ANDing of the result.

BIT 5: SELECT SAWTOOTH WAVEFORM

Setting this bit to 1 enables the sawtooth waveform for voice 2. Selecting additional waveforms while sawtooth is enabled is not additive, but produces a logical ANDing of the result.

BIT 4: SELECT TRIANGLE WAVEFORM

Setting this bit to 1 enables the triangle waveform for voice 2. Selecting additional waveforms while triangle is enabled is not additive, but produces a logical ANDing of the result.

BIT 3: TEST BIT

Setting this bit to 1 causes the output of voice 2 to be disabled. It is only enabled again when this bit is cleared. This is useful in generating very complex waveforms, software speech synthesis, synchronisation of the voice to external events etc. This bit must be set in order to reset voice 2 if it has locked up from, say selecting multiple waveforms.

BIT 2: RING MODULATE VOICE 1 WITH VOICE 1

Setting this bit to 1 causes the triangle waveform of voice 2 to be replaced with a ring modulated combination of voices 2 and 1. By varying the frequency of voice 2 with respect to voice 1, a range of non-harmonic overtones can be

produced for special effects etc. Note that only the frequency register of voice 1 has any influence on ring modulation, and the voice 2 triangle waveform must be selected.

BIT 1: SYNCHRONISE VOICE 2 WITH VOICE 1 FREQUENCY

Setting this bit to 1 causes the fundamental frequency of voice 2 to be synchronised with the fundamental frequency of voice 1. By varying the frequency of voice 2 with respect to voice 1, a range of harmonics can be produced from voice 2. Note that only the frequency register of voice 1 has any influence on synchronisation.

BIT 0: GATE BIT

Setting this bit to 1 causes the sound output from voice 2 to be triggered (gated). This will start the attack/decay/sustain part of the cycle. Clearing this bit to 0 causes the release part of the cycle to be started.

$D40C: VOICE 2 ENVELOPE 54284

BITS 7-4: ATTACK CYCLE DURATION

The value in these four bits determines the time taken for the note to reach its peak volume once the voice has been gated. This is known as the attack phase of the envelope cycle. A value of 0 here will cause the attack phase to last 2 milliseconds, while a value of 15 causes the phase to last 8 seconds.

BITS 3-0: DECAY CYCLE DURATION

The value in these four bits determines the time taken for the note to decay from its peak value to its sustain level. This is known as the decay phase of the envelope cycle. A value of 0 here will cause the attack phase to last 6 milliseconds, while a value of 15 causes the phase to last 24 seconds.

$D40D: VOICE 2 ENVELOPE 54285

BITS 7-4: SUSTAIN VOLUME LEVEL

The value in these four bits determine the volume of the note during the sustain phase of the envelope cycle. The volume levels are the same as those of the master volume register ($D418). The sustain phase will continue until the

gate bit of $D40B is cleared to zero.

BITS 3-0: RELEASE CYCLE DURATION

The value in these four bits determines the time taken for the note to decay from its sustain level to zero volume. This is known as the release phase of the envelope cycle. The release phase is only started once the gate bit of $D40B is cleared to zero. A value of 0 here will cause the attack phase to last 6 milliseconds, while a value of 15 causes the phase to last 24 seconds.

$D40E: VOICE 3 FREQUENCY LOW BYTE 54286

This register is used in conjunction with the next register ($D40F) to form a 16 bit number. This value is used to control the frequency output of voice 1. The frequency can be determined by the following formula:

FREQUENCY = REGISTER * o2 CLOCK / 1677216 Hz.

Where REGISTER is the value of the 16 bit number stored in $D40E and $D40F, o2 CLOCK is the system o2 clock frequency (1.02273 MHz for NTSC machines, and 0.98525 MHz for PAL machines).

As an approximation, a low value in the frequency registers gives a low note, and a high value gives a high note.

$D40F: VOICE 3 FREQUENCY HIGH BYTE 54287

This is the high order byte of the 16 bit frequency register for voice 3. The value of the 16 bit number can be calculated from BASIC using the following formula:

REGISTER = (HIGH BYTE * 256) + LOW BYTE

$D410: VOICE 3 PULSE WIDTH LOW BYTE 54288

When voice 3 is played using the pulse waveform, this register (and the following one) are used to form a 12 bit number. This value is used to control the pulse width (also called duty cycle) for voice 3.

The duty cycle of the waveform indicates what proportion of the total cycle time will be spent in the 'high' state. The range of values is from 0%, ie. no time spent high, to 100%, ie. always high in 4096 steps.

The following formula can be used to calculate the proportion of the cycle spent in the high state:

PULSE WIDTH = REGISTER / 40.95 %

Note that the frequency or cycle time for the note is determined by the frequency control register ($D40E-$D40F).

$D411: VOICE 3 PULSE WIDTH HIGH NYBBLE 54289

Bits 0-3 of this register form the high order nybble of the 12 bit pulse width register for voice 3. The value of the 12 bit number can be calculated from BASIC using the following formula:

REGISTER = (HIGH NYBBLE * 256) + LOW BYTE

Note that the high order nybble (bits 4-7) are unused.

$D412: VOICE 3 CONTROL REGISTER 54290

BIT 7: SELECT RANDOM NOISE

Setting this bit to 1 enables the noise waveform for voice 3. This waveform produces a random noise output, centred around the voice 3 frequency. It is not recommended that any other waveform be selected whilst noise is enabled, since this can cause the oscillator to lock up.

BIT 6: SELECT PULSE WAVEFORM

Setting this bit to 1 enables the pulse waveform for voice 3. This waveform produces a rectangular pulse output, centred at the voice 3 frequency. The pulse width or duty cycle can be varied by using the pulse width register ($D410-$D411). Values can be made to vary from dc to a square wave. Selecting additional waveforms while pulse is enabled is not additive, but produces a logical ANDing of the result.

BIT 5: SELECT SAWTOOTH WAVEFORM

Setting this bit to 1 enables the sawtooth waveform for voice 3. Selecting additional waveforms while sawtooth is enabled is not additive, but produces a logical ANDing of the result.

BIT 4: SELECT TRIANGLE WAVEFORM

Setting this bit to 1 enables the triangle waveform for

voice 3. Selecting additional waveforms while triangle is enabled is not additive, but produces a logical ANDing of the result.

