

Commander X16 Programmer's Reference Guide

Michael Steil, mist64@mac.com

This is the PRELIMINARY Programmer's Reference Guide for the Commander X16 computer. Every and any information in this document can change, as the product is still in development!

Table of contents

[Chapter 1: Overview](#)

[Chapter 2: Editor](#)

[Chapter 3: BASIC](#)

[Chapter 4: KERNAL](#)

[Chapter 5: Math Library](#)

[Chapter 6: Machine Language Monitor](#)

[Chapter 7: Memory Map](#)

[Chapter 8: Video Programming](#)

[Chapter 9: Sound Programming](#)

[Chapter 10: I/O Programming](#)

[Chapter 11: Working with CMDR-DOS](#)

[Chapter 12: Hardware Pinouts](#)

[Chapter 13: Upgrade Guide](#)

[Appendix A: Sound](#)

Chapter 1: Overview

The Commander X16 is a modern home computer in the philosophy of Commodore computers like the VIC-20 and the C64.

Features:

- 8-bit 65C02 CPU at 8 MHz
- 512 KB or 2 MB RAM banked RAM
- 512 KB ROM
- Expansion Cards (Gen 1) & Cartridges (Gen 1 and Gen 2)
 - Up to 3.5MB of RAM/ROM
 - 5 32-byte Memory-Mapped IO slots
- VERA video controller
 - Up to 640x480 resolution
 - 256 colors from a palette of 4096
 - 128 sprites
 - VGA, NTSC and RGB output
- three sound generators
 - Yamaha YM2151: 8 channels, FM synthesis
 - VERA PSG: 16 channels, 4 waveforms
 - VERA PCM: 48 kHz, 16 bit, stereo
- Connectivity:
 - PS/2 keyboard and mouse
 - 4 NES/SNES controllers
 - SD card
 - Commodore Serial Bus ("IEC")
 - Many Free GPIOs ("user port")

As a modern sibling of the line of Commodore home computers, the Commander X16 is reasonably compatible with computers of that line.

- Pure BASIC programs are fully backwards compatible with the VIC-20 and the C64.
- POKes for video and audio are not compatible with any Commodore computer. (There are no VIC or SID chips, for example.)
- Pure machine language programs (\$FF81+ KERNAL API) are compatible with Commodore computers.

Chapter 2: Editor

The X16 has a built-in screen editor that is backwards-compatible with the C64, but has many new features.

Modes

The editor's default mode is 80x60 text mode. The following text mode resolutions are supported:

Mode	Description
\$00	80x60 text
\$01	80x30 text
\$02	40x60 text
\$03	40x30 text
\$04	40x15 text
\$05	20x30 text
\$06	20x15 text
\$07	22x23 text
\$08	64x50 text
\$09	64x25 text
\$0A	32x50 text
\$0B	32x25 text
\$80	320x240@256c 40x30 text

Mode \$80 contains two layers: a text layer on top of a graphics screen. In this mode, text color 0 is translucent instead of black.


To switch modes, use the BASIC statement `SCREEN` or the KERNAL API `screen_mode`. In the BASIC editor, the F4 key toggles between modes 0 (80x60) and 3 (40x30).

ISO Mode

In addition to PETSCII, the X16 also supports the ISO-8859-15 character encoding. In ISO-8859-15 mode ("ISO mode"):

- The character set is switched from Commodore-style (with PETSCII drawing characters) to a new ASCII/ISO-8859-15 compatible set, which covers most Western European writing systems.
- The encoding (`CHR$()` in BASIC and `BSOUT` in machine language) now complies with ASCII and ISO-8859-15.
- The keyboard driver will return ASCII/ISO-8859-15 codes.

This is the encoding:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x																
1x																
2x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
8x																
9x																
Ax		ı	¢	£	€	¥	Š	§	š	©	ª	«	¬		®	ˆ
Bx	°	±	²	³	Ž	μ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	ı
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

- The non-printable areas \$00-\$1F and \$80-\$9F in the character set are filled with inverted variants of the codes \$40-\$5F and \$60-\$7F, respectively.
- The code \$AD is a non-printable soft hyphen in ISO-8859-15. The ROM character set contains the Commander X16 logo at this location.

ISO mode can be enabled and disabled using two new control codes:

- `CHR$($0F)` : enable ISO mode
- `CHR$($8F)` : enable PETSCII mode (default)

You can also enable ISO mode in direct mode by pressing `Ctrl+ 0` .

Important: In ISO mode, BASIC keywords need to be written in upper case, that is, they have to be entered with the Shift key down, and abbreviating keywords is no longer possible.

Background Color

In regular BASIC text mode, the video controller supports 16 foreground colors and 16 background colors for each character on the screen.

The new "swap fg/bg color" code is useful to change the background color of the cursor, like this:

```
PRINT CHR$(1); : REM SWAP FG/BG
PRINT CHR$(1C); : REM SET FG COLOR TO RED
PRINT CHR$(1); : REM SWAP FG/BG
```

The new BASIC instruction `COLOR` makes this easier, but the trick above can also be used from machine code programs.

To set the background color of the complete screen, it just has to be cleared after setting the color:

```
PRINT CHR$(147);
```

Scrolling

The C64 editor could only scroll the screen up (when overflowing the last line or printing or entering DOWN on the last line). The X16 editor scrolls both ways: When the cursor is on the first line and UP is printed or entered, the screen contents scroll down by a line.

New Control Characters

This is the set of all supported PETSCII control characters. Entries in bold indicate new codes compared to the C64:

If there are two meanings listed, the first indicates input (a keypress) and the second indicates output.

Code			Code
\$00	NULL	VERBATIM MODE	\$80
\$01	SWAP COLORS	COLOR: ORANGE	\$81
\$02	PAGE DOWN	PAGE UP	\$82
\$03	STOP	RUN	\$83
\$04	END	HELP	\$84
\$05	COLOR: WHITE	F1	\$85
\$06	MENU	F3	\$86
\$07	BELL	F5	\$87
\$08	DISALLOW CHARSET SW (SHIFT+ALT)	F7	\$88
\$09	TAB / ALLOW CHARSET SW	F2	\$89
\$0A	LF	F4	\$8A
\$0B	-	F6	\$8B
\$0C	-	F8	\$8C
\$0D	RETURN	SHIFTED RETURN	\$8D

\$0E	CHARSET: LOWER/UPPER	CHARSET: UPPER/PETSCII	\$8E
\$0F	CHARSET: ISO ON	CHARSET: ISO OFF	\$8F
\$10	F9	COLOR: BLACK	\$90
\$11	CURSOR: DOWN	CURSOR: UP	\$91
\$12	REVERSE ON	REVERSE OFF	\$92
\$13	HOME	CLEAR	\$93
\$14	DEL (PS/2 BACKSPACE)	INSERT	\$94
\$15	F10	COLOR: BROWN	\$95
\$16	F11	COLOR: LIGHT RED	\$96
\$17	F12	COLOR: DARK GRAY	\$97
\$18	SHIFT+TAB	COLOR: MIDDLE GRAY	\$98
\$19	FWD DEL (PS/2 DEL)	COLOR: LIGHT GREEN	\$99
\$1A	-	COLOR: LIGHT BLUE	\$9A
\$1B	ESC	COLOR: LIGHT GRAY	\$9B
\$1C	COLOR: RED	COLOR: PURPLE	\$9C
\$1D	CURSOR: RIGHT	CURSOR: LEFT	\$9D
\$1E	COLOR: GREEN	COLOR: YELLOW	\$9E
\$1F	COLOR: BLUE	COLOR: CYAN	\$9F

Notes:

- \$01: SWAP COLORS swaps the foreground and background colors in text mode
- \$07/\$09/\$0A/\$18/\$1B: have been added for ASCII compatibility. [*\$0A/\$18/\$1B do not have any effect on output. Outputs of \$08/\$09 have their traditional C64 effect*]
- \$80: VERBATIM MODE prints the next character (only!) as a glyph without interpretation. This is similar to quote mode, but also includes codes CR (\$0D) and DEL (\$14).
- F9-F12: these codes match the C65 additions
- \$84: This code is generated when pressing SHIFT+END.
- Additionally, the codes \$04/\$06/\$0B/\$0C are interpreted when printing in graphics mode using `GRAPH_put_char`.

Keyboard Layouts

The editor supports multiple keyboard layouts.

Default Layout

On boot, the US layout (`ABC/X16`) is active:

- In PETSCII mode, it matches the US layout where possible, and can reach all PETSCII symbols.
- In ISO mode, it matches the Macintosh US keyboard and can reach all ISO-8859-15 characters. Some characters are reachable through key combinations:

Key	Result
Alt+1	ı
Alt+3	£
Alt+4	¢
Alt+5	§
Alt+7	¶
Alt+9	ª
Alt+0	º
Alt+q	œ
Alt+r	®
Alt+t	þ
Alt+y	¥
Alt+o	ø
Alt+\	«
Alt+s	ß
Alt+d	ð
Alt+g	©
Alt+l	¬
Alt+'	æ
Alt+m	μ
Alt+/'	÷
Shift+Alt+2	€
Shift+Alt+8	°
Shift+Alt+9	·
Shift+Alt+-	X16 logo

Shift+Alt+=	±
Shift+Alt+q	Œ
Shift+Alt+t	þ
Shift+Alt+\	»
Shift+Alt+a	¹
Shift+Alt+d	Ð
Shift+Alt+k	X16 logo
Shift+Alt+'	/Æ
Shift+Alt+c	³
Shift+Alt+b	²
Shift+Alt+/,	¿

(The X16 logo is code point \xad, SHY, soft-hyphen.)

The following combinations are dead keys:

- Alt+ `
- Alt+ 6
- Alt+ e
- Alt+ u
- Alt+ p
- Alt+ a
- Alt+ k
- Alt+ ;
- Alt+ x
- Alt+ c
- Alt+ v
- Alt+ n
- Alt+ ,
- Alt+ .
- Shift+Alt+ S

They generate additional characters when combined with a second keypress:

First Key	Second Key	Result
Alt+ `	a	à
Alt+ `	e	è
Alt+ `	i	ì
Alt+ `	o	ò
Alt+ `	u	ù
Alt+ `	A	À
Alt+ `	E	È

Alt+`	I	Ì
Alt+`	o	Ò
Alt+`	u	Ù
Alt+`	u	`
Alt+6	e	ê
Alt+6	u	û
Alt+6	i	î
Alt+6	o	ô
Alt+6	a	â
Alt+6	E	Ê
Alt+6	u	Û
Alt+6	I	Î
Alt+6	o	Ô
Alt+6	A	Â
Alt+e	e	é
Alt+e	y	ý
Alt+e	u	ú
Alt+e	i	í
Alt+e	o	ó
Alt+e	a	á
Alt+e	E	É
Alt+e	Y	Ý
Alt+e	U	Ú
Alt+e	I	Í
Alt+e	o	Ó
Alt+e	A	Á
Alt+u	e	ë
Alt+u	y	ÿ
Alt+u	u	ü
Alt+u	i	ï
Alt+u	o	ö
Alt+u	a	ä

Alt+u	E	Ě
Alt+u	Y	Ÿ
Alt+u	U	Ü
Alt+u	I	İ
Alt+u	O	Ö
Alt+u	A	Ä
Alt+p	␣	,
Alt+a	␣	-
Alt+k	a	å
Alt+k	A	Å
Alt+x	␣	.
Alt+c	c	ç
Alt+c	C	Ç
Alt+v	s	š
Alt+v	z	ž
Alt+v	S	Š
Alt+v	Z	Ž
Alt+n	o	õ
Alt+n	a	ã
Alt+n	n	ñ
Alt+n	O	Õ
Alt+n	A	Ã
Alt+n	N	Ñ
Shift+Alt+s	␣	\xa0
Shift+Alt+;	=	×

"␣" denotes the space bar.

ROM Keyboard Layouts

The following keyboard layouts are available from ROM. You can select one directly with the BASIC `KEYMAP` command, e.g. `KEYMAP"ABC/X16"`, or via the X16 Control Panel with the BASIC `MENU` command.

Identifier	Description	Code
ABC/X16	ABC - Extended (X16)	-
EN-US/INT	United States - International	00020409

EN-GB	United Kingdom	00000809
SV-SE	Swedish	0000041D
DE-DE	German	00000407
DA-DK	Danish	00000406
IT-IT	Italian	00000410
PL-PL	Polish (Programmers)	00000415
NB-NO	Norwegian	00000414
HU-HU	Hungarian	0000040E
ES-ES	Spanish	0000040A
FI-FI	Finnish	0000040B
PT-BR	Portuguese (Brazil ABNT)	00000416
CS-CZ	Czech	00000405
JA-JP	Japanese	00000411
FR-FR	French	0000040C
DE-CH	Swiss German	00000807
EN-US/DV0	United States - Dvorak	00010409
ET-EE	Estonian	00000425
FR-BE	Belgian French	0000080C
EN-CA	Canadian French	00001009
IS-IS	Icelandic	0000040F
PT-PT	Portuguese	00000816
HR-HR	Croatian	0000041A
SK-SK	Slovak	0000041B
SL-SI	Slovenian	00000424
LV-LV	Latvian	00000426
LT-LT	Lithuanian IBM	00000427

All remaining keyboards are based on the respective Windows layouts. `EN-US/INT` differs from `EN-US` only in Alt/AltGr combinations and some dead keys.

The BASIC command `KEYMAP` allows activating a specific keyboard layout. It can be added to the auto-boot file, e.g.:

```
10 KEYMAP"NB-NO"
SAVE"AUTOBOOT.X16"
```

Loadable Keyboard Layouts

The tables for the active keyboard layout reside in banked RAM, at \$A000 on bank 0:

Addresses	Description
\$A000-\$A07F	Table 0
\$A080-\$A0FF	Table 1
\$A100-\$A17F	Table 2
\$A180-\$A1FF	Table 3
\$A200-\$A27F	Table 4
\$A280-\$A07F	Table 5
\$A300-\$A37F	Table 6
\$A380-\$A3FF	Table 7
\$A400-\$A47F	Table 8
\$A480-\$A4FF	Table 9
\$A500-\$A57F	Table 10
\$A580-\$A58F	big-endian bitfield: keynum codes for which Caps means Shift
\$A590-\$A66F	dead key table
\$A670-\$A67E	ASCIIZ identifier (e.g. "ABC/X16")

The first byte of each of the 11 tables is the table ID which contains the encoding and the combination of modifiers that this table is for.

Bit	Description
7	0: PETSCII, 1: ISO
6-3	always 0
2	Ctrl
1	Alt
0	Shift

- AltGr is represented by Ctrl+Alt.
- ID \$C6 represents Alt *or* AltGr (ISO only)
- ID \$C7 represents Shift+Alt *or* Shift+AltGr (ISO only)
- Empty tables have an ID of \$FF.

The identifier is followed by 127 output codes for the keynum inputs 1-127.

- Dead keys (i.e. keys that don't generate anything by themselves but modify the next key) have a code of 0 and are further described in the dead key table (ISO only)
- Keys that produce nothing have an entry of 0. (They can be distinguished from dead keys as they don't have an entry in the dead key table.)

The dead key table has one section for every dead key with the following layout:

Byte	Description
0	dead key ID (PETSCII/ISO and Shift/Alt/Ctrl)
1	dead key scancode
2	full length of this table in bytes
3	first additional key ISO code
4	first effective key ISO code
5	second additional key ISO code
6	second effective key ISO code
...	...
n-1	terminator 0xFF

Custom layouts can be loaded from disk like this:

```
BLOAD"KEYMAP",8,0,$A000
```

Here is an example that activates a layout derived from "ABC/X16", with unshifted Y and Z swapped in PETSCII mode:

```
100 KEYMAP"ABC/X16" :REM START WITH DEFAULT LAYOUT
110 BANK 0 :REM ACTIVATE RAM BANK 0
120 FORI=0TO11:B=$A000+128*I:IFPEEK(B)<>0THENNEXT :REM SEARCH FOR TABLE $00
130 POKEB+$2E,ASC("Y") :REM SET KEYNUM $2E ('Z') to 'Y'
140 POKEB+$16,ASC("Z") :REM SET KEYNUM $16 ('Y') to 'Z'
170 REM
180 REM *** DOING THE SAME FOR SHIFTED CHARACTERS
190 REM *** IS LEFT AS AN EXERCISE TO THE READER
```

Custom BASIN PETSCII code override handler

Note: This is a new feature in R44

Some use cases of the BASIN (CHRIN) API call may benefit from being able to modify its behavior, such as intercepting or redirecting certain PETSCII codes. The Machine Language Monitor uses this mechanism to implement custom behavior for F-keys and for loading additional disassembly or memory display when scrolling the screen.

To set up a custom handler, one must configure it before each call to BASIN.

The key handler vector is in RAM bank 0 at addresses \$ac03-\$ac05. The first two bytes are the call address, and the next byte is the RAM or ROM bank. If your callback routine is in low ram, specifying the bank in \$ac05 is not necessary.

The editor will call your callback for every keystroke received and pass the PETSCII code in the A register with carry set. If your handler does not want to override, simply return with carry set.

If you do wish to override, return with carry clear. The editor will then unblink the cursor and call your callback a second time with carry clear *for the same PETSCII code*. This is your opportunity to override. Before returning, you are free to update the screen or perform other KERNAL API calls (with the exception of BASIN). At the end of your routine, set A to the PETSCII code you wish the editor to process. If you wish to suppress the input keystroke, set A to 0.

```

ram_bank = $00
edkeyvec = $ac03
edkeybk  = $ac05

BASIN    = $ffcfcf
BSOUT    = $fffd2

.segment "ONCE"
.segment "STARTUP"
    jmp start
.segment "CODE"

keyhandler:
    bcs @check
    cmp #$54 ; 'T'
    bne :+
    lda #$57 ; 'W'
    rts
:   lda #$54 ; 'T'
    rts
@check:
    cmp #$54 ; 'T'
    beq @will_override
    cmp #$57 ; 'W'
    beq @will_override
    sec
    rts
@will_override:
    clc
    rts

enable_basin_callback:
    lda ram_bank
    pha
    stz ram_bank ; RAM bank 0 contains the handler vector
    php
    sei
    lda #<keyhandler
    sta edkeyvec
    lda #>keyhandler
    sta edkeyvec+1
    ; setting the bank is optional and unnecessary
    ; if the handler is in low RAM.
    ; lda #0
    ; sta edkeybk
    plp
    pla
    sta ram_bank
    rts

start:
    jsr enable_basin_callback
    ; T and W are swapped
@1: jsr BASIN
    cmp #13

```

```

    bne @1
    jsr BSOUT
    ; normal BASIN
@2: jsr BASIN
    cmp #13
    bne @2

    rts

```

Custom Keyboard Keynum Code Handler

Note: This is new behavior for R43, differing from previous releases.

If you need more control over the translation of keynum codes into PETSCII/ISO codes, or if you need to intercept any key down or up event, you can hook the custom scancode handler vector at \$032E/\$032F.

On all key down and key up events, the keyboard driver calls this vector with

- .A: keycode, where bit 7 (most-significant) is clear on key down, and set on key up.

The keynum codes are enumerated [here](#), and their names, similar to that of PS/2 codes, are based on their function in the US layout.

The handler needs to return a key event the same way in .A

- To remove a keypress so that it is not added to the keyboard queue, return .A = 0.
- To manually add a key to the keyboard queue, use the `kdbbuf_put` KERNAL API.

You can even write a completely custom keyboard translation layer:

- Place the code at \$A000-\$A58F in RAM bank 0. This is safe, since the tables won't be used in this case, and the active RAM bank will be set to 0 before entry to the handler.
- Fill the locale at \$A590.
- For every keynum that should produce a PETSCII/ISO code, use `kdbbuf_put` to store it in the keyboard buffer.
- Always set .A = 0 before return from the custom handler.

;EXAMPLE: A custom handler that prints "A" on Alt key down

```

setup:
    sei
    lda #<keyhandler
    sta $032e
    lda #>keyhandler
    sta $032f
    cli
    rts

keyhandler:
    pha

    and #$ff    ;ensure A sets flags
    bmi exit    ;A & 0x80 is key up

    cmp #$3c    ;Left Alt keynum
    bne exit

    lda #'a'

```

```

    jsr $ffd2

exit:
    pla
    rts

```

Function Key Shortcuts

The following Function key macros are pre-defined for your convenience. These shortcuts only work in the screen editor. When a program is running, the F-keys generate the corresponding PETSCII character code.

Key	Function	Comment
F1	LIST:	Lists the current program
F2	SAVE"@:	Press F2 and then type a filename to save your program. The @: instructs DOS to allow overwrite.
F3	LOAD "	Load a file directly, or cursor up over a file listing and press F3 to load a program.
F4	40/80	Toggles between 40 and 80 column screen modes, clearing the screen. Pressing return is required to prevent accidental mode switches.
F5	RUN:	Run the current program.
F6	MONITOR	Opens the Supermon machine language monitor.
F7	DOS"\$<cr>	Displays a directory listing.
F8	DOS"	Issue DOS commands.
F9	-	Not defined. Formerly cycled through keyboard layouts. Instead, use the MENU command to enter the X16 Control Panel, select one, and optionally save the layout as a boot preference.
F10	-	Not defined
F11	-	Not defined
F12	debug	debug features in emulators

Chapter 3: BASIC Programming

Table of BASIC statements and functions

Keyword	Type	Summary	Origin
ABS	function	Returns absolute value of a number	C64
AND	operator	Returns boolean "AND" or bitwise intersection	C64
ASC	function	Returns numeric PETSCII value from string	C64
ATN	function	Returns arctangent of a number	C64
BANK	command	Sets the RAM and ROM banks to use for PEEK, POKE, and SYS	C128
BINS	function	Converts numeric to a binary string	X16
BINPUT#	command	Reads a fixed-length block of data from an open file	X16
BLOAD	command	Loads a headerless binary file from disk to a memory address	X16
BOOT	command	Loads and runs <code>AUTOBOOT.X16</code>	X16
BSAVE	command	Saves a headerless copy of a range of memory to a file	X16
BVERIFY	command	Verifies that a file on disk matches RAM contents	X16
BVLOAD	command	Loads a headerless binary file from disk to VRAM	X16
CHAR	command	Draws a text string in graphics mode	X16
CHR\$	function	Returns PETSCII character from numeric value	C64
CLOSE	command	Closes a logical file number	C64
CLR	command	Clears BASIC variable state	C64
CLS	command	Clears the screen	X16
CMD	command	Redirects output to non-screen device	C64
CONT	command	Resumes execution of a BASIC program	C64
COLOR	command	Sets text fg and bg color	X16
COS	function	Returns cosine of an angle in radians	C64
DA\$	variable	Returns the date in YYYYMMDD format from the system clock	X16
DATA	command	Declares one or more constants	C64
DEF	command	Defines a function for use later in BASIC	C64
DIM	command	Allocates storage for an array	C64
DOS	command	Disk and SD card directory operations	X16
EDIT	command	Open the built-in text editor	X16
END	command	Terminate program execution and return to <code>READY</code> .	C64

EXEC	command	Play back a script from RAM into the BASIC editor	X16
EXP	function	Returns the inverse natural log of a number	C64
FMCHORD	command	Start or stop simultaneous notes on YM2151	X16
FMDRUM	command	Plays a drum sound on YM2151	X16
FMFREQ	command	Plays a frequency in Hz on YM2151	X16
FMINIT	command	Stops sound and reinitializes YM2151	X16
FMINST	command	Loads a patch preset into a YM2151 channel	X16
FMNOTE	command	Plays a musical note on YM2151	X16
FMPAN	command	Sets stereo panning on YM2151	X16
FMPLAY	command	Plays a series of notes on YM2151	X16
FMPOKE	command	Writes a value into a YM2151 register	X16
FMVIB	command	Controls vibrato and tremolo on YM2151	X16
FMVOL	command	Sets channel volume on YM2151	X16
FN	function	Calls a previously defined function	C64
FOR	command	Declares the start of a loop construct	C64
FRAME	command	Draws an unfilled rectangle in graphics mode	X16
FRE	function	Returns the number of unused BASIC bytes free	C64
GET	command	Polls the keyboard cache for a single keystroke	C64
GET#	command	Polls an open logical file for a single character	C64
GOSUB	command	Jumps to a BASIC subroutine	C64
GOTO	command	Branches immediately to a line number	C64
HELP	command	Displays a brief summary of online help resources	X16
HEX\$	function	Converts numeric to a hexadecimal string	X16
I2CPEEK	function	Reads a byte from a device on the I ² C bus	X16
I2CPOKE	command	Writes a byte to a device on the I ² C bus	X16
IF	command	Tests a boolean condition and branches on result	C64
INPUT	command	Reads a line or values from the keyboard	C64
INPUT#	command	Reads lines or values from a logical file	C64
INT	function	Discards the fractional part of a number	C64
JOY	function	Reads gamepad button state	X16
KEYMAP	command	Changes the keyboard layout	X16
LEFT\$	function	Returns a substring starting from the beginning of a string	C64

LEN	function	Returns the length of a string	C64
LET	command	Explicitly declares a variable	C64
LINE	command	Draws a line in graphics mode	X16
LINPUT	command	Reads a line from the keyboard	X16
LINPUT#	command	Reads a line or other delimited data from an open file	X16
LIST	command	Outputs the program listing to the screen	C64
LOAD	command	Loads a program from disk into memory	C64
LOCATE	command	Moves the text cursor to new location	X16
LOG	function	Returns the natural logarithm of a number	C64
MENU	command	Invokes the Commander X16 utility menu	X16
MID\$	function	Returns a substring from the middle of a string	C64
MON	command	Enters the machine language monitor	X16
MOUSE	command	Hides or shows mouse pointer	X16
MOVSPR	command	Set the X/Y position of a sprite	X16
MX/MY/MB	variable	Reads the mouse position and button state	X16
MWHEEL	variable	Reads the mouse wheel movement	X16
NEW	command	Resets the state of BASIC and clears program memory	C64
NEXT	command	Declares the end of a loop construct	C64
NOT	operator	Bitwise or boolean inverse	C64
OLD	command	Undoes a NEW command or warm reset	X16
ON	command	A GOTO/GOSUB table based on a variable value	C64
OPEN	command	Opens a logical file to disk or other device	C64
OR	operator	Bitwise or boolean "OR"	C64
PEEK	function	Returns a value from a memory address	C64
π	function	Returns the constant for the value of pi	C64
POINTER	function	Returns the address of a BASIC variable	C128
POKE	command	Assigns a value to a memory address	C64
POS	function	Returns the column position of the text cursor	C64
POWEROFF	command	Immediately powers down the Commander X16	X16
PRINT	command	Prints data to the screen or other output	C64
PRINT#	command	Prints data to an open logical file	C64
PSET	command	Changes a pixel's color in graphics mode	X16

PSGCHORD	command	Starts or stops simultaneous notes on VERA PSG	X16
PSGFREQ	command	Plays a frequency in Hz on VERA PSG	X16
PSGINIT	command	Stops sound and reinitializes VERA PSG	X16
PSGNOTE	command	Plays a musical note on VERA PSG	X16
PSGPAN	command	Sets stereo panning on VERA PSG	X16
PSGPLAY	command	Plays a series of notes on VERA PSG	X16
PSGVOL	command	Sets voice volume on VERA PSG	X16
PSGWAV	command	Sets waveform on VERA PSG	X16
READ	command	Assigns the next DATA constant to one or more variables	C64
REBOOT	command	Performs a warm reboot of the system	X16
RECT	command	Draws a filled rectangle in graphics mode	X16
REM	command	Declares a comment	C64
REN	command	Renumbers a BASIC program	X16
RESET	command	Performs a hard reset of the system	X16
RESTORE	command	Resets the READ pointer to a DATA constant	C64
RETURN	command	Returns from a subroutine to the statement following a GOSUB	C64
RIGHT\$	function	Returns a substring from the end of a string	C64
RND	function	Returns a floating point number $0 \leq n < 1$	C64
RPT\$	function	Returns a string of repeated characters	X16
RUN	command	Clears the variable state and starts a BASIC program	C64
SAVE	command	Saves a BASIC program from memory to disk	C64
SCREEN	command	Selects a text or graphics mode	X16
SGN	function	Returns the sign of a numeric value	C64
SIN	function	Returns the sine of an angle in radians	C64
SLEEP	command	Introduces a delay in program execution	X16
SPC	function	Returns a string with a set number of spaces	C64
SPRITE	command	Sets attributes for a sprite including visibility	X16
SPRMEM	command	Set the VRAM address for a sprite's visual data	X16
SQR	function	Returns the square root of a numeric value	C64
ST	variable	Returns the status of certain DOS/peripheral operations	C64
STEP	keyword	Used in a FOR declaration to declare the iterator step	C64
STOP	command	Breaks out of a BASIC program	C64

STR\$	function	Converts a numeric value to a string	C64
STRPTR	function	Returns the address of a BASIC string	X16
SYS	command	Transfers control to machine language at a memory address	C64
TAB	function	Returns a string with spaces used for column alignment	C64
TAN	function	Return the tangent for an angle in radians	C64
THEN	keyword	Control structure as part of an IF statement	C64
TI	variable	Returns the jiffy timer value	C64
TI\$	variable	Returns the time HHMMSS from the system clock	C64
TITLE	command	Changes a tile or character on the tile/text layer	X16
TO	keyword	Part of the FOR loop declaration syntax	C64
USR	function	Call a user-defined function in machine language	C64
VAL	function	Parse a string to return a numeric value	C64
VERIFY	command	Verify that a BASIC program was written to disk correctly	C64
VPEEK	function	Returns a value from VERA's VRAM	X16
VPOKE	command	Sets a value in VERA's VRAM	X16
VLOAD	command	Loads a file to VERA's VRAM	X16
WAIT	command	Waits for a memory location to match a condition	C64

Commodore 64 Compatibility

The Commander X16 BASIC interpreter is 100% backwards-compatible with the Commodore 64 one. This includes the following features:

- All statements and functions
- Strings, arrays, integers, floats
- Max. 80 character BASIC lines
- Printing control characters like cursor control and color codes, e.g.:
 - `CHR$(147)` : clear screen
 - `CHR$(5)` : white text
 - `CHR$(18)` : reverse
 - `CHR$(14)` : switch to upper/lowercase font
 - `CHR$(142)` : switch to uppercase/graphics font
- The BASIC vector table (\$0300-\$030B, \$0311/\$0312)
- [SYS](#) arguments in RAM

Because of the differences in hardware, the following functions and statements are incompatible between C64 and X16 BASIC programs.

- `POKE` : write to a memory address
- `PEEK` : read from a memory address
- `WAIT` : wait for memory contents
- `SYS` : execute machine language code (when used with ROM code)

The BASIC interpreter also currently shares all problems of the C64 version, like the slow garbage collector.

Saving Files

By default, you cannot automatically overwrite a file with SAVE, BSAVE, or OPEN. To overwrite a file, you must prefix the filename with @: , like this: SAVE "@:HELLO WORLD" . ("@@:filename" is also acceptable.)

This follows the Commodore convention, which extended to all of their diskette drives and third party hard drives and flash drive readers.

Always confirm you have successfully saved a file by checking the DOS status. When you use the SAVE command from Immediate (or Direct) mode, the system does this for you. In Program mode, you have to do it yourself.

There are two ways to check the error channel from inside a program:

1. You can use the DOS command and make the user perform actions necessary to recover from an error (such as re-saving the file with an @: prefix).
2. You can read the error yourself, using the following BASIC code:

```
10 OPEN 15,8,15
20 INPUT#15,A,B$
30 PRINT A;B$
40 CLOSE 15
```

Refer to [Chapter 11](#) for more details on CMDR-DOS and the command channel.

New Statements and Functions

There are several new statement and functions. Note that all BASIC keywords (such as FOR) get converted into tokens (such as \$81), and the tokens for the new keywords have likely shifted from one ROM version to the next. Therefore, loading BASIC program saved from an old revision of BASIC may mix up keywords. As of ROM version R42, the keyword token positions should no longer shift and programs saved in R42 BASIC should be compatible with future versions.

ASC

TYPE: Integer Function

FORMAT: ASC(<string>)

Action: Returns an integer value representing the PETSCII code for the first character of string . If string is the empty string, ASC() returns 0.

EXAMPLE of ASC Function:

```
?ASC("A")
65

?ASC("")
0
```

BIN\$

TYPE: String Function

FORMAT: BIN\$(n)

Action: Return a string representing the binary value of n. If n <= 255, 8 characters are returned and if 255 < n <= 65535, 16 characters are returned.

EXAMPLE of BIN\$ Function:

```
PRINT BIN$(200) : REM PRINTS 11001000 AS BINARY REPRESENTATION OF 200
PRINT BIN$(45231) : REM PRINTS 1011000010101111 TO REPRESENT 16 BITS
```

BANK

TYPE: Command

FORMAT: BANK m[,n]

Action: Set the active RAM (m) and ROM bank (n) for the purposes of `PEEK` , `POKE` , and `SYS` . Specifying the ROM bank is optional. If it is not specified, its previous value is retained.

EXAMPLE of BANK Statement:

```
BANK 1,10 : REM SETS THE RAM BANK TO 1 AND THE ROM BANK TO 10
?PEEK($A000) : REM PRINTS OUT THE VALUE STORED IN $A000 IN RAM BANK 1
SYS $C063 : REM CALLS ROUTINE AT $C09F IN ROM BANK 10 AUDIO (YM_INIT)
```

Note: In the above example, the `SYS $C063` in ROM bank 10 is a call to [ym_init](#) , which does the first half of what the BASIC command `FMINIT` does, without setting any default instruments. It is generally not recommended to call routines in ROM directly this way, and most BASIC programmers will never have a need to call `SYS` directly, but advanced users may find a good reason to do so.

Note: `BANK` uses its own register to store the the command's desired bank numbers; this will not always be the same as the value stored in `$00` or `$01` . In fact, `$01` is always going to read `4` when `PEEK`ing from BASIC. If you need to know the currently selected RAM and/or ROM banks, you should explicitly set them and use variables to track your selected bank number(s).

Note: Memory address `$00` , which is the hardware RAM bank register, will usually report the bank set by the `BANK` command. The one exception is after a `BLOAD` or `BVERIFY` inside of a running BASIC program. `BLOAD` and `BVERIFY` change the RAM bank (as if you called `BANK`) to the bank that `BLOAD` or `BVERIFY` stopped at.

BINPUT#

TYPE: Command

FORMAT: BINPUT# <n>,<var\$>,<len>

Action: `BINPUT#` Reads a block of data from an open file and stores the data into a string variable. If there are fewer than <len> bytes available to be read from the file, fewer bytes will be stored. If the end of the file is reached, `ST AND 64` will be true.

EXAMPLE of BINPUT# Statement:

```
10 OPEN 8,8,8,"FILE.BIN,S,R"
20 BINPUT#8,A$,10
30 PRINT "I GOT";LEN(A$);"BYTES"
40 IF ST<>0 THEN 20
50 CLOSE 8
```

BOOT

TYPE: Command

FORMAT: BOOT

Action: Load and run a PRG file named `AUTOBOOT.X16` from device 8. If the file is not found, nothing is done and no error is printed.

EXAMPLE of BOOT Statement:

```
BOOT
```

BLOAD**TYPE: Command****FORMAT: BLOAD <filename>, <device>, <bank>, <address>****Action:** Loads a binary file directly into RAM

Note: If the file is loaded to high RAM (starting in the range \$A000-\$BFFF), and the file is larger than what would fit in the current bank, the load will wrap around into subsequent banks.