BIT 3: TEST BIT

Setting this bit to 1 causes the output of voice 3 to be disabled. It is only enabled again when this bit is cleared. This is useful in generating very complex waveforms, software speech synthesis, synchronisation of the voice to external events etc. This bit must be set in order to reset voice 3 if it has locked up from, say selecting multiple waveforms.

BIT 2: RING MODULATE VOICE 3 WITH VOICE 2

Setting this bit to 1 causes the triangle waveform of voice 3 to be replaced with a ring modulated combination of voices 3 and 2. By varying the frequency of voice 3 with respect to voice 2, a range of non-harmonic overtones can be produced for special effects etc. Note that only the frequency register of voice 2 has any influence on ring modulation, and the voice 3 triangle waveform nust be selected.

BIT 1: SYNCHRONISE VOICE 3 WITH VOICE 2 FREQUENCY

Setting this bit to 1 causes the fundamental frequency of voice 3 to be synchronised with the fundamental frequency of voice 2. By varying the frequency of voice 3 with respect to voice 2, a range of harmonics can be produced from voice 3. Note that only the frequency register of voice 2 has any influence on synchronisation.

BIT 0: GATE BIT

Setting this bit to 1 causes the sound output from voice 3 to be triggered (gated). This will start the attack/decay/sustain part of the cycle. Clearing this bit to 0 causes the release part of the cycle to be started.

$D413: VOICE 3 ENVELOPE 54291

BITS 7-4: ATTACK CYCLE DURATION

The value in these four bits determines the time taken for the note to reach its peak volume once the voice has been gated. This is known as the attack phase of the envelope

cycle. A value of 0 here will cause the attack phase to last 2 milliseconds, while a value of 15 causes the phase to last 8 seconds.

BITS 3-0: DECAY CYCLE DURATION

The value in these four bits determines the time taken for the note to decay from its peak value to its sustain level. This is known as the decay phase of the envelope cycle. A value of 0 here will cause the attack phase to last 6 milliseconds, while a value of 15 causes the phase to last 24 seconds.

$D414: VOICE 3 ENVELOPE 54292

BITS 7-4: SUSTAIN VOLUME LEVEL

The value in these four bits determine the volume of the note during the sustain phase of the envelope cycle. The volume levels are the same as those of the master volume register ($D418). The sustain phase will continue until the gate bit of $D412 is cleared to zero.

BITS 3-0: RELEASE CYCLE DURATION

The value in these four bits determines the time taken for the note to decay from its sustain level to zero volume. This is known as the release phase of the envelope cycle. The release phase is only started once the gate bit of $D412 is cleared to zero. A value of 0 here will cause the attack phase to last 6 milliseconds, while a value of 15 causes the phase to last 24 seconds.

$D415: FILTER CUTOFF FREQUENCY LOW BITS 54293

BITS 7-3: UNUSED

BITS 2-0: FILTER FREQUENCY LOW BITS

These are the 3 low order bits of the 11 bit filter cutoff frequency register, the high order bits being held in $D416. This register controls the cutoff frequency for the programmable low and high pass filters, and the centre frequency for the bandpass filter. The cutoff frequency ranges from 30 Hz with a register value of 0, to 12 KHz with a register value of 2047.

$D416: FILTER CUTOFF FREQUENCY LOW BITS 54294

This is the high order byte of the 11 bit filter cutoff frequency register.

$D417: FILTER RESONANCE CONTROL 54295

BITS 7-4: FILTER RESONANCE

The value in these 4 bits control the resonance of the filter. A value of 0 gives no resonance, while a value of 15 gives maximum resonance. The effect of resonance is to emphasize the frequencies around the cutoff value, causing a sharper filtering effect.

BIT 3: FILTER EXTERNAL INPUT

Setting this bit to 1 causes the sound from the external audio input to be processed through the programmable filter.

BIT 2: FILTER VOICE 3

Setting this bit to 1 causes the sound from voice 3 to be processed through the programmable filter.

BIT 1: FILTER VOICE 2

Setting this bit to 1 causes the sound from voice 2 to be processed through the programmable filter.

BIT 0: FILTER VOICE 1

Setting this bit to 1 causes the sound from voice 1 to be processed through the programmable filter.

$D418: VOLUME / FILTER REGISTER 54296

BIT 7: DISABLE VOICE 3 OUTPUT

Setting this bit to 1 causes the audio output from voice 3 to be disabled. This allows voice 3 to be used for ring modulation etc without actually being heard.

BIT 6: SELECT HIGH PASS FILTER

Setting this bit to 1 causes the filter to operate in high pass mode. This means that frequencies above the cutoff frequency are left alone, while frequencies below the cutoff frequency are attenuated at the rate of 12 dB per octave.

BIT 5: SELECT BAND PASS FILTER

Setting this bit to 1 causes the filter to operate in band pass mode. This means that frequencies above and below the cutoff frequency are attenuated at the rate of 12 dB per octave.

BIT 4: SELECT LOW PASS FILTER

Setting this bit to 1 causes the filter to operate in low pass mode. This means that frequencies below the cutoff frequency are left alone, while frequencies above the cutoff frequency are attenuated at the rate of 12 dB per octave.

BITS 3-0: SELECT OUTPUT VOLUME

The value in these four bits determines the overall output volume of all three voices. A value of 0 will produce no sound, whilst a value of 15 will produce maximum volume.

$D419: A/D CONVERTER 1 54297

This register can be read by the microprocessor to determine the position of a potentiometer attached to the POTX line of the 6581 chip. The pot varies the voltage on the pin between 0 and +5V, and the register returns linear values between 0 and 255 for these voltages. This register is updated by the chip every 512 o2 clock cycles.

$D41A: A/D CONVERTER 2 54298

This register can be read by the microprocessor to determine the position of a potentiometer attached to the POTY line of the 6581 chip. The pot varies the voltage on the pin between 0 and +5V, and the register returns linear values between 0 and 255 for these voltages. This register is updated by the chip every 512 o2 clock cycles.

$D41B: VOICE 3 OSCILLATOR OUTPUT 54299

This register allows the upper 8 bits of the output from voice 3. (Most of the other registers on this chip are write only, and will return a value of 0 if read from.) The pattern of variation of the numbers in this register is dependant on the waveform selected.

A sawtooth waveform will produce values increasing from 0 to 255, then starting again from 0. A trianle waveform will produce numbers increasing from 0 to 255 and then decreasing back to 0 again. A pulse waveform will produce a value alternating between 0 and 255. A noise waveform will

produce a series of random numbers between 0 and 255. The rate at which these values change is determined by the frequency of the voice.

$D41C: VOICE 3 ENVELOPE OUTPUT 54300

This register allows the output of the voice 3 envelope generator to be read. This can be added to other voice registers to produce wah-wah etc. effects. By adding this output to the filter cutoff frequency, harmonic envelopes can be produced.

6526 CIA CHIP #1

$DC00: DATA PORT A 56320

This register is used in the process of reading the keyboard matrix. The column to be read is written to this register, and the keyboard row is read from data port B ($DC01). The procedure is detailed at $EA87 - $ECB8 in the ROM GUIDE.

Additionally, the following bits of this register are used for other purposes:

BITS 7-6: READ PADDLES ON GAME PORT 1 OR 2

These bits determine whether the SID chip reads the game paddles from game port 1 or 2. This is needed because there can be four paddles attached to the computer, but there are only 2 A/D converters on the SID chip. Setting these bits to the value 01, allows the paddles on port 1 to be read. Setting these bits to the value 10, allows the paddles on port 2 to be read.

BIT 4: JOYSTICK 2 FIRE BUTTON

Reading this bit will return a value of 1 if the fire button on a joystick attached to game port 2 is pressed.

BITS 3-0: JOYSTICK 2 DIRECTION

Reading these bits will return a value dependant on the direction selected on a joystick attached to game port 2.

BITS 3-2: PADDLE FIRE BUTTONS

Reading these bits will return a 1 if the relevant fire button is pressed on the paddles attached to game port 2.

$DC01: DATA PORT B 56321

This register is used in the process of reading the keyboard matrix. The column to be read is written to data port A ($DC00), and the keyboard row is read from this register. The procedure is detailed at $EA87 - ECB8 in the ROM GUIDE.

Additionally, the following bits of this register are used for other purposes:

BIT 7: TIMER B TOGGLE/PULSE OUTPUT

This bit can be used by the CIA timer B as an output. It will either toggle the value here between 0 and 1 or pulse the bit for one machine cycle, depending on the setup of bits 1 and 2 of $DC0F.

BIT 6: TIMER A TOGGLE/PULSE OUTPUT

This bit can be used by the CIA timer A as an output. It will either toggle the value here between 0 and 1 or pulse the bit for one machine cycle, depending on the setup of bits 1 and 2 of $DC0E.

BIT 4: JOYSTICK 1 FIRE BUTTON

Reading this bit will return a value of 1 if the fire button on a joystick attached to game port 1 is pressed.

BITS 3-0: JOYSTICK 1 DIRECTION

Reading these bits will return a value dependant on the direction selected on a joystick attached to game port 1.

BITS 3-2: PADDLE FIRE BUTTONS

Reading these bits will return a 1 if the relevant fire button is pressed on the paddles attached to game port 1.

$DC02: DATA DIRECTION REGISTER A 56322

This register controls the direction of flow of data over port A. By setting a bit of this register to 1, the corresponding bit of the data port is defined as an output. The default value of this register is #FF, ie all outputs.

$DC03: DATA DIRECTION REGISTER B 56323

This register controls the direction of flow of data over port B. By setting a bit of this register to 1, the corresponding bit of the data port is defined as an output.

The default value of this register is #00, ie all inputs.

$DC04: TIMER A LOW BYTE 56324

This register along with $DC05 form a 16 bit timer. The action of this timer is described in detail on page 524. Timer A is used by the operating system to generate an interrupt every 60th of a second. This interrupt is used for reading the keyboard etc.

Timer A is also used for timing during the cassette tape and serial bus I/O routines.

$DC05: TIMER A HIGH BYTE 56325

This is the high byte of the 16 bit timer A.

$DC06: TIMER B LOW BYTE 563246

This register along with $DC07 form a 16 bit timer. The action of this timer is described in detail on page 524. Timer B is used by the operating system for timing during cassette tape and serial bus I/O.

$DC07: TIMER B HIGH BYTE 56327

This is the high byte of the 16 bit timer B.

$DC08: TOD CLOCK 10THS OF SECOND 56328

This is the first of four registers that comprise the Time Of Day (TOD) clock. This is a timer, organised into registers of hours, minutes, seconds and 10ths of seconds. Each register counts upwards in BCD format. This enables easy conversion into ASCII digits.

$DC09: TOD CLOCK SECONDS REGISTER 56329

This register indicates seconds in the Time of Day (TOD) clock in BCD format. Bit 7 of this register is not used.

$DC0A: TOD CLOCK MINUTES REGISTER 56330

This register indicates minutes in the Time of Day (TOD) clock in BCD format. Bit 7 of this register is not used.

$DC0B: TOD CLOCK HOURS AND AM/PM 56331

This register indicates hours in the Time of Day (TOD) clock in 12 hour BCD format. In addition, bit 7 is used to indicate AM or PM (0=AM, 1=PM). Bits 5 and 6 of this register are not used.

$DC0C: SERIAL DATA REGISTER 56332

This register forms a synchronous serial I/O port. The direction of data flow is controlled by bit 6 of $DC0E. In input mode, data is read from the SP pin of the chip whenever a pulse appears on the CNT pin. After 8 bits have been read, the byte can be read from this register. In the output mode, a CNT pulse appears whenever data is available at the SP pin.

The speed of data transfer on the port is determined by timer A, with a maximum speed of the system o2 clock / 4. Data is always transferred starting with the MSB (bit 7). Once a byte has been sent or received, a flag is set in the interrupt control register ($DC0D).

The 6526 serial data register is not used by the Commodore 64, preferring to use parallel I/O ports for asynchronous serial I/O.

$DC0D: INTERRUPT CONTROL REGISTER 56333

BIT 7: IRQ OCCURRED FLAG

If one of the 5 sources of interrupt on the 6526 has caused an interrupt, then this bit is set to 1. This can be used for polling during an interrupt routine to determine the source of the interrupt.