After a successful load, \$030D and \$030E will contain the address of the final byte loaded + 1. If relevant, the value in memory location \$00 will point to the bank in which the next byte would have been loaded.

EXAMPLES of BLOAD:

```
BLOAD "MYFILE.BIN", 8, 1, $A000:REM LOADS A FILE NAMED MYFILE.BIN FROM DEVICE 8 STARTING IN BANK 1 AT $A000.
BLOAD "WHO.PCX", 8, 10, $B000:REM LOADS A FILE NAMED WHO.PCX INTO RAM STARTING IN BANK 10 AT $B000.
```

BSAVE**TYPE: Command****FORMAT: BSAVE <filename>, <device>, <bank>, <start address>, <end address>****Action:** Saves a region of memory to a binary file.

Note: The save will stop one byte before end address .

This command does not allow for automatic bank advancing, but you can achieve a similar result with successive BSAVE invocations to append additional memory locations to the same file.

EXAMPLES of BSAVE:

```
BSAVE "MYFILE.BIN", 8, 1, $A000, $C000
```

The above example saves a region of memory from \$A000 in bank 1 through and including \$BFFF, stopping before \$C000.

```
BSAVE "MYFILE.BIN, S, A", 8, 2, $A000, $B000
```

The above example appends a region of memory from \$A000 through and including \$AFFF, stopping before \$B000. Running both of the above examples in succession will result in a file MYFILE.BIN 12KiB in size.

BVLOAD**TYPE: Command****FORMAT: BVLOAD <filename>, <device>, <VERA_high_address>, <VERA_low_address>****Action:** Loads a binary file directly into VERA RAM.**EXAMPLES of BVLOAD:**


```
BVLOAD "MYFILE.BIN", 8, 0, $4000 :REM LOADS MYFILE.BIN FROM DEVICE 8 TO VRAM $4000.
BVLOAD "MYFONT.BIN", 8, 1, $F000 :REM LOAD A FONT INTO THE DEFAULT FONT LOCATION ($1F000).
```

CHAR

TYPE: Command

FORMAT: CHAR <x>,<y>,<color>,<string>

Action: This command draws a text string on the graphics screen in a given color.

The string can contain printable ASCII characters (CHR\$(\$20) to CHR\$(\$7E)), as well most PETSCII control codes.

EXAMPLE of CHAR Statement:

```
10 SCREEN $80
20 A$="The quick brown fox jumps over the lazy dog."
24 CHAR 0,6,0,A$
30 CHAR 0,6+12,0,CHR$($04)+A$ :REM UNDERLINE
40 CHAR 0,6+12*2,0,CHR$($06)+A$ :REM BOLD
50 CHAR 0,6+12*3,0,CHR$($0B)+A$ :REM ITALICS
60 CHAR 0,6+12*4,0,CHR$($0C)+A$ :REM OUTLINE
70 CHAR 0,6+12*5,0,CHR$($12)+A$ :REM REVERSE
```

CLS

TYPE: Command

FORMAT: CLS

Action: Clears the screen. Same effect as ?CHR\$(147); .

EXAMPLE of CLS Statement:

```
CLS
```

COLOR

TYPE: Command

FORMAT: COLOR <fgcol>[,<bgcol>]

Action: This command works sets the text mode foreground color, and optionally the background color.

EXAMPLES of COLOR Statement:

```
COLOR 2 : REM SET FG COLOR TO RED, KEEP BG COLOR
COLOR 2,0 : REM SET FG COLOR TO RED, BG COLOR TO BLACK
```

DOS

TYPE: Command

FORMAT: DOS <string>

Action: This command works with the command/status channel or the directory of a Commodore DOS device and has different functionality depending on the type of argument.

- Without an argument, DOS prints the status string of the current device.
- With a string argument of "8" or "9", it switches the current device to the given number.

- With an argument starting with "\$" , it shows the directory of the device.
- Any other argument will be sent as a DOS command.

EXAMPLES of DOS Statement:

```
DOS"$"      : REM SHOWS DIRECTORY
DOS"S:BAD_FILE" : REM DELETES "BAD_FILE"
DOS         : REM PRINTS DOS STATUS, E.G. "01,FILES SCRATCHED,01,00"
```

EDIT**TYPE: Command****FORMAT: EDIT [<filename>]****Action:** Opens the built-in text editor, X16-Edit, a modeless editor with features similar to GNU Nano.

- Without an argument, the editor begins with an empty file.
- With a string argument, it attempts to load a file before displaying it.

The EDIT command loads the editor in the screen mode and character set that was active at the time the command was run.

EXAMPLE of EDIT Statement:

```
EDIT "README.TXT"
```

A more elaborate X16-Edit manual can be found [here](#)

EXEC**TYPE: Command****FORMAT: EXEC <memory address>[,<ram bank>]**

Action: Plays back a null-terminated script from MEMORY into the BASIC editor. Among other uses, this can be used to "type" in a program from a plain text file.

- If the ram bank argument is omitted and the address is in the range \$A000-\$BFFF, the RAM bank selected by the BANK command is used.
- The input can span multiple RAM banks. The input will stop once it reaches a null byte (\$00) or if a BASIC error occurs.
- The redirected input only applies to BASIC immediate mode. While programs are running, the EXEC handling is suspended.

EXAMPLE of EXEC Statement:

```
BLOAD "MYPROGRAM.BAS",8,1,$A000 : REM "BANK PEEK(0)" NO LONGER NEEDED
POKE PEEK($30D)+(PEEK($30E)*256),0 : REM NULL TERMINATE IN END BANK
EXEC $A000,1
```

FMCHORD**TYPE: Command****FORMAT: FMCHORD <first channel>,<string>**

Action: This command uses the same syntax as FMPLAY , but instead of playing a series of notes, it will start all of the notes in the string simultaneously on one or more channels. The first parameter to FMCHORD is the first channel to use, and

will be used for the first note in the string, and subsequent notes in the string will be started on subsequent channels, with the channel after 7 being channel 0.

All macros are supported, even the ones that only affect the behavior of `PSGPLAY` and `FMPLAY`.

The full set of macros is documented [here](#).

EXAMPLE of FMCHORD statement:

```
10 FMINIT
20 FMVIB 195,10
30 FMINST 1,16:FMINST 2,16:FMINST 3,16 : REM ORGAN
40 FMVOL 1,50:FMVOL 2,50:FMVOL 3,50 : REM MAKE ORGAN QUIETER
50 FMINST 0,11 : REM VIBRAPHONE
60 FMCHORD 1,"03CG>E T90" : REM START SOME ORGAN CHORDS (CHANNELS 1,2,3)
70 FMPLAY 1,"04G4.A8G4E2." : REM PLAY MELODY (CHANNEL 0)
80 FMPLAY 0,"04G4.A8G4E2."
90 FMCHORD 1,"02G>DB" : REM SWITCH ORGAN CHORDS (CHANNELS 1,2,3)
100 FMPLAY 0,"05D2D4<B2" : REM PLAY MORE MELODY
110 FMCHORD 1,"02F" : REM SWITCH ONE OF THE ORGAN CHORD NOTES
120 FMPLAY 0,"R4" : REM PAUSE FOR THE LENGTH OF ONE QUARTER NOTE
130 FMCHORD 1,"03CEG" : REM SWITCH ALL THREE CHORD NOTES
140 FMPLAY 0,"05C2C4<G2." : REM PLAY THE REST OF THE MELODY
150 FMCHORD 1,"RRR" : REM RELEASE THE CHANNELS THAT ARE PLAYING THE CHORD
```

This will play the first few lines of *Silent Night* with a vibraphone lead and organ accompaniment.

FMDRUM

TYPE: Command

FORMAT: FMDRUM <channel>,<drum number>

Action: Loads a [drum preset](#) onto the YM2151 and triggers it. Valid range is from 25 to 87, corresponding to the General MIDI percussion note values. FMDRUM will load a patch preset corresponding to the selected drum into the channel. If you then try to play notes on that same channel without loading an instrument patch, it will use the drum patch that was loaded for the drum sound instead, which may not sound particularly musical.

FMFREQ

TYPE: Command

FORMAT: FMFREQ <channel>,<frequency>

Action: Play a note by frequency on the YM2151. The accepted range is in Hz from 17 to 4434. FMFREQ also accepts a frequency of 0 to release the note.

EXAMPLE of FMFREQ statement:

```
0 FMINST 0,160 : REM LOAD PURE SINE PATCH
10 FMINST 1,160 : REM HERE TOO
20 FMFREQ 0,350 : REM PLAY A SINE WAVE AT 350 HZ
30 FMFREQ 1,440 : REM PLAY A SINE WAVE AT 440 HZ ON ANOTHER CHANNEL
40 FOR X=1 TO 10000 : NEXT X : REM DELAY A BIT
50 FMFREQ 0,0 : FMFREQ 1,0 : REM RELEASE BOTH CHANNELS
```

The above BASIC program plays a sound similar to a North American dial tone for a few seconds.

FMINIT**TYPE: Command****FORMAT: FMINIT****Action:** Initialize YM2151, silence all channels, and load a set of default patches into all 8 channels.**FMINST****TYPE: Command****FORMAT: FMINST <channel>,<patch>**

Load an instrument onto the YM2151 in the form of a [patch preset](#) into a channel. Valid channels range from 0 to 7. Valid patches range from 0 to 162.

FMNOTE**TYPE: Command****FORMAT: FMNOTE <channel>,<note>**

Action: Play a note on the YM2151. The note value is constructed as follows. Using hexadecimal notation, the first nybble is the octave, 0-7, and the second nybble is the note within the octave as follows:

\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7	\$x8	\$x9	\$xA	\$xB	\$xC	\$xD-\$xF
Release	C	C#/D \flat	D	D#/E \flat	E	F	F#/G \flat	G	G#/A \flat	A	A#/B \flat	B	no-op

Notes can also be represented by negative numbers to skip retriggering, and will thus snap to another note without restarting the playback of the note.

EXAMPLE of FMNOTE statement:

```

0 FMINST 1,64 : REM LOAD SOPRANO SAX
10 FMNOTE 1,$4A : REM PLAYS CONCERT A
20 FOR X=1 TO 5000 : NEXT X : REM DELAYS FOR A BIT
30 FMNOTE 1,0 : REM RELEASES THE NOTE
40 FOR X=1 TO 1000 : NEXT X : REM DELAYS FOR A BIT
50 FMNOTE 1,$3A : REM PLAYS A IN THE 3RD OCTAVE
60 FOR X=1 TO 2500 : NEXT X : REM SHORT DELAY
70 FMNOTE 1,-$3B : REM UP A HALF STEP TO A# WITHOUT RETRIGGERING
80 FOR X=1 TO 2500 : NEXT X : REM SHORT DELAY
90 FMNOTE 1,0 : REM RELEASES THE NOTE

```

FMPAN**TYPE: Command****FORMAT: FMPAN <channel>,<panning>****Action:** Sets the simple stereo panning on a YM2151 channel. Valid values are as follows:

- 1 = left
- 2 = right
- 3 = both

FMPLAY**TYPE: Command****FORMAT: FMPLAY <channel>,<string>**

Action: This command is very similar to `PLAY` on other BASICs such as GWBASIC. It takes a string of notes, rests, tempo changes, note lengths, and other macros, and plays all of the notes synchronously. That is, the `FMPLAY` command will not return control until all of the notes and rests in the string have been fully played.

The full set of macros is documented [here](#).

EXAMPLE of FMPLAY statement:

```
10 FMINIT : REM INITIALIZE AND LOAD DEFAULT PATCHES, WILL USE E.PIANO
20 FMPLAY 1,"T90 04 L4" : REM TEMPO 90 BPM, OCTAVE 4, NOTE LENGTH 4 (QUARTER)
30 FMPLAY 1,"CDECCDECEFGREFGR" : REM FIRST TWO LINES OF TUNE
40 FMPLAY 1,"G8A8G8F8EC G8A8G8F8EC" : REM THIRD LINE
50 FMPLAY 1,"C<G>CRC<G>CR" : REM FOURTH LINE
```

FMPOKE

TYPE: Command

FORMAT: `FMPOKE <register>,<value>`

Action: This command uses the AUDIO API to write a value to one of the the YM2151's registers at a low level.

EXAMPLE of FMPOKE statement:

```
10 FMINIT
20 FMPOKE $28,$4A : REM SET KC TO A4 (A-440) ON CHANNEL 0
30 FMPOKE $08,$00 : REM RELEASE CHANNEL 0
40 FMPOKE $08,$78 : REM START NOTE PLAYBACK ON CHANNEL 0 W/ ALL OPERATORS
```

FMVIB

TYPE: Command

FORMAT: `FMVIB <speed>,<depth>`

Action: This command sets the LFO speed and the phase and amplitude modulation depth values on the YM2151. The speed value ranges from 0 to 255, and corresponds to an LFO frequency from 0.008 Hz to 32.6 Hz. The depth value ranges from 0-127 and affects both AMD and PMD.

Only some patch presets (instruments) are sensitive to the LFO. Those are marked in [this table](#) with the † symbol. The LFO affects all channels equally, and it depends on the instrument as to whether it is affected.

Good values for most instruments are speed somewhere between 190-220. A good light vibrato for most wind instruments would have a depth of 10-15, while tremolo instruments like the Vibraphone or Tremolo Strings are most realistic around 20-30.

EXAMPLE of FMVIB statement:

```
10 FMVIB 200,30
20 FMINST 0,11 : REM VIBRAPHONE
30 FMPLAY 0,"T60 04 CDEFGAB>C"
40 FMVIB 0,0
50 FMPLAY 0,"C<BAGFEDC"
```

The above BASIC program plays a C major scale with a vibraphone patch, first with a vibrato/tremolo effect, and then plays the scale in reverse with the vibrato turned off.

FMVOL

TYPE: Command**FORMAT: FMVOL <channel>,<volume>**

Action: This command sets the channel's volume. The volume remains at the requested level until another `FMVOL` command for that channel or `FMINIT` is called. Valid range is from 0 (completely silent) to 63 (full volume)

FRAME**TYPE: Command****FORMAT: FRAME <x1>,<y1>,<x2>,<y2>,<color>**

Action: This command draws a rectangle frame on the graphics screen in a given color.

EXAMPLE of FRAME Statement:

```
10 SCREEN$80
20 FOR I=1 TO 20: FRAMERND(1)*320,RND(1)*200,RND(1)*320,RND(1)*200,RND(1)*128:NEXT I
30 GOTO 20
```

HEX\$**TYPE: String Function****FORMAT: HEX\$(n)**

Action: Return a string representing the hexadecimal value of n. If n <= 255, 2 characters are returned and if 255 < n <= 65535, 4 characters are returned.

EXAMPLE of HEX\$ Function:

```
PRINT HEX$(200) : REM PRINTS C8 AS HEXADECIMAL REPRESENTATION OF 200
PRINT HEX$(45231) : REM PRINTS B0AF TO REPRESENT 16 BIT VALUE
```

HELP**TYPE: Command****FORMAT: HELP**

Action: The `HELP` command displays a brief summary of the ROM build, and points users to this guide at its home on GitHub, and to the community forums website.

I2CPEEK**TYPE: Integer Function****FORMAT: I2CPEEK(<device>,<register>)**

Action: Returns the value from a register on an I²C device.

EXAMPLE of I2CPEEK Function:

```
PRINT HEX$(I2CPEEK($6F,0) AND $7F)
```

This command reports the seconds counter from the RTC by converting its internal BCD representation to a string.

I2CPOKE**TYPE: Command****FORMAT: I2CPOKE <device>,<register>,<value>**

Action: Sets the value to a register on an I²C device.

EXAMPLE of I2CPOKE Function:

```
I2CPOKE $6F,$40,$80
```

This command sets a byte in NVRAM on the RTC to the value `$80`

JOY

TYPE: Integer Function

FORMAT: JOY(n)

Action: Return the state of a joystick.

`JOY(1)` through `JOY(4)` return the state of SNES controllers connected to the system, and `JOY(0)` returns the state of the "keyboard joystick", a set of keyboard keys that map to the SNES controller layout. See [joystick_get](#) for details.

If no controller is connected to the SNES port (or no keyboard is connected), the function returns -1. Otherwise, the result is a bit field, with pressed buttons OR ed together:

Value	Button
\$800	A
\$400	X
\$200	L
\$100	R
\$080	B
\$040	Y
\$020	SELECT
\$010	START
\$008	UP
\$004	DOWN
\$002	LEFT
\$001	RIGHT

Note that this bitfield is different from the `joystick_get` KERNEL API one. Also note that the keyboard joystick will allow LEFT and RIGHT as well as UP and DOWN to be pressed at the same time, while controllers usually prevent this mechanically.

EXAMPLE of JOY Function:

```
10 REM DETECT CONTROLLER, FALL BACK TO KEYBOARD
20 J = 0: FOR I=1 TO 4: IF JOY(I) >= 0 THEN J = I: GOTO40
30 NEXT
40 :
50 V=JOY(J)
60 PRINT CHR$(147);V;" : "
```

```

70 IF V = -1 THEN PRINT"DISCONNECTED ": GOT050
80 IF V AND 8 THEN PRINT"UP ";
90 IF V AND 4 THEN PRINT"DOWN ";
100 IF V AND 2 THEN PRINT"LEFT ";
110 IF V AND 1 THEN PRINT"RIGHT ";
120 GOT050

```

KEYMAP

TYPE: Command

FORMAT: KEYMAP <string>

Action: This command sets the current keyboard layout. It can be put into an AUTOBOOT file to always set the keyboard layout on boot.

EXAMPLE of KEYMAP Statement:

```

10 KEYMAP"SV-SE"      :REM SMALL BASIC PROGRAM TO SET LAYOUT TO SWEDISH/SWEDEN
SAVE"AUTOBOOT.X16"    :REM SAVE AS AUTOBOOT FILE

```

LINE

TYPE: Command

FORMAT: LINE <x1>,<y1>,<x2>,<y2>,<color>

Action: This command draws a line on the graphics screen in a given color.