In order for an interrupt bit in this register to cause an IRQ at the processor, the IRQ output must be set for that bit. Similarly, to prevent it from causing an IRQ, that bit must be cleared. Both actions are performed by writing a 1 to that bit. The action taken is determined by writing a value to this bit. Write a value 0, and other bits written will be cleared. Write a value 1, and other bits written will be set.

BITS 6-5: UNUSED

BIT 4: FLAG1

This bit will be set to 1 whenever a signal is received on the CIA#1 FLAG line. This line is used as the cassette read

line, and also as the serial bus SRQ input.

Writing a 1 to this bit will enable or disable the FLAG1 IRQ, depending on the value written to bit 7.

BIT 3: SERIAL DATA REGISTER

This bit will be set to 1 whenever the chip serial port has finished reading or writing a byte of data.

Writing a 1 to this bit will enable or disable the SDR IRQ, depending on the value written to bit 7.

BIT 2: TOD CLOCK ALARM

This bit will be set to 1 whenever the chip TOD clock has reached the time set in the TOD alarm registers.

Writing a 1 to this bit will enable or disable the TOD alarm IRQ, depending on the value written to bit 7.

BIT 1: TIMER A

This bit will be set to 1 whenever timer A reaches zero.

Writing a 1 to this bit will enable or disable the timer A IRQ depending on the value written to bit 7.

BIT 0: TIMER B

This bit will be set to 1 whenever timer B reaches zero.

Writing a 1 to this bit will enable or disable the timer B IRQ depending on the value written to bit 7.

$DC0E: CONTROL REGISTER A 56334

The 8 bits in this register are used to control timer A.

BIT 7: TOD FREQUENCY

This bit determines whether a 50 Hz (1) or 60Hz (0) signal has to be applied to the TOD pin in order for the TOD clock to keep accurate time.

BIT 6: SERIAL PORT MODE

This bit is used to determine whether the on-chip serial port operates in input mode or output mode. A value of 0

indicates input, and 1 indicates output.

BIT 5: TIMER A COUNT MODE

This bit determines the manner in which timer A counts down. A value of 0 here, and the timer counts system o2 clock pulses. A value of 1, and the timer counts pulses on the external CNT line.

BIT 4: FORCE LOAD TIMER A

Writing a 1 to this bit forces the contents of the timer A low and high byte registers to be loaded into the timer itself. Reading this bit has no effect.

BIT 3: TIMER A RUN MODE

Setting this bit to 1 causes timer A to operate in one-shot mode. This means that once the timer has reached zero, it will stop. The original value is loaded back into the timer.

Clearing this bit to 0 causes the timer to operate in continuous mode. This means that once the timer has reached zero, the starting value is re-loaded into the timer, and counting is continued.

BIT 2: TIMER A OUTPUT ON PB6

This bit is only operative when bit 1 is set. Writing a 1 here causes the current value of PB6 to be toggled whenever the timer reaches zero. Writing a 0 here causes a single pulse of approximately 1 microsecond (ie. 1 clock cycle) to appear on the port.

BIT 1: TIMER A OUTPUT ON PB6

Setting this bit to 1 enables the timer output on PB6. This output occurs whenever the timer reaches zero, but its nature is determined by bit 2

BIT 0: START / STOP TIMER A

Setting this bit to 1 will start the timer counting from the value loaded into it dowm to zero. Clearing this bit to 0 will stop the timer from counting.

$DC0F: CONTROL REGISTER B 56335

The 8 bits in this register are used to control timer B and the TOD clock.

BIT 7: SET ALARM / TOD CLOCK

This bit determines what happens when the TOD clock registers are written to. If this bit is set to 1, then writing to the registers sets the alarm. If this bit is cleared to 0, then writing to the registers sets the TOD clock.

BITS 6-5: TIMER B COUNT MODE

These four bits are used to determine what is counted by timer B. The four modes are as follows:

Count system o2 clock pulses 00 -

Count CNT pulses 01 -

Count timer A underflow pulses 10 -

Count timer A underflows while CNT is high 11 -

BIT 4: FORCE LOAD TIMER B

Writing a 1 to this bit forces the contents of the timer B low and high byte registers to be loaded into the timer itself. Reading this bit has no effect.

BIT 3: TIMER B RUN MODE

Setting this bit to 1 causes timer B to operate in one-shot mode. This means that once the timer has reached zero, it will stop. The original value is loaded back into the timer.

Clearing this bit to 0 causes the timer to operate in continuous mode. This means that once the timer has reached zero, the starting value is re-loaded into the timer, and counting is continued.

BIT 2: TIMER B OUTPUT ON PB7

This bit is only operative when bit 1 is set. Writing a 1 here causes the current value of PB7 to be toggled whenever the timer reaches zero. Writing a 0 here causes a single pulse of approximately 1 microsecond (ie. 1 clock cycle) to appear on the port.

BIT 1: TIMER B OUTPUT ON PB7

Setting this bit to 1 enables the timer output on PB7. This output occurs whenever the timer reaches zero, but its nature is determined by bit 2

BIT 0: START / STOP TIMER B

Setting this bit to 1 will start the timer counting from the value loaded into it dowm to zero. Clearing this bit to 0 will stop the timer from counting.

6526 CIA CHIP #2

$DD00: DATA PORT A — 56576

This register is used to handle the serial bus I/O, the RS-232 data output line and the VIC II chip memory management lines.

The operation of the serial bus is explained in detail on page 517, and the RS-232 port is explained on page 513..

BIT 7: SERIAL BUS DATA INPUT

BIT 6: SERIAL BUS CLK PULSE INPUT

BIT 5: SERIAL BUS DATA OUTPUT

BIT 4: SERIAL BUS CLK PULSE OUTPUT

BIT 3: SERIAL BUS ATN OUTPUT

BIT 2: RS-232 DATA OUTPUT

BITS 1-0: VIC II CHIP MEMORY BANK SELECT

These two bits are used to control which of the four 16K blocks of memory is 'seen' by the VIC II chip. The default bank for the VIC II chip is 0 ($0000 - $3FFF), but, for hires graphics especially, this conflicts with program memory usage. A suitable solution to this is to move the graphics bank somewhere out of the way. A favoured location for this is bank 3 ($C000-$FFFF), where the graphics screen can sit in the RAM that is hidden underneath the Commodore 64 ROM. The bit values needed to select each bank are shown below:

00 - BANK 3 ($C000 - $FFFF)
01 - BANK 2 ($8000 - $BFFF)
10 - BANK 1 ($4000 - $7FFF)
11 - BANK 0 ($0000 - $3FFF)

$DD01: DATA PORT B 56577

This register is used to handle the RS-232 serial I/O port, and to provide the user-defined general purpose I/O port. All 8 bits of this port appear at the user port connector on the back of the Commodore 64, along with the serial ports and CNT strobes of both CIA chips. The RS-232 port is assigned to this register in the following way:

BIT 7 - Data Set Ready (DSR)
BIT 6 - Clear To Send (CTS)
BIT 4 - Carrier Detect
BIT 3 - Ring Indicator
BIT 2 - Data Terminal Ready (DTR)
BIT 1 - Request To Send (RTS)
BIT 0 - Received Data

$DD02: DATA DIRECTION REGISTER A 56578

This register controls the direction of flow of data over port A. By setting a bit of this register to 1, the corresponding bit of the data port is defined as an output. The default value of this register is #FF, ie all outputs.

$DD03: DATA DIRECTION REGISTER B 56579

This register controls the direction of flow of data over port B. By setting a bit of this register to 1, the corresponding bit of the data port is defined as an output. The default value of this register is #00, ie all inputs.

$DD04: TIMER A LOW BYTE 56580

This register along with $DD05 form a 16 bit timer. The action of this timer is described in detail on page 524. Timer A is used by the operating system to time RS-232 transmit and receive operations.

$DD05: TIMER A HIGH BYTE 56581

This is the high byte of the 16 bit timer A.

$DD06: TIMER B LOW BYTE 56582

This register along with $DD07 form a 16 bit timer. The action of this timer is described in detail on page 524. Timer B is used by the operating system to time RS-232 transmit and receive operations. It is also used for all serial bus timing.

$DD07: TIMER B HIGH BYTE 56583

This is the high byte of the 16 bit timer B.

$DD08: TOD CLOCK 10THS OF SECOND 56584

This is the first of four registers that comprise the Time Of Day (TOD) clock. This is a timer, organised into registers of hours, minutes, seconds and 10ths of seconds. Each register counts upwards in BCD format. This enables easy conversion into ASCII digits.

$DD09: TOD CLOCK SECONDS REGISTER 56585

This register indicates seconds in the Time of Day (TOD) clock in BCD format. Bit 7 of this register is not used.

$DD0A: TOD CLOCK MINUTES REGISTER 56586

This register indicates minutes in the Time of Day (TOD) clock in BCD format. Bit 7 of this register is not used.

$DD0B: TOD CLOCK HOURS AND AM/PM 56587

This register indicates hours in the Time of Day (TOD) clock in 12 hour BCD format. In addition, bit 7 is used to indicate AM or PM (0=AM, 1=PM). Bits 5 and 6 of this register are not used.

$DD0C: SERIAL DATA REGISTER 56588

This register forms a synchronous serial I/O port. The direction of data flow is controlled by bit 6 of $DC0E. In input mode, data is read from the SP pin of the chip whenever a pulse appears on the CNT pin. After 8 bits have been read, the byte can be read from this register. In the output mode, a CNT pulse appears whenever data is available at the SP pin. Both the SP and CNT pins are available on the user port.

The speed of data transfer on the port is determined by timer A, with a maximum speed of the system o2 clock / 4. Data is always transferred starting with the MSB (bit 7). Once a byte has been sent or received, a flag is set in the interrupt control register ($DC0D).

$DD0D: INTERRUPT CONTROL REGISTER 56589

BIT 7: NMI OCCURRED FLAG

If one of the 5 sources of interrupt on the 6526 has caused an interrupt, then this bit is set to 1. This can be used for polling during an interrupt routine to determine the source of the interrupt.

In order for an interrupt bit in this register to cause an NMI at the processor, the NMI output must be set for that bit. Similarly, to prevent it from causing an NMI, that bit must be cleared. Both actions are performed by writing a 1 to that bit. The action taken is determined by writing a value to this bit. Write a value 0, and other bits written will be cleared. Write a value 1, and other bits written will be set.

BITS 6-5: UNUSED

BIT 4: FLAG1

This bit will be set to 1 whenever a signal is received on the CIA#1 FLAG line. This line is used as the cassette read line, and also as the serial bus SRQ input.

Writing a 1 to this bit will enable or disable the FLAG1 NMI, depending on the value written to bit 7.

BIT 3: SERIAL DATA REGISTER

This bit will be set to 1 whenever the chip serial port has finished reading or writing a byte of data.

Writing a 1 to this bit will enable or disable the SDR NMI, depending on the value written to bit 7.

BIT 2: TOD CLOCK ALARM

This bit will be set to 1 whenever the chip TOD clock has reached the time set in the TOD alarm registers.

Writing a 1 to this bit will enable or disable the TOD alarm NMI, depending on the value written to bit 7.

BIT 1: TIMER A

This bit will be set to 1 whenever timer A reaches zero.

Writing a 1 to this bit will enable or disable the timer A NMI depending on the value written to bit 7.

BIT 0: TIMER B

This bit will be set to 1 whenever timer B reaches zero.

Writing a 1 to this bit will enable or disable the timer B NMI depending on the value written to bit 7.

$DD0E: CONTROL REGISTER A 56590

The 8 bits in this register are used to control timer A.

BIT 7: TOD FREQUENCY

This bit determines whether a 50 Hz (1) or 60Hz (0) signal has to be applied to the TOD pin in order for the TOD clock to keep accurate time.

BIT 6: SERIAL PORT MODE

This bit is used to determine whether the on-chip serial port operates in input mode or output mode. A value of 0 indicates input, and 1 indicates output.

BIT 5: TIMER A COUNT MODE

This bit determines the manner in which timer A counts down. A value of 0 here, and the timer counts system o2 clock pulses. A value of 1, and the timer counts pulses on the external CNT line.