EXAMPLE of LINE Statement:

```

10 SCREEN128
20 FORA=0 TO 2*PI STEP 2*PI/200
30 : LINE100,100,100+SIN(A)*100,100+COS(A)*100
40 NEXT

```

If you're pasting this example into the Commander X16 emulator, use this code block instead so that the π symbol is properly received.

```

10 SCREEN128
20 FORA=0 TO 2*\XFF STEP 2*\XFF/200
30 : LINE100,100,100+SIN(A)*100,100+COS(A)*100
40 NEXT

```

LINPUT

TYPE: Command

****FORMAT: LINPUT <var\$>**

Action: LINPUT Reads a line of data from the keyboard and stores the data into a string variable. Unlike INPUT, no parsing or cooking of the input is done, and therefore quotes, commas, and colons are stored in the string as typed. No prompt is displayed, either.

The input is taken from the KERNAL editor, hence the user will have the freedom of all of the features of the editor such as cursor movement, mode switching, and color changing.

Due to how the editor works, an empty line will return " " - a string with a single space, and trailing spaces are not preserved.

EXAMPLE of LINPUT Statement:

```

10 LINPUT A$
20 IF A$="" THEN 50
30 PRINT "YOU TYPED: ";A$
40 END
50 PRINT "YOU TYPED AN EMPTY STRING: ";A$

```

LINPUT#**TYPE: Command****FORMAT: LINPUT# <n>,<var\$>[,<delimiter>]**

Action: LINPUT# Reads a line of data from an open file and stores the data into a string variable. The delimiter of a line by default is 13 (carriage return). The delimiter is not part of the stored value. If the end of the file is reached while reading, ST AND 64 will be true.

LINPUT# can be used to read structured data from files. It can be particularly useful to extract quoted or null-terminated strings from files while reading.

EXAMPLE of LINPUT# Statement:

```

10 I=0
20 OPEN 1,8,0,"$"
30 LINPUT#1,A$, $22
40 IF ST<>0 THEN 130
50 LINPUT#1,A$, $22
60 IF I=0 THEN 90
70 PRINT "ENTRY: ";
80 GOTO 100
90 PRINT "LABEL: ";
100 PRINT CHR$( $22 );A$;CHR$( $22 )
110 I=I+1
120 IF ST=0 THEN 30
130 CLOSE 1

```

The above example parses and prints out the filenames from a directory listing.

LOCATE**TYPE: Command****FORMAT: LOCATE <line>[,<column>]**

Action: This command positions the text mode cursor at the given location. The values are 1-based. If no column is given, only the line is changed.

EXAMPLE of LOCATE Statement:

```

100 REM DRAW CIRCLE ON TEXT SCREEN
110 SCREEN0
120 R=25
130 X0=40
140 Y0=30
150 FORT=0T0360STEP1
160 : X=X0+R*COS(T)
170 : Y=Y0+R*SIN(T)

```

```
180 : LOCATEY,X:PRINTCHR$($12);" ";
190 NEXT
```

MENU

TYPE: Command

FORMAT: MENU

Action: This command currently invokes the Commander X16 Control Panel. In the future, the menu may instead present a menu of ROM-based applications and routines.

EXAMPLE of MON Statement:

```
MENU
```

MON

TYPE: Command

FORMAT: MON (Alternative: MONITOR)

Action: This command enters the machine language monitor. See the [dedicated chapter](#) for a description.

EXAMPLE of MON Statement:

```
MON
MONITOR
```

MOUSE

TYPE: Command

FORMAT: MOUSE <mode>

Action: This command configures the mouse pointer.

Mode	Description
0	Hide mouse
1	Show mouse, set default mouse pointer
-1	Show mouse, don't configure mouse cursor

`MOUSE 1` turns on the mouse pointer and `MOUSE 0` turns it off. If the BASIC program has its own mouse pointer sprite configured, it can use `MOUSE -1`, which will turn the mouse pointer on, but not set the default pointer sprite.

The size of the mouse pointer's area will be configured according to the current screen mode. If the screen mode is changed, the `MOUSE` statement has to be repeated.

EXAMPLES of MOUSE Statement:

```
MOUSE 1 : REM ENABLE MOUSE
MOUSE 0 : REM DISABLE MOUSE
```

MOVSPR

TYPE: Command

FORMAT: MOVSPR <sprite idx>,<x>,<y>

Action: This command positions a sprite's upper left corner at a specific pixel location. It does not change its visibility.

`sprite idx` is a value between 0-127 inclusive. `x` and `y` have a range of -32768 to 32767 inclusive, but their meanings wrap every 1024 values. Values approaching 1023 will peek out from the left and top of the screen for `x` and `y` respectively as if they were negative and approaching 0. -1024, 1024, 0, and 2048 are all equivalent. Likewise, -10 and 1014 are equivalent.

EXAMPLE of MOVSPR Statement:

```
10 BVLOAD "MYSPRITE.BIN",8,1,$3000
20 SPRMEM 1,1,$3000,1
30 SPRITE 1,3,0,0,3,3
40 MOVSPR 1,320,200
```

MX/MY/MB

TYPE: System variable

FORMAT: MX

FORMAT: MY

FORMAT: MB

Action: Return the horizontal (`MX`) or vertical (`MY`) position of the mouse pointer, or the mouse button state (`MB`).

`MB` returns the sum of the following values depending on the state of the buttons:

Value	Button
0	none
1	left
2	right
4	third

EXAMPLE of MX/MY/MB variables:

```
REM SIMPLE DRAWING PROGRAM
10 SCREEN$80
20 MOUSE1
25 OB=0
30 TX=MX:TY=MY:TB=MB
35 IFTB=0GOTO25
40 IF0BTHENLINEOX,OY,TX,TY,16
50 IF0B=0THENPSETTX,TY,16
60 OX=TX:OY=TY:OB=TB
70 GOTO30
```

MWHEEL

TYPE: System variable

FORMAT: MWHEEL

Action: Return the mouse scroll wheel movement since the value was last read. The value is negative if the scroll wheel is moved away from the user, and positive if it is moved towards the user. The range of the returned value is -128 to +127.

OLD

TYPE: Command

FORMAT: OLD

Action: This command recovers the BASIC program in RAM that has been previously deleted using the `NEW` command or through a RESET.

EXAMPLE of OLD Statement:

```
OLD
```

POINTER

TYPE: Function

FORMAT: POINTER(<variable>)

Action: Returns the memory address of the internal structure representing a BASIC variable.

EXAMPLE of POINTER function:

```
10 A$="MOO"  
20 PRINT HEX$(POINTER(A$))  
RUN  
0823
```

POWEROFF

TYPE: Command

FORMAT: POWEROFF

Action: This command instructs the SMC to power down the system. This is equivalent to pressing the physical power switch.

EXAMPLE of POWEROFF Statement:

```
POWEROFF
```

PSET

TYPE: Command

FORMAT: PSET <x>,<y>,<color>

Action: This command sets a pixel on the graphics screen to a given color.

EXAMPLE of PSET Statement:

```
10 SCREEN$80  
20 FOR I=1 TO 20: PSET RND(1)*320, RND(1)*200, RND(1)*256: NEXT  
30 GOTO 20
```

PSGCHORD

TYPE: Command

FORMAT: PSGCHORD <first voice>,<string>

Action: This command uses the same syntax as `PSGPLAY`, but instead of playing a series of notes, it will start all of the notes in the string simultaneously on one or more voices. The first parameter to `PSGCHORD` is the first voice to use, and will be used for the first note in the string, and subsequent notes in the string will be started on subsequent voices, with the voice after 15 being voice 0.

All macros are supported, even the ones that only affect `PSGPLAY` and `FMPLAY`.

The full set of macros is documented [here](#).

EXAMPLE of PSGCHORD statement:

```
10 PSGINIT
20 PSGCHORD 15,"03G>CE" : REM STARTS PLAYING A CHORD ON VOICES 15, 0, AND 1
30 PSGPLAY 14,">C<DGB>CDE" : REM PLAYS A SERIES OF NOTES ON VOICE 14
40 PSGCHORD 15,"RRR" : REM RELEASES CHORD ON VOICES 15, 0, AND 1
50 PSGPLAY 14,"04CAG>C<A" : REM PLAYS A SERIES OF NOTES ON VOICE 14
60 PSGCHORD 0,"03A>CF" : REM STARTS PLAYING A CHORD ON VOICES 0, 1, AND 2
70 PSGPLAY 14,"L16FGAB->CDEF4" : REM PLAYS A SERIES OF NOTES ON VOICE
80 PSGCHORD 0,"RRR" : REM RELEASES CHORD ON VOICES 0, 1, AND 2
```

PSGFREQ

TYPE: Command

FORMAT: PSGFREQ <voice>,<frequency>

Action: Play a note by frequency on the VERA PSG. The accepted range is in Hz from 1 to 24319. `PSGFREQ` also accepts a frequency of 0 to release the note.

EXAMPLE of PSGFREQ statement:

```
10 PSGINIT : REM RESET ALL VOICES TO SQUARE WAVEFORM
20 PSGFREQ 0,350 : REM PLAY A SQUARE WAVE AT 350 HZ
30 PSGFREQ 1,440 : REM PLAY A SQUARE WAVE AT 440 HZ ON ANOTHER VOICE
40 FOR X=1 TO 10000 : NEXT X : REM DELAY A BIT
50 PSGFREQ 0,0 : PSGFREQ 1,0 : REM RELEASE BOTH VOICES
```

The above BASIC program plays a sound similar to a North American dial tone for a few seconds.

PSGINIT

TYPE: Command

FORMAT: PSGINIT

Action: Initialize VERA PSG, silence all voices, set volume to 63 on all voices, and set the waveform to pulse and the duty cycle to 63 (50%) for all 16 voices.

PSGNOTE

TYPE: Command

FORMAT: PSGNOTE <voice>,<note>

Action: Play a note on the VERA PSG. The note value is constructed as follows. Using hexadecimal notation, the first nybble is the octave, 0-7, and the second nybble is the note within the octave as follows:

\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7	\$x8	\$x9	\$xA	\$xB	\$xC	\$xD-\$xF
Release	C	C#/D \flat	D	D#/E \flat	E	F	F#/G \flat	G	G#/A \flat	A	A#/B \flat	B	no-op

EXAMPLE of PSGNOTE statement:

```

10 PSGNOTE 1,$4A : REM PLAYS CONCERT A
20 FOR X=1 TO 5000 : NEXT X : REM DELAYS FOR A BIT
30 PSGNOTE 1,0 : REM RELEASES THE NOTE
40 FOR X=1 TO 2500 : NEXT X : REM SHORT DELAY
50 PSGNOTE 1,$3A : REM PLAYS A IN THE 3RD OCTAVE
60 FOR X=1 TO 2500 : NEXT X : REM SHORT DELAY
70 PSGNOTE 1,0 : REM RELEASES THE NOTE

```

PSGPAN**TYPE: Command****FORMAT: PSGPAN <voice>,<panning>****Action:** Sets the simple stereo panning on a VERA PSG voice. Valid values are as follows:

- 1 = left
- 2 = right
- 3 = both

PSGPLAY**TYPE: Command****FORMAT: PSGPLAY <voice>,<string>****Action:** This command is very similar to `PLAY` on other BASICs such as GWBASIC. It takes a string of notes, rests, tempo changes, note lengths, and other macros, and plays all of the notes synchronously. That is, the PSGPLAY command will not return control until all of the notes and rests in the string have been fully played.The full set of macros is documented [here](#).**EXAMPLE of PSGPLAY statement:**

```

10 PSGWAV 0,31 : REM PULSE, 25% DUTY
20 PSGPLAY 0,"T180 S0 05 L32" : REM TEMPO 180 BPM, LEGATO, OCTAVE 5, 32ND NOTES
30 PSGPLAY 0,"C<G>CEG>C<G<A-"
40 PSGPLAY 0,">CE-A-E-A->CE-A-"
50 PSGPLAY 0,"E-<<B->DFB-FB->DFB-F" : REM GRAB YOURSELF A MUSHROOM

```

PSGVOL**TYPE: Command****FORMAT: PSGVOL <voice>,<volume>****Action:** This command sets the voice's volume. The volume remains at the requested level until another `PSGVOL` command for that voice or `PSGINIT` is called. Valid range is from 0 (completely silent) to 63 (full volume).**PSGWAV****TYPE: Command****FORMAT: PSGWAV <voice>,<w>****Action:** Sets the waveform and duty cycle for a PSG voice.

- w = 0-63 -> Pulse: Duty cycle is $(w+1)/128$. A value of 63 means 50% duty.
- w = 64-127 -> Sawtooth (all values have identical effect)
- w = 128-191 -> Triangle (all values have identical effect)

- w = 192-255 -> Noise (all values have identical effect)

EXAMPLE of PSGWAV Statement:

```

10 FOR O=$20 TO $50 STEP $10:REM OCTAVE LOOP
20 FOR N=1 TO 11 STEP 2:REM NOTE LOOP, EVERY OTHER NOTE
30 PSGNOTE 0,0+N:REM START PLAYBACK OF THE NOTE
40 FOR P=0 TO 30:REM PULSE WIDTH MODULATION LOOP (INCREASING DUTY)
50 PSGWAV 0,P:REM SET PW
60 FOR D=1 TO 30:NEXT D:REM DELAY LOOP
70 NEXT P
80 PSGNOTE 0,0+N+1:REM START PLAYBACK OF THE NOTE + A SEMITONE
90 FOR P=31 TO 1 STEP -1:REM PWM LOOP (DECREASING DUTY)
100 PSGWAV 0,P:REM SET PW
110 FOR D=1 TO 30:NEXT D:REM DELAY LOOP
120 NEXT P
130 NEXT N
140 NEXT O
150 PSGNOTE 0,0:REM STOP SOUND

```

This example plays a chromatic scale while applying pulse-width modulation on the voice.

RECT

TYPE: Command

FORMAT: RECT <x1>,<y1>,<x2>,<y2>,<color>

Action: This command draws a solid rectangle on the graphics screen in a given color.

EXAMPLE of RECT Statement:

```

10 SCREEN$80
20 FOR I=1 TO 20:RECT RND(1)*320,RND(1)*200,RND(1)*320,RND(1)*200,RND(1)*256:NEXT
30 GOTO 20

```

REBOOT

TYPE: Command

FORMAT: REBOOT

Action: Performs a software reset of the system by calling the ROM reset vector.

EXAMPLE of REBOOT Statement:

```
REBOOT
```

REN

TYPE: Command

FORMAT: REN [<new line num>[, <increment>[, <first old line num>]]]

Action: Renumbers a BASIC program while updating the line number arguments of GOSUB, GOTO, RESTORE, RUN, and THEN.

Optional arguments:

- The line number of the first line after renumbering, default: **10**

- The value of the increment for subsequent lines, default **10**
- The earliest old line to start renumbering at, default: **0**

THIS STATEMENT IS EXPERIMENTAL. Please ensure you have saved your program before using this command to renumber.

KNOWN BUG: In release R43, due to improper parsing of escape tokens, REN will improperly treat arguments to these statements as line numbers:

- FRAME
- RECT
- MOUSE
- COLOR
- PSGWAV

This behavior has been fixed in R44.

EXAMPLE of REN Statement:

```
10 PRINT "HELLO"
20 DATA 1,2,3
30 DATA 4,5,6
40 READ X
50 PRINT X
60 RESTORE 30
70 READ X
80 PRINT X
90 GOTO 10

REN 100,5

LIST
100 PRINT "HELLO"
105 DATA 1,2,3
110 DATA 4,5,6
115 READ X
120 PRINT X
125 RESTORE 110
130 READ X
135 PRINT X
140 GOTO 100
READY.
```

RESET

TYPE: Command

FORMAT: RESET

Action: This command instructs the SMC to assert the reset line on the system, which performs a hard reset. This is equivalent to pressing the physical reset switch.

EXAMPLE of RESET Statement:

```
RESET
```

RESTORE

TYPE: Command**FORMAT: RESTORE [<linenum>]**

Action: This command resets the pointer for the `READ` command. Without arguments, it will reset the pointer to the first `DATA` constant in the program. With a parameter `linenum`, the command will reset the pointer to the first `DATA` constant at or after that line number.

EXAMPLE of RESTORE Statement:

```
10 DATA 1,2,3
20 DATA 4,5,6
30 READ Y
40 PRINT Y
50 RESTORE 20
60 READ Y
70 PRINT Y
```

This program will output the number 1 followed by the number 4.

RPT\$**TYPE: Function****FORMAT: RPT\$(<byte>,<count>)**

Action: Returns a string of <count> instances of the PETSCII character represented by the numeric value <byte>. This function is similar in behavior to `CHR$()` but takes a second argument as a repeat count.

`RPT$(A,1)` is functionally equivalent to `CHR$(A)`.

EXAMPLE of RPT\$ function:

```
10 REM TEN EXCLAMATION MARKS
20 PRINT RPT$(33,10)
READY.
RUN
!!!!!!!!!!
READY.
```

SCREEN**TYPE: Command****FORMAT: SCREEN <mode>**

Action: This command switches screen modes.

For a list of supported modes, see [Chapter 2: Editor](#). The value of -1 toggles between modes \$00 and \$03.

EXAMPLE of SCREEN Statement:

```
SCREEN 3 : REM SWITCH TO 40 CHARACTER MODE
SCREEN 0 : REM SWITCH TO 80 CHARACTER MODE
SCREEN -1 : REM SWITCH BETWEEN 40 and 80 CHARACTER MODE
```

SLEEP

TYPE: Command**FORMAT: SLEEP [<jiffies>]**

Action: With the default interrupt source configured and enabled, this command waits for `jiffies + 1` VSYNC events and then resumes program execution. In other words, `SLEEP` with no arguments is equivalent to `SLEEP 0`, which waits until the beginning of the next frame. Another useful example, `SLEEP 60`, pauses for approximately 1 second.