BIT 4: FORCE LOAD TIMER A

Writing a 1 to this bit forces the contents of the timer A low and high byte registers to be loaded into the timer itself. Reading this bit has no effect.

BIT 3: TIMER A RUN MODE

Setting this bit to 1 causes timer A to operate in one-shot mode. This means that once the timer has reached zero, it will stop. The original value is loaded back into the timer.

Clearing this bit to 0 causes the timer to operate in continuous mode. This means that once the timer has reached zero, the starting value is re-loaded into the timer, and counting is continued.

BIT 2: TIMER A OUTPUT ON PB6

This bit is only operative when bit 1 is set. Writing a 1 here causes the current value of PB6 to be toggled whenever the timer reaches zero. Writing a 0 here causes a single pulse of approximately 1 microsecond (ie. 1 clock cycle) to appear on the port.

BIT 1: TIMER A OUTPUT ON PB6

Setting this bit to 1 enables the timer output on PB6. This output occurs whenever the timer reaches zero, but its nature is determined by bit 2

BIT 0: START / STOP TIMER A

Setting this bit to 1 will start the timer counting from the value loaded into it down to zero. Clearing this bit to 0 will stop the timer from counting.

$DD0F: CONTROL REGISTER B 56591

The 8 bits in this register are used to control timer B and the TOD clock.

BIT 7: SET ALARM / TOD CLOCK

This bit determines what happens when the TOD clock registers are written to. If this bit is set to 1, then writing to the registers sets the alarm. If this bit is cleared to 0, then writing to the registers sets the TOD clock.

BITS 6-5: TIMER B COUNT MODE

These four bits are used to determine what is counted by timer B. The four modes are as follows:

Count system o2 clock pulses	00 -
Count CNT pulses	01 -
Count timer A underflow pulses	10 -
Count timer A underflows while CNT is high	11 -

BIT 4: FORCE LOAD TIMER B

Writing a 1 to this bit forces the contents of the timer B low and high byte registers to be loaded into the timer itself. Reading this bit has no effect.

BIT 3: TIMER B RUN MODE

Setting this bit to 1 causes timer B to operate in one-shot mode. This means that once the timer has reached zero, it will stop. The original value is loaded back into the timer.

Clearing this bit to 0 causes the timer to operate in continuous mode. This means that once the timer has reached zero, the starting value is re-loaded into the timer, and counting is continued.

BIT 2: TIMER B OUTPUT ON PB7

This bit is only operative when bit 1 is set. Writing a 1 here causes the current value of PB7 to be toggled whenever the timer reaches zero. Writing a 0 here causes a single pulse of approximately 1 microsecond (ie. 1 clock cycle) to appear on the port.

BIT 1: TIMER B OUTPUT ON PB7

Setting this bit to 1 enables the timer output on PB7. This output occurs whenever the timer reaches zero, but its nature is determined by bit 2

BIT 0: START / STOP TIMER B

Setting this bit to 1 will start the timer counting from the value loaded into it down to zero. Clearing this bit to 0 will stop the timer from counting.

Appendix 1
The version 3 KERNAL ROM

Commodore have recently introduced a new version of the KERNAL ROM for the Commodore 64. This ROM, dubbed version 3, fixes several of the known 'bugs' of version 2. The changes in the ROM are as follows:

1. A patch has been added to the RS-232 input routine to initialise the RS-232 parity byte RIPRTY ($AB) on reception of a start bit.

2. The clear screen routine now fills colour RAM with the current character colour instead of the screen background colour. This enables characters to be POKEd into screen memory without having to POKE colour memory in order to make them visible.

3. The screen edit bug which sometimes causes the computer to crash when deleting characters from the bottom line of the screen has now been fixed.

4. The input prompt bug, that caused the prompt string to be included with the input data when text went beyond the end of a screen line has now been fixed.

A disassembly of the new sections of code is given below:

RS-232 FIX

```
., ef94  4c d3 e4 jmp $e4d3    ;jump to new RS-232 patch
., e4d3  85 a9    sta $a9      ;RINONE - check for start bit
., e4d5  a9 01    lda #$01
., e4d7  85 ab    sta $ab      ;RIPRTY - RS-232 input parity
., e4d9  60       rts
```

SCREEN CLEAR FIX

```
., e4da  ad 86 02 lda $0286  ;COLOR - current character colour

., ea07  20 da e4 jsr $e4da    ;reset character colour
., ea0a  a9 20    lda #$20     ;ASCII space
., ea0c  91 d1    sta ($d1),y  ;store character on screen
., ea0e  88       dey
., ea0f  10 f6    bpl $ea07
., ea11  60       rts
., ea12  ea       nop
```

INPUT / DELETE FIX

```
., e621  20 92 e5 jsr $e592    ;check cursor position
., e57c  20 f0 e9 jsr $e9f0    ;set start of line
., e57f  a9 27    lda #$27
., e581  e8       inx
., e582  b4 d9    ldy $d9,x    ;LDTB1 - screen line link
                                table
., e584  30 06    bmi $e58c
., e586  18       clc
., e587  69 28    adc #$28
., e589  e8       inx
., e58a  10 f6    bpl $e582
., e58c  85 d5    sta $d5      ;LNMX - physical screen line
                                length
., e58e  4c 24 ea jmp $ea24    ;synchronise colour pointer
., e591  ea       nop
., e592  e4 c9    cpx $c9      ;LXSP - cursor at start of
                                INPUT
., e594  f0 03    beq $e599
., e596  4c ed e6 jmp $e6ed    ;retreat cursor
., e599  60       rts
., e59a  ea       nop
```

CHECKSUM CORRECTION

```
.: e4ac  81
```

KERNAL VERSION ID

```
.: ff80  03
```

Appendix 2
HEX/Decimal converter

LOW BYTE

LOW BIT

		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
H	4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
I	5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
G	6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
H	7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
	8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
B	9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
I	A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
T	B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
	C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
	D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
	E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
	F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