Allowed values for `jiffies` is from 0 to 65535, inclusive.

EXAMPLE of SLEEP Statement:

```
10 FOR I=1 TO 10
20 PRINT I
30 SLEEP 60
40 NEXT
```

SPRITE**TYPE: Command****FORMAT: SPRITE <sprite idx>,<priority>[,<palette offset>[,<flip>[,<x-width>[,<y-width>[,<color depth>]]]]]**

Action: This command configures a sprite's geometry, palette, and visibility.

The first two arguments are required, but the remainder are optional.

- `sprite idx` is a value between 0-127 inclusive.
- `priority`, also known as z-depth changes the visibility of the sprite and above which layer it is rendered. Range is 0-3 inclusive. 0 = off, 1 = below layer 0, 2 = in between layers 0 and 1, 3 = above layer 1
- `palette offset` is the palette offset for the sprite. Range is 0-15 inclusive. This value is multiplied by 16 to determine the starting palette index.
- `flip` controls the X and Y flipping of the sprite. Range is 0-3 inclusive. 0 = unflipped, 1 = X is flipped, 2 = Y is flipped, 3 = both X and Y are flipped.
- `x-width` and `y-width` represent the dimensions of the sprite. Range is 0-3 inclusive. 0 = 8px, 1 = 16px, 2 = 32px, 3 = 64px.
- `color depth` selects either 4 or 8-bit color depth for the sprite. 0 = 4-bit, 1 = 8-bit. This attribute can also be set by the `SPRMEM` command.

Note: If VERA's sprite layer is disabled when the `SPRITE` command is called, the sprite layer will be enabled, regardless of the arguments to `SPRITE`.

EXAMPLE of SPRITE Statement:

```
10 BVLOAD "MYSprite.BIN",8,1,$3000
20 SPRMEM 1,1,$3000,1
30 SPRITE 1,3,0,0,3,3
40 MOVSPR 1,320,200
```

SPRMEM**TYPE: Command****FORMAT: SPRMEM <sprite idx>,<VRAM bank>,<VRAM address>[,<color depth>]**

Action: This command configures the address of where the sprite's pixel data is to be found. It also can change or set the color depth of the sprite.

The first three arguments are required, but the last one is optional.

- `sprite idx` is a value between 0-127 inclusive.
- `VRAM bank` is a value, 0 or 1, which represents which of the two 64k regions of VRAM to select.
- `VRAM address` is a 16-bit value, \$0000-\$FFFF, is the address within the VRAM bank to point the sprite to. The lowest 5 bits are ignored.
- `color depth` selects either 4 or 8-bit color depth for the sprite. 0 = 4-bit, 1 = 8-bit. This attribute can also be set by the `SPRITE` command.

EXAMPLE of SPRITE Statement:

```
10 BVLOAD "MYSPRITE.BIN",8,1,$3000
20 SPRMEM 1,1,$3000,1
30 SPRITE 1,3,0,0,3,3
40 MOVSPR 1,320,200
```

STRPTR**TYPE: Function****FORMAT: STRPTR(<variable>)**

Action: Returns the memory address of the first character of a string contained within a string variable. If the string variable has zero length, this function will likely still return a non-zero value pointing either to the close quotation mark in the literal assignment, or to somewhere undefined in string memory. Programs should check the `LEN()` of string variables before using the pointer returned from `STRPTR`.

EXAMPLE of STRPTR function:

```
10 A$="MOO"
20 P=STRPTR(A$)
30 FOR I=0 TO LEN(A$)-1
40 PRINT CHR$(PEEK(P+I));
50 NEXT
60 A$=""
70 P=STRPTR(A$)
80 FOR I=0 TO LEN(A$)-1 : REM THIS LOOP WILL STILL ALWAYS HAPPEN ONCE
90 PRINT CHR$(PEEK(P+I));
100 NEXT
RUN
MOO"
READY.
```

In this case, the pointer returned on line 70 pointed to the first character after the open quote on line 60. Since it was an empty string, the pointer ended up pointing to the close quote. To avoid this scenario, we should have checked the `LEN(A$)` before line 80 and skipped over the loop.

SYS**TYPE: Command****FORMAT: SYS <address>**

Action: The SYS command executes a machine language subroutine located at <address>. Execution continues until an RTS is executed, and control returns to the BASIC program.

In order to communicate with the routine, you can pre-load the CPU registers by using POKE to write to the following memory locations:

- `$030C` : Accumulator

- \$030D : X Register
- \$030E : Y Register
- \$030F : Status Register/Flags

When the routine is over, the CPU registers will be loaded back in to these locations. So you can read the results of a machine language routine by PEEKing these locations.

EXAMPLE of SYS statemet:

Push a <CR> into the keyboard buffer.

```
POKE $30C,13
SYS $FEC3
```

Run the Machine Language Monitor (Supermon)

```
SYS $FECC
```

TILE

TYPE: Command

FORMAT: TILE <x>,<y>,<tile/screen code>[,<attribute>]

Action: The TILE command sets the tile or text character at the given x/y tile/character coordinate to the given screen code or tile index, optionally resetting the attribute byte. It works for tiles or text on Layer 1.

In the default text mode, this can be used to quickly change a character on the screen and optionally its fg/bg color without needing to calculate the VRAM address for VPOKE.

However, it can also be used if VERA Layer 1's map base value is changed or the map size is changed.

EXAMPLE of TILE command:

```
10 REM VERY SLOWLY CLEAR THE SCREEN IN STYLE
20 FOR Y=59 TO 0 STEP -1
30 FOR X=79 TO 0 STEP -1
40 FOR I=255 TO 32 STEP -1
50 TILE X,Y,I
60 NEXT:NEXT:NEXT
```

VPEEK

TYPE: Integer Function

FORMAT: VPEEK (<bank>,<address>)

Action: Return a byte from the video address space. The video address space has 17 bit addresses, which is exposed as 2 banks of 65536 addresses each.

In addition, VPEEK can reach add-on VERA cards with higher bank numbers.

BANK 2-3 is for IO3 (VERA at \$9F60-\$9F7F)

BANK 4-5 is for IO4 (VERA at \$9F80-\$9F9F)

EXAMPLE of VPEEK Function:

```
PRINT VPEEK(1,$B000) : REM SCREEN CODE OF CHARACTER AT 0/0 ON SCREEN
```

VPOKE

TYPE: Command

FORMAT: VPOKE <bank>, <address>, <value>

Action: Set a byte in the video address space. The video address space has 17 bit addresses, which is exposed as 2 banks of 65536 addresses each.

In addition, VPOKE can reach add-on VERA cards with higher bank numbers.

BANK 2-3 is for IO3 (VERA at \$9F60-\$9F7F)

BANK 4-5 is for IO4 (VERA at \$9F80-\$9F9F)

EXAMPLE of VPOKE Statement:

```
VPOKE 1,$B000+1,1 * 16 + 2 : REM SETS THE COLORS OF THE CHARACTER
                                REM AT 0/0 TO RED ON WHITE
```

VLOAD

TYPE: Command

FORMAT: VLOAD <filename>, <device>, <VERA_high_address>, <VERA_low_address>

Action: Loads a file directly into VERA RAM, skipping the two-byte header that is presumed to be in the file.

EXAMPLES of VLOAD:

```
VLOAD "MYFILE.PRG", 8, 0, $4000 :REM LOADS MYFILE.PRG FROM DEVICE 8 TO VRAM $4000
                                REM WHILE SKIPPING THE FIRST TWO BYTES OF THE FILE.
```

To load a raw binary file without skipping the first two bytes, use [BVLOAD](#)

Other New Features

Hexadecimal and Binary Literals

The numeric constants parser supports both hex (\$) and binary (%) literals, like this:

```
PRINT $EA31 + %1010
```

The size of hex and binary values is only restricted by the range that can be represented by BASIC's internal floating point representation.

LOAD into VRAM

In BASIC, the contents of files can be directly loaded into VRAM with the `LOAD` statement. When a secondary address greater than one is used, the KERNAL will now load the file into the VERA's VRAM address space. The first two bytes of the file are used as lower 16 bits of the address. The upper 4 bits are (SA-2) & 0x0ff where SA is the secondary address.

Examples:

```
10 REM LOAD VERA SETTINGS
20 LOAD"VERA.BIN",1,17 : REM SET ADDRESS TO $FXXXX
30 REM LOAD TILES
40 LOAD"TILES.BIN",1,3 : REM SET ADDRESS TO $1XXXX
```

```
50 REM LOAD MAP
60 LOAD"MAP.BIN",1,2 : REM SET ADDRESS TO $0XXXX
```

Default Device Numbers

In BASIC, the LOAD, SAVE and OPEN statements default to the last-used IEEE device (device numbers 8 and above), or 8.

Internal Representation

Like on the C64, BASIC keywords are tokenized.

- The C64 BASIC V2 keywords occupy the range of \$80 (`END`) to \$CB (`GO`).
- BASIC V3.5 also used \$CE (`RGR`) to \$FD (`WHILE`).
- BASIC V7 introduced the \$CE escape code for function tokens \$CE-\$02 (`POT`) to \$CE-\$0A (`POINTER`), and the \$FE escape code for statement tokens \$FE-\$02 (`BANK`) to \$FE-\$38 (`SLOW`).
- The unreleased BASIC V10 extended the escaped tokens up to \$CE-\$0D (`RPALETTE`) and \$FE-\$45 (`EDIT`).

The X16 BASIC aims to be as compatible as possible with this encoding. Keywords added to X16 BASIC that also exist in other versions of BASIC match the token, and new keywords are encoded in the ranges \$CE-\$80+ and \$FE-\$80+.

Auto-Boot

When BASIC starts, it automatically executes the `BOOT` command, which tries to load a PRG file named `AUTOBOOT.X16` from device 8 and, if successful, runs it. Here are some use cases for this:

- An SD card with a game can auto-boot this way.
- An SD card with a collection of applications can show a menu that allows selecting an application to load.
- The user's "work" SD card can contain a small auto-boot BASIC program that sets the keyboard layout and changes the screen colors, for example.

Chapter 3: KERNAL

The Commander X16 contains a version of KERNAL as its operating system in ROM. It contains

- "Channel I/O" API for abstracting devices
- a variable size screen editor
- a color bitmap graphics API with proportional fonts
- simple memory management
- timekeeping
- drivers
 - PS/2 keyboard and mouse
 - NES/SNES controller
 - Commodore Serial Bus ("IEC")
 - I2C bus

KERNAL Version

The KERNAL version can be read from location \$FF80 in ROM. A value of \$FF indicates a custom build. All other values encode the build number. Positive numbers are release versions (\$02 = release version 2), two's complement negative numbers are prerelease versions (\$FE = $100 - 2$ = prerelease version 2).

Compatibility Considerations

For applications to remain compatible between different versions of the ROM, they can rely upon:

- the KERNAL API calls at \$FF81-\$FFF3
- the KERNAL vectors at \$0314-\$0333

The following features must not be relied upon:

- the zero page and \$0200+ memory layout
- direct function offsets in the ROM

Commodore 64 API Compatibility

The KERNAL [fully supports](#) the C64 KERNAL API.

These routines have been stable ever since the C64 came out and are extensively documented in various resources dedicated to the C64. Currently, they are not documented *here* so if you need to look them up, here is a very thorough [reference of these standard kernal routines](#) (hosted on M. Steil's website). It integrates a dozen or so different sources for documentation about these routines.

Commodore 128 API Compatibility

In addition, the X16 [supports a subset](#) of the C128 API.

The following C128 APIs have equivalent functionality on the X16 but are not compatible:

Address	C128 Name	X16 Name
\$FF5F	SWAPPER	screen_mode
\$FF62	DLCHR	screen_set_charset
\$FF74	FETCH	fetch
\$FF77	STASH	stash

New API for the Commander X16

There are lots of new APIs. Please note that their addresses and their behavior is still preliminary and can change between revisions.

Some new APIs use the "16 bit" ABI, which uses virtual 16 bit registers r0 through r15, which are located in zero page locations \$02 through \$21: r0 = r0L = \$02, r0H = \$03, r1 = r1L = \$04 etc.

The 16 bit ABI generally follows the following conventions:

- arguments
 - word-sized arguments: passed in r0-r5
 - byte-sized arguments: if three or less, passed in .A, .X, .Y; otherwise in 16 bit registers
 - boolean arguments: .C, .N
- return values
 - basic rules as above
 - function takes no arguments: r0-r5, else indirect through passed-in pointer
 - arguments in r0-r5 can be "inout", i.e. they can be updated
- saved/scratch registers
 - r0-r5: arguments (saved)
 - r6-r10: saved
 - r11-r15: scratch
 - .A, .X, .Y, .C, .N: scratch (unless used otherwise)


KERNAL API functions

Label	Address	Class	Description	Inputs	Affects	Origin
ACPTR	\$FFA5	CPB	Read byte from peripheral bus		A X	C64
BASIN	\$FFCF	ChIO	Get character		A X	C64
BSAVE	\$FEBA	ChIO	Like SAVE but omits the 2-byte header	A X Y	A X Y	X16
BSOUT	\$FFD2	ChIO	Write byte in A to default output. For writing to a file must call OPEN and CHKOUT beforehand.	A	C	C64
CIOUT	\$FFA8	CPB	Send byte to peripheral bus	A	A X	C64
CLALL	\$FFE7	ChIO	Close all channels		A X	C64
CLOSE	\$FFC3	ChIO	Close a channel	A	A X Y P	C64
CHKIN	\$FFC6	ChIO	Set channel for character input	X	A X	C64
clock_get_date_time	\$FF50	Time	Get the date and time	none	r0 r1 r2 r3 A X Y P	X16
clock_set_date_time	\$FF4D	Time	Set the date and time	r0 r1 r2 r3	A X Y P	X16

CHRIN	\$FFCF	ChIO	Alias for BASIN		A X	C64
CHROUT	\$FFD2	ChIO	Alias for BSOUT	A	C	C64
CLOSE_ALL	\$FF4A	ChIO	Close all files on a device			C128
CLRCHN	\$FFCC	ChIO	Restore character I/O to screen/keyboard		A X	C64
console_init	\$FEDB	Video	Initialize console mode	none	r0 A P	X16
console_get_char	\$FEE1	Video	Get character from console	A	r0 r1 r2 r3 r4 r5 r6 r12 r13 r14 r15 A X Y P	X16
console_put_char	\$FEDE	Video	Print character to console	A C	r0 r1 r2 r3 r4 r5 r6 r12 r13 r14 r15 A X Y P	X16
console_put_image	\$FED8	Video	Draw image as if it was a character	r0 r1 r2	r0 r1 r2 r3 r4 r5 r14 r15 A X Y P	X16
console_set_paging_message	\$FED5	Video	Set paging message or disable paging	r0	A P	X16
enter_basic	\$FF47	Misc	Enter BASIC	C	A X Y P	X16
entropy_get	\$FECF	Misc	get 24 random bits	none	A X Y P	X16
fetch	\$FF74	Mem	Read a byte from any RAM or ROM bank	(A) X Y	A X P	X16
FB_cursor_next_line †	\$FF02	Video	Move direct-access cursor to next line	r0†	A P	X16
FB_cursor_position	\$FEFF	Video	Position the direct-access cursor	r0 r1	A P	X16
FB_fill_pixels	\$FF17	Video	Fill pixels with constant color, update cursor	r0 r1 A	A X Y P	X16
FB_filter_pixels	\$FF1A	Video	Apply transform to pixels, update cursor	r0 r1	r14H r15 A X Y P	X16
FB_get_info	\$FEF9	Video	Get screen size and color depth	none	r0 r1 A P	X16
FB_get_pixel	\$FF05	Video	Read one pixel, update cursor	none	A	X16
FB_get_pixels	\$FF08	Video	Copy pixels into RAM, update cursor	r0 r1	(r0) A X Y P	X16
FB_init	\$FEF6	Video	Enable graphics mode	none	A P	X16
FB_move_pixels	\$FF1D	Video	Copy horizontally consecutive pixels to a	r0 r1 r2 r3 r4	A X Y P	X16

			different position			
FB_set_8_pixels	\$FF11	Video	Set 8 pixels from bit mask (transparent), update cursor	A X	A P	X16
FB_set_8_pixels_opaque	\$FF14	Video	Set 8 pixels from bit mask (opaque), update cursor	r0L A X Y	r0L A P	X16
FB_set_palette	\$FEFC	Video	Set (parts of) the palette	A X r0	A X Y P	X16
FB_set_pixel	\$FF0B	Video	Set one pixel, update cursor	A	none	X16
FB_set_pixels	\$FF0E	Video	Copy pixels from RAM, update cursor	r0 r1	A X P	X16
GETIN	\$FFE4	Kbd	Get character from keyboard		A X	C64
GRAPH_clear	\$FF23	Video	Clear screen	none	r0 r1 r2 r3 A X Y P	X16
GRAPH_draw_image	\$FF38	Video	Draw a rectangular image	r0 r1 r2 r3 r4	A P	X16
GRAPH_draw_line	\$FF2C	Video	Draw a line	r0 r1 r2 r3	r0 r1 r2 r3 r7 r8 r9 r10 r12 r13 A X Y P	X16
GRAPH_draw_oval 	\$FF35	Video	Draw an oval or circle	-	-	X16
GRAPH_draw_rect †	\$FF2F	Video	Draw a rectangle (optionally filled)	r0 r1 r2 r3 r4 C	A P	X16
GRAPH_get_char_size	\$FF3E	Video	Get size and baseline of a character	A X	A X Y P	X16
GRAPH_init	\$FF20	Video	Initialize graphics	r0	r0 r1 r2 r3 A X Y P	X16
GRAPH_move_rect †	\$FF32	Video	Move pixels	r0 r1 r2 r3 r4 r5	r1 r3 r5 A X Y P	X16
GRAPH_put_char †	\$FF41	Video	Print a character	r0 r1 A	r0 r1 A X Y P	X16
GRAPH_set_colors	\$FF29	Video	Set stroke, fill and background colors	A X Y	none	X16
GRAPH_set_font	\$FF3B	Video	Set the current font	r0	r0 A Y P	X16
GRAPH_set_window †	\$FF26	Video	Set clipping region	r0 r1 r2 r3	A P	X16
i2c_batch_read	\$FEB4	I2C	Read multiple bytes from an I2C device	X r0 r1 C	A Y C	X16
i2c_batch_write	\$FEB7	I2C	Write multiple bytes to an I2C device	X r0 r1 C	A Y r2 C	X16

i2c_read_byte	\$FEC6	I2C	Read a byte from an I2C device	A X Y	A C	X16
i2c_write_byte	\$FEC9	I2C	Write a byte to an I2C device	A X Y	A C	X16
IOBASE	\$FFF3	Misc	Return start of I/O area		X Y	C64
JSRFAR	\$FF6E	Misc	Execute a routine on another RAM or ROM bank	PC+3 PC+5	none	X16
joystick_get	\$FF56	Joy	Get one of the saved controller states	A	A X Y P	X16
joystick_scan	\$FF53	Joy	Poll controller states and save them	none	A X Y P	X16
kdbuf_get_modifiers	\$FEC0	Kbd	Get currently pressed modifiers	A	A X P	X16
kdbuf_peek	\$FEBD	Kbd	Get next char and keyboard queue length	A X	A X P	X16
kdbuf_put	\$FEC3	Kbd	Append a character to the keyboard queue	A	X	X16
keymap	\$FED2	Kbd	Set or get the current keyboard layout Call address	X Y C	A X Y C	X16
LISTEN	\$FFB1	CPB	Send LISTEN command	A	A X	C64
LKUPLA	\$FF59	ChIO	Search tables for given LA			C128
LKUPSA	\$FF5C	ChIO	Search tables for given SA			C128
LOAD	\$FFD5	ChIO	Load a file into main memory or VRAM	A X Y	A X Y	C64
MACPTR	\$FF44	CPB	Read multiple bytes from the peripheral bus	A X Y C	A X Y P	X16
MCIOUT	\$FEB1	CPB	Write multiple bytes to the peripheral bus	A X Y C	A X Y P	X16
MEMBOT	\$FF9C	Mem	Get address of start of usable RAM			C64
memory_copy	\$FEE7	Mem	Copy a memory region to a different region	r0 r1 r2	r2 A X Y P	X16
memory_crc	\$FEEA	Mem	Calculate the CRC16 of a memory region	r0 r1	r2 A X Y P	X16
memory_decompress	\$FEED	Mem	Decompress an LZSA2 block	r0 r1	r1 A X Y P	X16
memory_fill	\$FEE4	Mem	Fill a memory region with a byte value	A r0 r1	r1 X Y P	X16

MEMTOP	\$FF99	Mem	Get address of end of usable RAM		A X Y	C64
monitor	\$FECC	Misc	Enter machine language monitor	none	A X Y P	X16
mouse_config	\$FF68	Mouse	Configure mouse pointer	A X Y	A X Y P	X16
mouse_get	\$FF6B	Mouse	Get saved mouse sate	X	A (X) P	X16
mouse_scan	\$FF71	Mouse	Poll mouse state and save it	none	A X Y P	X16
OPEN	\$FFC0	ChIO	Open a channel/file.		A X Y	C64
PFKEY 	\$FF65	Kbd	Program a function key <i>[not yet implemented]</i>			C128
PLOT	\$FFF0	Video	Read/write cursor position	A X Y	A X Y	C64
PRIMM	\$FF7D	Misc	Print string following the caller's code			C128
RDTIM	\$FFDE	Time	Read system clock		A X Y	C64
READST	\$FFB7	ChIO	Return status byte		A	C64
SAVE	\$FFD8	ChIO	Save a file from memory	A X Y	A X Y C	C64
SCNKEY	\$FF9F	Kbd	Scan the keyboard	none	A X Y P	C64
SCREEN	\$FFED	Video	Get the screen resolution		X Y	C64
screen_mode	\$FF5F	Video	Get/set screen mode	A C	A X Y P	X16
screen_set_charset	\$FF62	Video	Activate 8x8 text mode charset	A X Y	A X Y P	X16
SECOND	\$FF93	CPB	Send LISTEN secondary address	A	A	C64
SETLFS	\$FFBA	ChIO	Set file parameters (LA, FA, and SA).	A X Y		C64
SETMSG	\$FF90	ChIO	Set verbosity	A		C64
SETNAM	\$FFBD	ChIO	Set file name.	A X Y		C64
SETTIM	\$FFDB	Time	Write system clock	A X Y	A X Y	C64
SETTMO	\$FFA2	CPB	Set timeout			C64
sprite_set_image †	\$FEF0	Video	Set the image of a sprite	r0 r1 r2L A X Y C	A P	X16
sprite_set_position	\$FEF3	Video	Set the position of a sprite	r0 r1 A	A X P	X16
stash	\$FF77	Mem	Write a byte to any RAM bank	stavec A X Y	(stavec) X P	X16

STOP	\$FFE1	Kbd	Test for STOP key		A X P	C64
TALK	\$FFB4	CPB	Send TALK command	A	A	C64
TKSA	\$FF96	CPB	Send TALK secondary address	A	A	C64
UDTIM	\$FFEA	Time	Increment the jiffies clock		A X	C64
UNLSN	\$FFAE	CPB	Send UNLISTEN command		A	C64
UNTLK	\$FFAB	CPB	Send UNTALK command		A	C64

⊘ = Currently unimplemented

† = Partially implemented

Some notes:

- For device #8, the Commodore Peripheral Bus calls first talk to the "Computer DOS" built into the ROM to detect an SD card, before falling back to the Commodore Serial Bus.
- The `IOWBASE` call returns \$9F00, the location of the first VIA controller.
- The `SETTMO` call has been a no-op since the Commodore VIC-20, and has no function on the X16 either.
- The `MEMTOP` call additionally returns the number of available RAM banks in the `.A` register.
- The layout of the zero page (\$0000-\$00FF) and the KERNAL/BASIC variable space (\$0200+) are generally **not** compatible with the C64.

The KERNAL vectors (\$0314-\$0333) are fully compatible with the C64:

\$0314-\$0315: `CINV` - IRQ Interrupt Routine
 \$0316-\$0317: `CBINV` - BRK Instruction Interrupt
 \$0318-\$0319: `NMINV` - Non-Maskable Interrupt
 \$031A-\$031B: `IOPEN` - Kernal OPEN Routine
 \$031C-\$031D: `ICLOSE` - Kernal CLOSE Routine
 \$031E-\$031F: `ICKIN` - Kernal CHKIN Routine
 \$0320-\$0321: `ICKOUT` - Kernal CKOUT Routine
 \$0322-\$0323: `ICLRCH` - Kernal CLRCHN Routine
 \$0324-\$0325: `IBASIN` - Kernal CHRIN Routine
 \$0326-\$0327: `IBSOUT` - Kernal CHROUT Routine
 \$0328-\$0329: `ISTOP` - Kernal STOP Routine
 \$032A-\$032B: `IGETIN` - Kernal GETIN Routine
 \$032C-\$032D: `ICLALL` - Kernal CLALL Routine
 \$0330-\$0331: `ILOAD` - Kernal LOAD Routine
 \$0332-\$0333: `ISAVE` - Kernal SAVE Routine

Commodore Peripheral Bus

The X16 adds two new functions for dealing with the Commodore Peripheral Bus ("IEEE"):

\$FEB1: `MCIOUT` - write multiple bytes to peripheral bus
 \$FF44: `MACPTR` - read multiple bytes from peripheral bus

Function Name: `ACPTR`

Purpose: Read a byte from the peripheral bus

Call address: \$FFA5

Communication registers: `.A`

Preparatory routines: `SETNAM` , `SETLFS` , `OPEN` , `CHKIN`

Error returns: None

Registers affected: .A, .X, .Y, .P

Description: This routine gets a byte of data off the peripheral bus. The data is returned in the accumulator. Errors are returned in the status word which can be read via the `READST` API call.

Function Name: MACPTR

Purpose: Read multiple bytes from the peripheral bus

Call address: \$FF44

Communication registers: .A, .X, .Y, .C

Preparatory routines: `SETNAM` , `SETLFS` , `OPEN` , `CHKIN`

Error returns: None

Registers affected: .A, .X, .Y

Description: The routine `MACPTR` is the multi-byte variant of the `ACPTR` KERNAL routine. Instead of returning a single byte in .A, it can read multiple bytes in one call and write them directly to memory.

The number of bytes to be read is passed in the .A register; a value of 0 indicates that it is up to the KERNAL to decide how many bytes to read. A pointer to where the data is supposed to be written is passed in the .X (lo) and .Y (hi) registers. If carry flag is clear, the destination address will advance with each byte read. If the carry flag is set, the destination address will not advance as data is read. This is useful for reading data directly into VRAM, PCM FIFO, etc.

For reading into Hi RAM, you must set the desired bank prior to calling `MACPTR` . During the read, `MACPTR` will automatically wrap to the next bank as required, leaving the new bank active when finished.

Upon return, a set .C flag indicates that the device or file does not support `MACPTR` , and the program needs to read the data byte-by-byte using the `ACPTR` call instead.

If `MACPTR` is supported, .C is clear and .X (lo) and .Y (hi) contain the number of bytes read.

Like with `ACPTR` , the status of the operation can be retrieved using the `READST` KERNAL call.

Function Name: MCIOUT

Purpose: Write multiple bytes to the peripheral bus

Call address: \$FEB1

Communication registers: .A, .X, .Y, .C

Preparatory routines: `SETNAM` , `SETLFS` , `OPEN` , `CHKOUT`

Error returns: None

Registers affected: .A, .X, .Y

Description: The routine `MCIOUT` is the multi-byte variant of the `CIOUT` KERNAL routine. Instead of writing a single byte, it can write multiple bytes from memory in one call.

The number of bytes to be written is passed in the .A register; a value of 0 indicates 256 bytes. A pointer to the data to be read from is passed in the .X (lo) and .Y (hi) registers. If carry flag is clear, the source address will advance with each byte read out. If the carry flag is set, the source address will not advance as data is read out. This is useful for saving data directly from VRAM.

For reading from Hi RAM, you must set the desired bank prior to calling `MCIOUT` . During the operation, `MCIOUT` will automatically wrap to the next bank as required, leaving the new bank active when finished.

Upon return, a set .C flag indicates that the device or file does not support `MCIOUT` , and the program needs to write the data byte-by-byte using the `CIOUT` call instead.

If `MCIOUT` is supported, .C is clear and .X (lo) and .Y (hi) contain the number of bytes written.

Like with `CIOUT`, the status of the operation can be retrieved using the `READST` KERNAL call. If an error occurred, `READST` should return nonzero.

Channel I/O

Function Name: `BSAVE`

Purpose: Save an area of memory to a file without writing an address header.

Call Address: `$FEBA`

Communication Registers: `.A`, `.X`, `.Y`

Preparatory routines: `SETNAM`, `SETLFS`

Error returns: `.C = 0` if no error, `.C = 1` in case of error and `A` will contain kernel error code

Registers affected: `.A`, `.X`, `.Y`, `.C`

Description: Save the contents of a memory range to a file. Unlike `SAVE`, this call does not write the start address to the beginning of the output file.

`SETLFS` and `SETNAM` must be called beforehand.

`A` is address of zero page pointer to the start address,

`X` and `Y` contain the *exclusive* end address to save. That is, these should contain the address immediately after the final byte: `X` = low byte, `Y` = high byte.

Upon return, if `C` is clear, there were no errors. `C` being set indicates an error in which case `A` will have the error number.

Function Name: `CLOSE`

Purpose: Close a logical file

Call address: `$FFC3`

Communication registers: `.A`

Preparatory routines: None

Error returns: None

Registers affected: `.A`, `.X`, `.Y`, `.P`

Description: `CLOSE` releases resources associated with a logical file number. If the associated device is a serial device on the IEC bus or is a simulated serial device such as CMDR-DOS backed by the X16 SD card, and the file was opened with a secondary address, a close command is sent to the device or to CMDR-DOS.

Function Name: `LOAD`

Purpose: Load the contents of a file from disk to memory

Call address: `$FFD5`

Communication registers: `.A`, `.X`, `.Y`

Preparatory routines: `SETNAM`, `SETLFS`

Error returns: None

Registers affected: `.A`, `.X`, `.Y`, `.P`

Description: Loads a file from disk to memory.

The behavior of `LOAD` can be modified by parameters passed to prior call to `SETLFS`. In particular, the `.Y` register, which usually denotes the *secondary address*, has a specific meaning as follows:

- `.Y = 0`: load to the address given in `.X/.Y` to the `LOAD` call, skipping the first two bytes of the file. (like `LOAD "FILE", 8` in BASIC)
- `.Y = 1`: load to the address given by the first two bytes of the file. The address in `.X/.Y` is ignored. (like `LOAD "FILE", 8, 1` in BASIC)
- `.Y = 2`: load the entire file to the address given in `.X/.Y` to the `LOAD` call. This is also known as a *headerless* load. (like `BLOAD "FILE", 8, 1, $A000` in BASIC)

For the `LOAD` call itself, `.X` and `.Y` is the memory address to load the file into. `.A` controls where the file is to be loaded. On the X16, `LOAD` has an additional feature to load the contents of a file directly into VRAM.

- If the A register is zero, the kernal loads into system memory.
- If the A register is 1, the kernal performs a verify.
- If the A register is 2, the kernal loads into VRAM, starting from \$00000 + the specified starting address.
- If the A register is 3, the kernal loads into VRAM, starting from \$10000 + the specified starting address.

(On the C64, if A is greater than or equal to 1, the kernal performs a verify)

For loads into the banked RAM area. The current RAM bank (in location `$00`) is used as the start point for the load along with the supplied address. If the load is large enough to advance to the end of banked RAM (`$BFFF`), the RAM bank is automatically advanced, and the load continues into the next bank starting at `$A000` .

After the load, if `.C` is set, an error occurred and `.A` will contain the error code. If `.C` is clear, `.X/.Y` will point to the address of final byte loaded + 1.

Note: One does not need to call `CLOSE` after `LOAD` .

Function Name: `OPEN`

Purpose: Opens a channel/file

Call address: `$FFC0`

Communication registers: None

Preparatory routines: `SETNAM`, `SETLFS`

Error returns: None

Registers affected: `.A`, `.X`, `.Y`

Description: Opens a file or channel.

The most common pattern is to then redirect the standard input or output to the file using `CHKIN` or `CHKOUT` respectively. Afterwards, I/O from or to the file or channel is done using `BASIN` (`CHRIN`) and `BSOUT` (`CHROUT`) respectively.

For file I/O, the lower level calls `ACPTR` and `MACPTR` can be used in place of `CHRIN` , since `CHKIN` does the low-level setup for this. Likewise `CIOUT` and `MCIOUT` can be used after `CHKOUT` for the same reason.

Function Name: `SAVE`

Purpose: Save an area of memory to a file.

Call Address: `$FFD8`

Communication Registers: `.A`, `.X`, `.Y`

Preparatory routines: `SETNAM`, `SETLFS`

Error returns: `.C` = 0 if no error, `.C` = 1 in case of error and `A` will contain kernel error code

Registers affected: `.A`, `.X`, `.Y`, `.C`

Description: Save the contents of a memory range to a file. The (little-endian) start address is written to the file as the first two bytes of output, followed by the requested data.

`SETLFS` and `SETNAM` must be called beforehand.

`A` is address of zero page pointer to the start address,

`X` and `Y` contain the *exclusive* end address to save. That is, these should contain the address immediately after the final byte: `X` = low byte, `Y` = high byte.

Upon return, if `C` is clear, there were no errors. `C` being set indicates an error in which case `A` will have the error number.

Function Name: `SETLFS`

Purpose: Set file parameters

Call Address: `$FFBA`

Communication Registers: .A, .X, .Y

Preparatory routines: none

Error returns: None

Registers affected: .A, .X, .Y

Description: Set file parameters typically after calling SETNAM

A is the logical file number, X is the device number, and Y is the secondary address.

Since multiple files can be open (with some exceptions), the value of A specifies the file number. If only one file is being opened at a time, \$01 can be used.

The device number corresponds to the hardware device where the file lives. On the X16, \$08 would be the SD card.

The secondary address has some special meanings:

When used with `OPEN` on disk/storage devices, the following applies:

- 0 = Load (open for read)
- 1 = Save (open for write)
- 2-14 = Read mode, by default. Write, Append, and Modify modes can be specified in the SETNAM filename string as the third argument, e.g. "FILE.DAT,S,w" for write mode. The command channel POSITION command "P" is available in any mode for seeking if the device is CMDR-DOS or HostFS.
- 15 = Command Channel (for sending special commands to CMDR-DOS/HostFS or the disk device)

When used with `LOAD` the following applies:

- 0 = Load the data to address specified in the X and Y register of the LOAD call, regardless of the address header. The two-byte header itself is not loaded into RAM.
- 1 = Load to the address specified in the file's header. The two-byte header itself is not loaded into RAM.
- 2 = Load the data to address specified in the X and Y register of the LOAD call. The entire file is loaded ("headerless").

For more information see [Chapter 11: Working with CMDR-DOS](#)

Function Name: SETNAM

Purpose: Set file name

Call Address: \$FFBD

Communication Registers: .A, .X, .Y

Preparatory routines: none

Error returns: None

Registers affected: .A, .X, .Y

Description: Inform the kernal the name of the file that is to later be opened. A is filename length, X is low byte of filename pointer, Y is high byte of filename pointer.