LOW BIT

		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0	256	512	768	1024	1280	1536	1792	2048	2304	2560	2816	3072	3328	3584	3840
	1	4096	4352	4608	4864	5120	5376	5632	5888	6144	6400	6656	6912	7168	7424	7680	7936
	2	8192	8448	8704	8960	9216	9472	9728	9984	10240	10496	10752	11008	11264	11520	11776	12032
	3	12288	12544	12800	13056	13312	13568	13824	14080	14336	14592	14848	15104	15360	15616	15872	16128
H	4	16384	16640	16896	17152	17408	17664	17920	18176	18432	18688	18944	19200	19456	19712	19968	20224
I	5	20480	20736	20992	21248	21504	21760	22016	22272	22528	22784	23040	23296	23552	23808	24064	24320
G	6	24576	24832	25088	25344	25600	25856	26112	26368	26624	26880	27136	27392	27648	27904	28160	28416
H	7	28672	28928	29184	29440	29696	29952	30208	30464	30720	30976	31232	31488	31744	32000	32256	32512
	8	32768	33024	33280	33536	33792	34048	34304	34560	34816	35072	35328	35584	35840	36096	36352	36608
B	9	36864	37120	37376	37632	37888	38144	38400	38656	38912	39168	39424	39680	39936	40192	40448	40704
I	A	40960	41216	41472	41728	41984	42240	42496	42752	43008	43264	43520	43776	44032	44288	44544	44800
T	B	45056	45312	45568	45824	46080	46336	46592	46848	47104	47360	47616	47872	48128	48384	48640	48896
	C	49152	49408	49664	49920	50176	50432	50688	50944	51200	51456	51712	51968	52224	52480	52736	52992
	D	53248	53504	53760	54016	54272	54528	54784	55040	55296	55552	55808	56064	56320	56576	56832	57088
	E	57344	57600	57856	58112	58368	58624	58880	59136	59392	59648	59904	60160	60416	60672	60928	61184
	F	61440	61696	61952	62208	62464	62720	62976	63232	63488	63744	64000	64256	64512	64768	65024	65280

Appendix 3
CBM ASCII codes

Code	Character
00	
01	<SHIFT>
02	<CBM>
03	<STOP>
04	<CTRL>
05	<WHITE>
06	
07	
08	<sC= OFF>
09	<sC= ON>
0A	
0B	
0C	
0D	<RETURN>
0E	<LOWER>
0F	
10	
11	<DOWN>
12	<RVS ON>
13	<HOME>
14	<DEL>
15	
16	
17	
18	
19	
1A	
1B	
1C	<RED>
1D	<RIGHT>
1E	<GREEN>
1F	<BLUE>
20	<SPACE>
21	!
22	"
23	#
24	$
25	%
26	&
27	'
28	(
29	)
2A	*
2B	+
2C	,
2D	–
2E	.
2F	/
30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9
3A	:
3B	;
3C	<
3D	=
3E	>
3F	?
40	@
41	a
42	b
43	c
44	d
45	e
46	f
47	g
48	h
49	i
4A	j
4B	k
4C	l
4D	m
4E	n
4F	o
50	p
51	q
52	r
53	s
54	t
55	u
56	v
57	w
58	x
59	y
5A	z
5B	[
5C	£
5D	]
5E	↑
5F	←
60	@
61	A
62	B
63	C
64	D
65	E
66	F
67	G
68	H
69	I
6A	J
6B	K
6C	L
6D	M
6E	N
6F	O
70	P
71	Q
72	R
73	S
74	T
75	U
76	V
77	W
78	X
79	Y
7A	Z
7B	

7C		A0	<s SPACE>	C4	E8
7D		A1		C5	E9
7E		A2		C6	EA
7F		A3		C7	EB
80		A4		C8	EC
81	<ORANGE>	A5		C9	ED
82		A6		CA	EE
83		A7		CB	EF
84		A8		CC	F0
85	<f1>	A9		CD	F1
86	<f3>	AA		CE	F2
87	<f5>	AB		CF	F3
88	<f7>	AC		D0	F4
89	<f2>	AD		D1	F5
8A	<f4>	AE		D2	F6
8B	<f6>	AF		D3	F7
8C	<f8>	B0		D4	F8
8D	<sh RET>	B1		D5	F9
8E	<UPPER>	B2		D6	FA
8F		B3		D7	FB
90	<BLACK>	B4		D8	FC
91	<UP>	B5		D9	FD
92	<RVS OFF>	B6		DA	FE
93	<CLR>	B7		DB	FF
94	<INST>	B8		DC	
95	<BROWN>	B9		DD	
96	<PINK>	BA		DE	
97	<GREY 1>	BB		DF	
98	<GREY 2>	BC		E0	
99	<L GREEN>	BD		E1	
9A	<L BLUE>	BE		E2	
9B	<GREY 3>	BF		E3	
9C	<PURPLE>	C0		E4	
9D	<LEFT>	C1		E5	
9E	(YELLOW>	C2		E6	
9F	<CYAN>	C3		E7	

INDEX

6510 onboard i/o port 352
6526 CIA chip 381,388
6566/7 VIC II chip 357
6581 SID chip 368
A/D Converters 380
ABS 15,185
ACPTR 250,319,322
Add digit to fac#1 111
Advance cursor 223
AND 13,134
ASC 14,166
Assign floating point 110
  integer 110
  string 110
ATN 16,209

?Bad subscript error 144
BASIC command vectors 80
  input buffer 46
  keyword table 81
  program space 76
  warm restart 210
  warm start 100

Cassette buffer 75
Cassette port 342
Character ROM image 77
Check for 8-ROM 306
CHKIN 10,265,319,322
CHKOUT 19,266,319,323
CHR$ 15,162
CHRGET subroutine 40,79,211
CHRGOT subroutine 40,79
CHRIN 10,262,319,323
CHROUT 10,264,319,324
CIA reg description 353
  timers 524
CINT 319,325
CIOUT 249,319,324
CLALL 10,319,325
Clear screen 217
CLOSE 10,16,204,267,319,326
CLR 12,95
CLRCHN 9,270,319,326
CMD 13,113
Compare fac#1 with mem 185
Concatenate two strings 159
Confirm char in (A) 130
  program mode 150
  result 124
CONT 12,103
Convert ASCII string to flpt 131
  fac#1 to integer 246,274
  fac#1 to string 149
  TI to string 192
Copy fac#1 into fac#2 183
  fac#2 into fac#1 183
COS 16,207
Current BASIC line number 34
  BASIC variable name 34
  character colour code 61
  DATA line number 37
  device number 42
  filename length 47
  I/O channel 39
  logical file number 48
  secondary address 48