For example:

```
lda #$08
ldx #<filename
ldy #>filename
jsr SETNAM
```

SETLFS and SETNAM both need to be called prior other file comamnds, such as `OPEN` or `SAVE` .

Memory

\$FEE4: `memory_fill` - fill memory region with a byte value
 \$FEE7: `memory_copy` - copy memory region
 \$FEEA: `memory_crc` - calculate CRC16 of memory region
 \$FEED: `memory_decompress` - decompress LZSA2 block
 \$FF74: `fetch` - read a byte from any RAM or ROM bank
 \$FF77: `stash` - write a byte to any RAM bank

Function Name: `memory_fill`

Signature: `void memory_fill(word address: r0, word num_bytes: r1, byte value: .a);`

Purpose: Fill a memory region with a byte value.

Call address: \$FEE4

Description: This function fills the memory region specified by an address (r0) and a size in bytes (r1) with the constant byte value passed in .A. r0 and .A are preserved, r1 is destroyed.

If the target address is in the \$9F00-\$9FFF range, all bytes will be written to the same address (r0), i.e. the address will not be incremented. This is useful for filling VERA memory (\$9F23 or \$9F24), for example.

Function Name: `memory_copy`

Signature: `void memory_copy(word source: r0, word target: r1, word num_bytes: r2);`

Purpose: Copy a memory region to a different region.

Call address: \$FEE7

Description: This function copies one memory region specified by an address (r0) and a size in bytes (r2) to a different region specified by its start address (r1). The two regions may overlap. r0 and r1 are preserved, r2 is destroyed.

Like with `memory_fill`, source and destination addresses in the \$9F00-\$9FFF range will not be incremented during the copy. This allows, for instance, uploading data from RAM to VERA (destination of \$9F23 or \$9F24), downloading data from VERA (source \$9F23 or \$9F24) or copying data inside VERA (source \$9F23, destination \$9F24). This functionality can also be used to upload, download or transfer data with other I/O devices that have an 8 bit data port.

Function Name: `memory_crc`

Signature: `(word result: r2) memory_crc(word address: r0, word num_bytes: r1);`

Purpose: Calculate the CRC16 of a memory region.

Call address: \$FEEA

Description: This function calculates the CRC16 checksum of the memory region specified by an address (r0) and a size in bytes (r1). The result is returned in r2. r0 is preserved, r1 is destroyed.

Like `memory_fill`, this function does not increment the address if it is in the range of \$9F00-\$9FFF, which allows checksumming VERA memory or data streamed from any other I/O device.

Function Name: `memory_decompress`

Signature: `void memory_decompress(word input: r0, inout word output: r1);`

Purpose: Decompress an LZSA2 block

Call address: \$FEED

Description: This function decompresses an LZSA2-compressed data block from the location passed in r0 and outputs the decompressed data at the location passed in r1. After the call, r1 will be updated with the location of the last output byte plus one.

If the target address is in the \$9F00-\$9FFF range, all bytes will be written to the same address (r0), i.e. the address will not be incremented. This is useful for decompressing directly into VERA memory (\$9F23 or \$9F24), for example. Note that decompressing *from* I/O is not supported.

Notes:

- To create compressed data, use the `lzsa` tool [^1](#) like this: `lzsa -r -f2 <original_file> <compressed_file>`
- This function cannot be used to decompress data in-place, as the output data would overwrite the input data before it is consumed. Therefore, make sure to load the input data to a different location.
- It is possible to have the input data stored in banked RAM, with the obvious 8 KB size restriction.

Function Name: fetch

Purpose: Read a byte from any RAM or ROM bank

Call address: \$FF74

Communication registers: .A, .X, .Y, .P

Description: This function performs an `LDA (ZP),Y` from any RAM or ROM bank. The the zero page address containing the base address is passed in .A, the bank in .X and the offset from the vector in .Y. The data byte is returned in .A. The flags are set according to .A, .X is destroyed, but .Y is preserved.

Function Name: stash

Purpose: Write a byte to any RAM bank

Call address: \$FF77

Communication registers: .A, .X, .Y

Description: This function performs an `STA (ZP),Y` to any RAM bank. The the zero page address containing the base address is passed in `stavec` (\$03B2), the bank in .X and the offset from the vector in .Y. After the call, .X is destroyed, but .A and .Y are preserved.

Clock

\$FF4D: `clock_set_date_time` - set date and time

\$FF50: `clock_get_date_time` - get date and time

Function Name: clock_set_date_time

Purpose: Set the date and time

Call address: \$FF4D

Communication registers: r0, r1, r2, r3L

Preparatory routines: None

Error returns: None

Stack requirements: 0

Registers affected: .A, .X, .Y

Description: The routine `clock_set_date_time` sets the system's real-time-clock.

Register	Contents
r0L	year (1900-based)
r0H	month (1-12)
r1L	day (1-31)
r1H	hours (0-23)
r2L	minutes (0-59)
r2H	seconds (0-59)

r3L	jiffies (0-59)
r3H	weekday (0-6)

Jiffies are 1/60th seconds.

Function Name: `clock_get_date_time`

Purpose: Get the date and time
 Call address: \$FF50
 Communication registers: r0, r1, r2, r3L
 Preparatory routines: None
 Error returns: None
 Stack requirements: 0
 Registers affected: .A, .X, .Y

Description: The routine `clock_get_date_time` returns the state of the system's real-time-clock. The register assignment is identical to `clock_set_date_time`.

On the Commander X16, the *jiffies* field is unsupported and will always read back as 0.

Keyboard

\$FEBD: `kdbuf_peek` - get first char in keyboard queue and queue length
 \$FEC0: `kdbuf_get_modifiers` - get currently pressed modifiers
 \$FEC3: `kdbuf_put` - append a char to the keyboard queue
 \$FED2: `keymap` - set or get the current keyboard layout

Function Name: `kdbuf_peek`

Purpose: Get next char and keyboard queue length
 Call address: \$FEBD
 Communication registers: .A, .X
 Preparatory routines: None
 Error returns: None
 Stack requirements: 0
 Registers affected: -

Description: The routine `kdbuf_peek` returns the next character in the keyboard queue in .A, without removing it from the queue, and the current length of the queue in .X. If .X is 0, the Z flag will be set, and the value of .A is undefined.

Function Name: `kdbuf_get_modifiers`

Purpose: Get currently pressed modifiers
 Call address: \$FEC0
 Communication registers: .A
 Preparatory routines: None
 Error returns: None
 Stack requirements: 0
 Registers affected: -

Description: The routine `kdbuf_get_modifiers` returns a bitmask that represents the currently pressed modifier keys in .A:

Bit	Value	Description	Comment
-----	-------	-------------	---------

0	1	Shift	
1	2	Alt	C64: Commodore
2	4	Control	
3	8	Logo/Windows	C128: Alt
4	16	Caps	

This allows detecting combinations of a regular key and a modifier key in cases where there is no dedicated PETSCII code for the combination, e.g. Ctrl+Esc or Alt+F1.

Function Name: **kbdbuf_put**

Purpose: Append a char to the keyboard queue

Call address: \$FEC3

Communication registers: .A

Preparatory routines: None

Error returns: None

Stack requirements: 0

Registers affected: .X

Description: The routine `kbdbuf_put` appends the char in .A to the keyboard queue.

Function Name: **keymap**

Purpose: Set or get the current keyboard layout Call address: \$FED2

Communication registers: .X , .Y Preparatory routines: None

Error returns: .C = 1 in case of error Stack requirements: 0

Registers affected: -

Description: If .C is set, the routine `keymap` returns a pointer to a zero-terminated string with the current keyboard layout identifier in .X/.Y. If .C is clear, it sets the keyboard layout to the zero-terminated identifier pointed to by .X/.Y. On return, .C is set in case the keyboard layout is unsupported.

Keyboard layout identifiers are in the form "DE", "DE-CH" etc.

Function Name: **SCNKEY**

Purpose: Poll the SMC for a keystroke, and add it to the X16's buffer.

Call address: \$FF9F

Communication registers: None Preparatory routines: None

Error returns: None Stack requirements: 0

Registers affected: .A, .X., .Y

Description:

This routine is called by the default KERNAL interrupt service routine in order to poll a keystroke from the System Management Controller and place it in the KERNAL's keyboard buffer. Unless the KERNAL ISR is being bypassed or supplemented, it is not normally necessary to call this routine from user code.

This routine is also called `kbd_scan` inside KERNAL code.

Mouse

\$FF68: `mouse_config` - configure mouse pointer

\$FF71: `mouse_scan` - query mouse

\$FF6B: `mouse_get` - get state of mouse

Function Name: `mouse_config`

Purpose: Configure the mouse pointer

Call address: \$FF68

Communication registers: .A, .X, .Y

Preparatory routines: None

Error returns: None

Stack requirements: 0

Registers affected: .A, .X, .Y

Description: The routine `mouse_config` configures the mouse pointer.

The argument in .A specifies whether the mouse pointer should be visible or not, and what shape it should have. For a list of possible values, see the basic statement `MOUSE`.

The arguments in .X and .Y specify the screen resolution in 8 pixel increments. The values .X = 0 and .Y = 0 keep the current resolution.

EXAMPLE:

```
SEC JSR screen_mode ; get current screen size (in 8px) into .X and .Y
LDA #1 JSR mouse_config ; show the default mouse pointer
```

Function Name: `mouse_scan`

Purpose: Query the mouse and save its state

Call address: \$FF71

Communication registers: None

Preparatory routines: None

Error returns: None

Stack requirements: ?

Registers affected: .A, .X, .Y

Description: The routine `mouse_scan` retrieves all state from the mouse and saves it. It can then be retrieved using `mouse_get`. The default interrupt handler already takes care of this, so this routine should only be called if the interrupt handler has been completely replaced.

Function Name: `mouse_get`

Purpose: Get the mouse state

Call address: \$FF6B

Communication registers: .X

Preparatory routines: `mouse_config`

Error returns: None

Stack requirements: 0

Registers affected: .A, .X

Description: The routine `mouse_get` returns the state of the mouse. The caller passes the offset of a zero-page location in .X, which the routine will populate with the mouse position in 4 consecutive bytes:

Offset	Size	Description
0	2	X Position
2	2	Y Position

The state of the mouse buttons is returned in the .A register:

Bit	Description
0	Left Button
1	Right Button
2	Middle Button
3	Unused
4	Button 4
5	Button 5

If a button is pressed, the corresponding bit is set. Buttons 4 and 5 are extended buttons not supported by all mice.

If available, the movement of the scroll wheel since the last call to this function is returned in the .X register as an 8-bit signed value. Moving the scroll wheel away from the user is represented by a negative value, and moving it towards the user is represented by a positive value. If the connected mouse has no scroll wheel, the value 0 is returned in the .X register.

EXAMPLE:

```
LDX #$70
JSR mouse_get ; get mouse position in $70/$71 (X) and $72/$73 (Y)
AND #1
BNE BUTTON_PRESSED
```

Joystick

\$FF53: joystick_scan - query joysticks

\$FF56: joystick_get - get state of one joystick

Function Name: joystick_scan

Purpose: Query the joysticks and save their state

Call address: \$FF53

Communication registers: None

Preparatory routines: None

Error returns: None

Stack requirements: 0

Registers affected: .A, .X, .Y

Description: The routine `joystick_scan` retrieves all state from the four joysticks and saves it. It can then be retrieved using `joystick_get`. The default interrupt handler already takes care of this, so this routine should only be called if the interrupt handler has been completely replaced.

Function Name: joystick_get

Purpose: Get the state of one of the joysticks

Call address: \$FF56

Communication registers: .A

Preparatory routines: `joystick_scan`

Error returns: None

Stack requirements: 0

Registers affected: .A, .X, .Y

Description: The routine `joystick_get` retrieves all state from one of the joysticks. The number of the joystick is passed in .A (0 for the keyboard joystick and 1 through 4 for SNES controllers), and the state is returned in .A, .X and .Y.

```
.A, byte 0:      | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
                SNES | B | Y | SEL | STA | UP | DN | LT | RT |

.X, byte 1:      | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
                SNES | A | X | L | R | 1 | 1 | 1 | 1 |

.Y, byte 2:
    $00 = joystick present
    $FF = joystick not present
```

If a button is pressed, the corresponding bit is zero.

(With a dedicated handler, the API can also be used for other devices with an SNES controller connector. The data returned in .A/.X/Y is just the raw 24 bits returned by the device.)

The keyboard joystick uses the standard SNES9X/ZSNES mapping:

SNES Button	Keyboard Key	Alt. Keyboard Key
A	X	Left Ctrl
B	Z	Left Alt
X	S	
Y	A	
L	D	
R	C	
START	Enter	
SELECT	Left Shift	
D-Pad	Cursor Keys	

Note that the keyboard joystick will allow LEFT and RIGHT as well as UP and DOWN to be pressed at the same time, while controllers usually prevent this mechanically.

How to Use:

If the default interrupt handler is used:

1. Call this routine.

If the default interrupt handler is disabled or replaced:

1. Call `joystick_scan` to have the system query the joysticks.
2. Call this routine.

EXAMPLE:

```
JSR joystick_scan
LDA #0
```



```

JSR joystick_get
TXA
AND #128
BEQ A_PRESSED

```

I2C

\$FEB4: `i2c_batch_read` - read multiple bytes from an I2C device

\$FEB7: `i2c_batch_write` - write multiple bytes to an I2C device

\$FEC6: `i2c_read_byte` - read a byte from an I2C device

\$FEC9: `i2c_write_byte` - write a byte to an I2C device

Function Name: `i2c_batch_read`

Purpose: Read bytes from a given I2C device into a RAM location

Call address: \$FEB4

Communication registers: .X, r0, r1, .C

Preparatory routines: None

Error returns: .C = 1 in case of error

Registers affected: .A .Y .P

Description: The routine `i2c_batch_read` reads a fixed number of bytes from an I2C device into RAM. To call, put I2C device (address) in .X, the pointer to the RAM location to which to place the data into r0, and the number of bytes to read into r1. If carry is set, the RAM location isn't advanced. This might be useful if you're reading from an I2C device and writing directly into VRAM.

If the routine encountered an error, carry will be set upon return.

EXAMPLE:

```

ldx #$50 ; One of the cartridge I2C flash devices
lda #<$0400
sta r0
lda #>$0400
sta r0+1
lda #<500
sta r1
lda #>500
sta r1+1
clc
jsr i2c_batch_read ; read 500 bytes from I2C device $50 into RAM starting at $0400

```

Function Name: `i2c_batch_write`

Purpose: Write bytes to a given I2C device with data in RAM

Call address: \$FEB7

Communication registers: .X, r0, r1, r2, .C

Preparatory routines: None

Error returns: .C = 1 in case of error

Registers affected: .A .Y .P r2

Description: The routine `i2c_batch_write` writes a fixed number of bytes from RAM to an I2C device. To call, put I2C device (address) in .X, the pointer to the RAM location from which to read into r0, and the number of bytes to write into r1. If carry is set, the RAM location isn't advanced. This might be useful if you're reading from an I/O device and writing that data to an I2C device.

The number of bytes written is returned in r2. If the routine encountered an error, carry will be set upon return.

EXAMPLE:

```
ldx #$50 ; One of the cartridge I2C flash devices
lda #<$0400
sta r0
lda #>$0400
sta r0+1
lda #<500
sta r1
lda #>500
sta r1+1
clc
jsr i2c_batch_write ; write 500 bytes to I2C device $50 from RAM
                    ; starting at $0400
                    ; for this example, the first two bytes in
                    ; the $0400 buffer would be the target address
                    ; in the I2C flash. This, of course, varies
                    ; between various I2C device types.
```

Function Name: i2c_read_byte

Purpose: Read a byte at a given offset from a given I2C device

Call address: \$FEC6

Communication registers: .A, .X, .Y

Preparatory routines: None

Error returns: .C = 1 in case of error

Registers affected: .A

Description: The routine `i2c_read_byte` reads a single byte at offset .Y from I2C device .X and returns the result in .A. .C is 0 if the read was successful, and 1 if no such device exists.

EXAMPLE:

```
LDX #$6F ; RTC device
LDY #$20 ; start of NVRAM inside RTC
JSR i2c_read_byte ; read first byte of NVRAM
```

Function Name: i2c_write_byte

Purpose: Write a byte at a given offset to a given I2C device

Call address: \$FEC9

Communication registers: .A, .X, .Y

Preparatory routines: None

Error returns: .C = 1 in case of error

Registers affected: .A

Description: The routine `i2c_write_byte` writes the byte in .A at offset .Y of I2C device .X. .C is 0 if the write was successful, and 1 if no such device exists.

EXAMPLES:

```
LDX #$6F ; RTC device
LDY #$20 ; start of NVRAM inside RTC
```

```
LDA #'X'
JSR i2c_write_byte ; write first byte of NVRAM
```

```
LDX #$42 ; System Management Controller
LDY #$01 ; magic location for system poweroff
LDA #$00 ; magic value for system poweroff
JSR i2c_write_byte ; power off the system
```

```
; Reset system at the end of your program
LDX #$42 ; System Management Controller
LDY #$02 ; magic location for system reset
LDA #$00 ; magic value for system poweroff
JSR $FEC9 ; power off the system
```

Sprites

\$FEF0: `sprite_set_image` - set the image of a sprite

\$FEF3: `sprite_set_position` - set the position of a sprite

Function Name: `sprite_set_image`

Purpose: Set the image of a sprite

Call address: \$FEF0

Signature: `bool sprite_set_image(byte number: .a, width: .x, height: .y, apply_mask: .c, word pixels: r0, word mask: r1, byte bpp: r2L);`

Error returns: `.C = 1` in case of error

Description: This function sets the image of a sprite. The number of the sprite is given in `.A`, The bits per pixel (bpp) in `r2L`, and the width and height in `.X` and `.Y`. The pixel data at `r0` is interpreted accordingly and converted into the graphics hardware's native format. If the `.C` flag is set, the transparency mask pointed to by `r1` is applied during the conversion. The function returns `.C = 0` if converting the data was successful, and `.C = 1` otherwise. Note that this does not change the visibility of the sprite.