DATA 12,106
DATA type flag 30
DEF 14,150
Default output device 43
DIM 14,136
Divide flpt 179
END 12,103
Error message table 82
  routine 86
Evaluate exp. in text 124
  single term 128
  text to 1 byte 142
EXP 15,196

FAC#1 39
FAC#2 40
Find any tape header 285
  specific tape header 288
Flag insert mode 52
  repeat keypress 62
FN 14,151
FOR 12,99
FRE 14,149

Garbage collection 156
GET 16,117
  from RS-232 260
GETIN 10,262,319,327
GOSUB 12,37
GOTO 4,105

Home cursor 217

Identify variable 136
IF 13,107
INPUT 13,37,118
  error messages 122
  from keyboard 219
INPUT# 13,117
INT 14,187
IOBASE 215,319,327
IOINIT 309,319,328
IRQ entry point 236,315

Kernal Jump table 316
Keyboard buffer queue 50

LEFT$ 14,163
LEN 14,165
LET 11,109
LIST 13,94
LISTEN 319,328
LOAD 10,16,203,275,319,328
  FAC#1 from memory 181
  FAC#2 from memory 177
  from serial bus 275
  from tape 277
LOAD/VERIFY flag 42,37
LOG 15,175
Log CIA key reading 282

MEMBOT 311,319,329
MEMTOP 311,319,330
MID$ 14,164
Multiply fac#1 by 10 179

Negate fac#1 196
NEW 12,95
NEXT 13,123
NOT 13,
ON 13,108
OPEN 10,16,204,270,319,330
  RS-232 273
OR 13,134
?Out of memory error 86
Output to screen 225
?Overflow error 173

Page 4,8,9,10,53,57
PAL/NTSC flag 65

PEEK 14,168
Perform basic keyword 102
  string housekeeping 161
PLOT 216,319,331
Pointer - bottom of basic mem 60
  bottom of strings 35
  current filename 47
  end of arrays +1 35
  highest basic address 36
  start of arrays 35
  start of basic text 34
  start of variables 34
  top of basic 60
POKE 15,169
POS 14,149
POWER (^) 15,195
Power reset entry point 305
Previous basic line number 36
PRINT 13,113
Print power up message 213
PRINT# 13,112
RAMTAS 307,319,331
RDTIM 283,319,332
READ 113,118
READST 310,319,332
REM 13,107
Repeat speed counter 62
Restart BASIC 88
RESTOR 307,319,333
RESTORE 12,102
Retreat cursor 224
RETURN 12,105
RIGHT$ 14,164
RND 16,199
RS-232 port 346
  register images 9,64
  timing table 215,313
RUN 12,104

SAVE 10,16,202,279,319,333
  to serial bus 280
  to tape 281
SCNKEY 237,319,334
SCREEN 319,334
Screen line link table 53
Scroll screen 231
Search for line number 94
  for variable 131
SECOND 248,319,334

Send tone to tape 349
Serial bus 260
Series evaluation 197
Set colour code 231
  I/O defaults 218
SETLFS 310,319,335
SETMSG 311,320,336
SETNAM 310,320,336
SETTIM 320,337
SETTMO 311,320,337
SGN 15,184
Shift-run equivalent 245
SIN 16,207
Smooth scroll horiz 362
Sprite collision registers 364
  colour registers 366
  enable register 360
  position registers 358
SQR 15,195
Stack 8,83
Start of current variable 37
STATUS - I/O status word 41
STOP 10,12,103,329,338
Store fac#1 in memory 182
  string in high ram 160
  tape characters in ram 297
STR$ 14,153
SYS 9,16,202
System hardware vectors 317

TAB(13
Table of active file numbers 58
  device numbers 59
  secondary addresses 59
TALK 245,320,338
TAN 26,208
Tape error log 44
Temporary fac for multiply 33
  string stack 33
TKSA 248,320,338
Tokenise input buffer 92
Top of screen memory 62

UDTIM 282,320,339
UNLSN 320,339
UNTLK 249,320
User port 351

VAL 14,166
VECTOR 320,340
  BASIC char dispatch 69
  BASIC warm start 69
  evaluate basic token 69
  hardware IRQ interrupt 70
  hardware NMI 71
  KERNAL CHKIN 72
  KERNAL CHKOUT 72
  KERNAL CHRIN 72
  KERNAL CHROUT 73
  KERNAL CLALL 73
  KERNAL CLOSE 72
  KERNAL CLRCHN 72
  KERNAL GETIN 73
  KERNAL LOAD 73
  KERNAL OPEN 72
  KERNAL SAVE 74
  KERNAL STOP 73
  keyboard decode table 53
  keyboard table setup 64
  list basic text 69
  print basic error message 69
  software BRK 71
  tokenise basic text 69
VERIFY 16,203
VIC control register 363
Voice 368,371,378

WAIT 15,169
Warm start BASIC 312
Write data to tape 301
  tape header 286

Commodore 64 Whole Memory Guide is much more than a memory map. Instead of just giving memory locations it gives you a detailed description of each location, explaining what it's for, how it is used by the computer, and, more importantly, how it can be used by the programmer.

The memory guide to the Commodore 64 is split up into three main sections; the RAM guide, the I/O guide, and the ROM guide. The ROM guide also includes a complete and annotated disassembly of the Commodore 64 ROMs.

If you are a machine code programmer, this book will be invaluable in helping you to write programs that incorporate the subroutines contained within the Commodore 64 ROM. It explains how to pass any parameters that may be required to the routines, and how to recover the results of calculations etc. that are returned by these routines. Also covered is the procedure used by the computer to cope with errors in the data operated on by the routines.

If you are a BASIC Programmer, this book will enable you to manipulate the system variables used by the Commodore 64 to your own ends. This will allow you to use the advanced features of the Commodore 64 to their full.

All in all, Commodore 64 Whole Memory Guide is a book for *everybody* wishing to utilise their Commodore 64 to its maximum.

Melbourne House Publishers

ISBN 0-86161-194-2