Note: There are certain limitations on the possible values of width, height, bpp and `apply_mask`:

- width and height may not exceed the hardware's capabilities.
 - Legal values for bpp are 1, 4 and 8. If the hardware only supports lower depths, the image data is converted down.
 - `apply_mask` is only valid for 1 bpp data.
-

Function Name: `sprite_set_position`

Purpose: Set the position of a sprite or hide it.

Call address: \$FEF3

Signature: `void sprite_set_position(byte number: .a, word x: r0, word y: r1);`

Error returns: None

Description: This function shows a given sprite (`.A`) at a certain position or hides it. The position is passed in `r0` and `r1`. If the x position is negative (`>$8000`), the sprite will be hidden.

Note: This routine only supports setting the position for sprite numbers 0-31.

Framebuffer

The framebuffer API is a low-level graphics API that completely abstracts the framebuffer by exposing a minimal set of high-performance functions. It is useful as an abstraction and as a convenience library for applications that need high

performance framebuffer access.

```
$FEF6: `FB_init` - enable graphics mode
$FEF9: `FB_get_info` - get screen size and color depth
$FEFC: `FB_set_palette` - set (parts of) the palette
$FEFF: `FB_cursor_position` - position the direct-access cursor
$FF02: `FB_cursor_next_line` - move direct-access cursor to next line
$FF05: `FB_get_pixel` - read one pixel, update cursor
$FF08: `FB_get_pixels` - copy pixels into RAM, update cursor
$FF0B: `FB_set_pixel` - set one pixel, update cursor
$FF0E: `FB_set_pixels` - copy pixels from RAM, update cursor
$FF11: `FB_set_8_pixels` - set 8 pixels from bit mask (transparent), update cursor
$FF14: `FB_set_8_pixels_opaque` - set 8 pixels from bit mask (opaque), update cursor
$FF17: `FB_fill_pixels` - fill pixels with constant color, update cursor
$FF1A: `FB_filter_pixels` - apply transform to pixels, update cursor
$FF1D: `FB_move_pixels` - copy horizontally consecutive pixels to a different position
```

All calls are vectored, which allows installing a replacement framebuffer driver.

```
$02E4: I_FB_init
$02E6: I_FB_get_info
$02E8: I_FB_set_palette
$02EA: I_FB_cursor_position
$02EC: I_FB_cursor_next_line
$02EE: I_FB_get_pixel
$02F0: I_FB_get_pixels
$02F2: I_FB_set_pixel
$02F4: I_FB_set_pixels
$02F6: I_FB_set_8_pixels
$02F8: I_FB_set_8_pixels_opaque
$02FA: I_FB_fill_pixels
$02FC: I_FB_filter_pixels
$02FE: I_FB_move_pixels
```

The model of this API is based on the direct-access cursor. In order to read and write pixels, the cursor has to be set to a specific x/y-location, and all subsequent calls will access consecutive pixels at the cursor position and update the cursor.

The default driver supports the VERA framebuffer at a resolution of 320x200 pixels and 256 colors. Using `screen_mode` to set mode \$80 will enable this driver.

Function Name: FB_init

Signature: void FB_init();

Purpose: Enter graphics mode.

Function Name: FB_get_info

Signature: void FB_get_info(out word width: r0, out word height: r1, out byte color_depth: .a);

Purpose: Return the resolution and color depth

Function Name: FB_set_palette

Signature: void FB_set_palette(word pointer: r0, index: .a, color count: .x);

Purpose: Set (parts of) the palette

Description: `FB_set_palette` copies color data from the address pointed to by r0, updates the color in VERA palette RAM starting at the index A, with the length of the update (in words) in X. If X is 0, all 256 colors are copied (512 bytes)

Function Name: FB_cursor_position

Signature: void FB_cursor_position(word x: r0, word y: r1);

Purpose: Position the direct-access cursor

Description: `FB_cursor_position` sets the direct-access cursor to the given screen coordinate. Future operations will access pixels at the cursor location and update the cursor.

Function Name: FB_cursor_next_line

Signature: void FB_cursor_next_line(word x: r0);

Purpose: Move the direct-access cursor to next line

Description: `FB_cursor_next_line` increments the y position of the direct-access cursor, and sets the x position to the same one that was passed to the previous `FB_cursor_position` call. This is useful for drawing rectangular shapes, and faster than explicitly positioning the cursor.

Function Name: FB_get_pixel

Signature: byte FB_get_pixel();

Purpose: Read one pixel, update cursor

Function Name: FB_get_pixels

Signature: void FB_get_pixels(word ptr: r0, word count: r1);

Purpose: Copy pixels into RAM, update cursor

Description: This function copies pixels into an array in RAM. The array consists of one byte per pixel.

Function Name: FB_set_pixel

Signature: void FB_set_pixel(byte color: .a);

Purpose: Set one pixel, update cursor

Function Name: FB_set_pixels

Signature: void FB_set_pixels(word ptr: r0, word count: r1);

Purpose: Copy pixels from RAM, update cursor

Description: This function sets pixels from an array of pixels in RAM. The array consists of one byte per pixel.

Function Name: FB_set_8_pixels

Signature: void FB_set_8_pixels(byte pattern: .a, byte color: .x);

Purpose: Set 8 pixels from bit mask (transparent), update cursor

Description: This function sets all 1-bits of the pattern to a given color and skips a pixel for every 0 bit. The order is MSB to LSB. The cursor will be moved by 8 pixels.

Function Name: FB_set_8_pixels_opaque

Signature: void FB_set_8_pixels_opaque(byte pattern: .a, byte mask: r0L, byte color1: .x, byte color2: .y);

Purpose: Set 8 pixels from bit mask (opaque), update cursor

Description: For every 1-bit in the mask, this function sets the pixel to color1 if the corresponding bit in the pattern is 1, and to color2 otherwise. For every 0-bit in the mask, it skips a pixel. The order is MSB to LSB. The cursor will be moved by 8 pixels.

Function Name: FB_fill_pixels

Signature: void FB_fill_pixels(word count: r0, word step: r1, byte color: .a);

Purpose: Fill pixels with constant color, update cursor

Description: `FB_fill_pixels` sets pixels with a constant color. The argument `step` specifies the increment between pixels. A value of 0 or 1 will cause consecutive pixels to be set. Passing a `step` value of the screen width will set vertically adjacent pixels going top down. Smaller values allow drawing dotted horizontal lines, and multiples of the screen width allow drawing dotted vertical lines.

Function Name: FB_filter_pixels

Signature: void FB_filter_pixels(word ptr: r0, word count: r1);

Purpose: Apply transform to pixels, update cursor

Description: This function allows modifying consecutive pixels. The function pointer will be called for every pixel, with the color in `.a`, and it needs to return the new color in `.a`.

Function Name: FB_move_pixels

Signature: void FB_move_pixels(word sx: r0, word sy: r1, word tx: r2, word ty: r3, word count: r4);

Purpose: Copy horizontally consecutive pixels to a different position

[Note: Overlapping regions are not yet supported.]

Graphics

The high-level graphics API exposes a set of standard functions. It allows applications to easily perform some common high-level actions like drawing lines, rectangles and images, as well as moving parts of the screen. All commands are completely implemented on top of the framebuffer API, that is, they will continue working after replacing the framebuffer driver with one that supports a different resolution, color depth or even graphics device.

\$FF20: `GRAPH_init` - initialize graphics
 \$FF23: `GRAPH_clear` - clear screen
 \$FF26: `GRAPH_set_window` - set clipping region
 \$FF29: `GRAPH_set_colors` - set stroke, fill and background colors
 \$FF2C: `GRAPH_draw_line` - draw a line
 \$FF2F: `GRAPH_draw_rect` - draw a rectangle (optionally filled)
 \$FF32: `GRAPH_move_rect` - move pixels
 \$FF35: `GRAPH_draw_oval` - draw an oval or circle
 \$FF38: `GRAPH_draw_image` - draw a rectangular image
 \$FF3B: `GRAPH_set_font` - set the current font
 \$FF3E: `GRAPH_get_char_size` - get size and baseline of a character
 \$FF41: `GRAPH_put_char` - print a character

Function Name: GRAPH_init

Signature: void GRAPH_init(word vectors: r0);

Purpose: Activate framebuffer driver, enter and initialize graphics mode

Description: This call activates the framebuffer driver whose vector table is passed in `r0`. If `r0` is 0, the default driver is activated. It then switches the video hardware into graphics mode, sets the window to full screen, initializes the colors and activates the system font.

Function Name: GRAPH_clear

Signature: void GRAPH_clear();

Purpose: Clear the current window with the current background color.

Function Name: GRAPH_set_window

Signature: void GRAPH_set_window(word x: r0, word y: r1, word width: r2, word height: r3);

Purpose: Set the clipping region

Description: All graphics commands are clipped to the window. This function configures the origin and size of the window. All 0 arguments set the window to full screen.

[Note: Only text output and GRAPH_clear currently respect the clipping region.]

Function Name: GRAPH_set_colors

Signature: void GRAPH_set_colors(byte stroke: .a, byte fill: .x, byte background: .y);

Purpose: Set the three colors

Description: This function sets the three colors: The stroke color, the fill color and the background color.

Function Name: GRAPH_draw_line

Signature: void GRAPH_draw_line(word x1: r0, word y1: r1, word x2: r2, word y2: r3);

Purpose: Draw a line using the stroke color

Function Name: GRAPH_draw_rect

Signature: void GRAPH_draw_rect(word x: r0, word y: r1, word width: r2, word height: r3, word corner_radius: r4, bool fill: .c);

Purpose: Draw a rectangle.

Description: This function will draw the frame of a rectangle using the stroke color. If `fill` is `true`, it will also fill the area using the fill color. To only fill a rectangle, set the stroke color to the same value as the fill color.

[Note: The border radius is currently unimplemented.]

Function Name: GRAPH_move_rect

Signature: void GRAPH_move_rect(word sx: r0, word sy: r1, word tx: r2, word ty: r3, word width: r4, word height: r5);

Purpose: Copy a rectangular screen area to a different location

Description: `GRAPH_move_rect` coll copy a rectangular area of the screen to a different location. The two areas may overlap.

[Note: Support for overlapping is not currently implemented.]

Function Name: GRAPH_draw_oval

Signature: void GRAPH_draw_oval(word x: r0, word y: r1, word width: r2, word height: r3, bool fill: .c);

Purpose: Draw an oval or a circle

Description: This function draws an oval filling the given bounding box. If width equals height, the resulting shape is a circle. The oval will be outlined by the stroke color. If `fill` is `true`, it will be filled using the fill color. To only fill an oval, set the stroke color to the same value as the fill color.

Function Name: GRAPH_draw_image

Signature: void GRAPH_draw_image(word x: r0, word y: r1, word ptr: r2, word width: r3, word height: r4);

Purpose: Draw a rectangular image from data in memory

Description: This function copies pixel data from memory onto the screen. The representation of the data in memory has to have one byte per pixel, with the pixels organized line by line top to bottom, and within the line left to right.

Function Name: GRAPH_set_font

Signature: void GRAPH_set_font(void ptr: r0);

Purpose: Set the current font

Description: This function sets the current font to be used for the remaining font-related functions. The argument is a pointer to the font data structure in memory, which must be in the format of a single point size GEOS font (i.e. one GEOS font file VLIR chunk). An argument of 0 will activate the built-in system font.

Function Name: GRAPH_get_char_size

Signature: (byte baseline: .a, byte width: .x, byte height_or_style: .y, bool is_control: .c) GRAPH_get_char_size(byte c: .a, byte format: .x);

Purpose: Get the size and baseline of a character, or interpret a control code

Description: This functionality of GRAPH_get_char_size depends on the type of code that is passed in: For a printable character, this function returns the metrics of the character in a given format. For a control code, it returns the resulting format. In either case, the current format is passed in .x, and the character in .a.

- The format is an opaque byte value whose value should not be relied upon, except for 0, which is plain text.
 - The resulting values are measured in pixels.
 - The baseline is measured from the top.
-

Function Name: GRAPH_put_char

Signature: void GRAPH_put_char(inout word x: r0, inout word y: r1, byte c: .a);

Purpose: Print a character onto the graphics screen

Description: This function prints a single character at a given location on the graphics screen. The location is then updated. The following control codes are supported:

- \$01: SWAP COLORS
- \$04: ATTRIBUTES: UNDERLINE
- \$06: ATTRIBUTES: BOLD
- \$07: BELL
- \$08: BACKSPACE
- \$09: TAB
- \$0A: LF
- \$0B: ATTRIBUTES: ITALICS
- \$0C: ATTRIBUTES: OUTLINE
- \$0D/\$8D: REGULAR/SHIFTED RETURN
- \$11/\$91: CURSOR: DOWN/UP
- \$12: ATTRIBUTES: REVERSE
- \$13/\$93: HOME/CLEAR
- \$14 DEL
- \$92: ATTRIBUTES: CLEAR ALL
- all color codes

Notes:

- CR (\$0D) SHIFT+CR (\$8D) and LF (\$0A) all set the cursor to the beginning of the next line. The only difference is that CR and SHIFT+CR reset the attributes, and LF does not.
- BACKSPACE (\$08) and DEL (\$14) move the cursor to the beginning of the previous character but does not actually clear it. Multiple consecutive BACKSPACE/DEL characters are not supported.

- There is no way to individually disable attributes (underlined, bold, reversed, italics, outline). The only way to disable them is to reset the attributes using code \$92, which switches to plain text.
- All 16 PETSCII color codes are supported. Code \$01 to swap the colors will swap the stroke and fill colors.
- The stroke color is used to draw the characters, and the underline is drawn using the fill color. In reverse text mode, the text background is filled with the fill color.
- *[BELL (\$07), TAB (\$09) and SHIFT+TAB (\$18) are not yet implemented.]*

Console

\$FEDB: `console_init` - initialize console mode
 \$FEDE: `console_put_char` - print character to console
 \$FED8: `console_put_image` - draw image as if it was a character
 \$FEE1: `console_get_char` - get character from console
 \$FED5: `console_set_paging_message` - set paging message or disable paging

The console is a screen mode that allows text output and input in proportional fonts that support the usual styles. It is useful for rich text-based interfaces.

Function Name: `console_init`

Signature: `void console_init(word x: r0, word y: r1, word width: r2, word height: r3);`
 Purpose: Initialize console mode.
 Call address: \$FEDB

Description: This function initializes console mode. It sets up the window (text clipping area) passed into it, clears the window and positions the cursor at the top left. All 0 arguments create a full screen console. You have to switch to graphics mode using `screen_mode` beforehand.

Function Name: `console_put_char`

Signature: `void console_put_char(byte char: .a, bool wrapping: .c);`
 Purpose: Print a character to the console.
 Call address: \$FEDE

Description: This function prints a character to the console. The .C flag specifies whether text should be wrapped at character (.C=0) or word (.C=1) boundaries. In the latter case, characters will be buffered until a SPACE, CR or LF character is sent, so make sure the text that is printed always ends in one of these characters.

Note: If the bottom of the screen is reached, this function will scroll its contents up to make extra room.

Function Name: `console_put_image`

Signature: `void console_put_image(word ptr: r0, word width: r1, word height: r2);`
 Purpose: Draw image as if it was a character.
 Call address: \$FEE1

Description: This function draws an image (in `GRAPH_draw_image` format) at the current cursor position and advances the cursor accordingly. This way, an image can be presented inline. A common example would be an emoji bitmap, but it is also possible to show full-width pictures if you print a newline before and after the image.

Notes:

- If the bottom of the screen is reached, this function will scroll its contents up to make extra room.
- Subsequent line breaks will take the image height into account, so that the new cursor position is below the image.

Function Name: `console_get_char`

Signature: (byte char: .a) console_get_char();

Purpose: Get a character from the console.

Call address: \$FEE1

Description: This function gets a character to the console. It does this by collecting a whole line of character, i.e. until the user presses RETURN. Then, the line will be sent character by character.

This function allows editing the line using BACKSPACE/DEL, but does not allow moving the cursor within the line, write more than one line, or using control codes.

Function Name: console_set_paging_message

Signature: void console_set_paging_message(word message: r0);

Purpose: Set the paging message or disable paging.

Call address: \$FED5

Description: The console can halt printing after a full screen height worth of text has been printed. It will then show a message, wait for any key, and continue printing. This function sets this message. A zero-terminated text is passed in r0. To turn off paging, call this function with r0 = 0 - this is the default.

Note: It is possible to use control codes to change the text style and color. Do not use codes that change the cursor position, like CR or LF. Also, the text must not overflow one line on the screen.

Other

\$FECF: entropy_get - get 24 random bits

\$FECC: monitor - enter machine language monitor

\$FF47: enter_basic - enter BASIC

\$FF5F: screen_mode - get/set screen mode

\$FF62: screen_set_charset - activate 8x8 text mode charset

Function Name: entropy_get

Purpose: Get 24 random bits

Call address: \$FECF

Communication registers: .A, .X, .Y

Preparatory routines: None

Error returns: None

Registers affected: .A, .X, .Y

Description: This routine returns 24 somewhat random bits in registers .A, .X, and .Y. In order to get higher-quality random numbers, this data should be fed into a pseudo-random number generator.

How to Use:

1. Call this routine.

EXAMPLE:

```
; throw a die
again:
    JSR entropy_get
    STX tmp    ; combine 24 bits
    EOR tmp    ; using exclusive-or
    STY tmp    ; to get a higher-quality
    EOR tmp    ; 8 bit random value
    STA tmp
    LSR
```

```

LSR
LSR
LSR      ; combine resulting 8 bits
EOR tmp  ; to get 4 bits
AND #7   ; we're down to values 0-7
CMP #0
BEQ again ; 0 is illegal
CMP #7
BEQ again ; 7 is illegal
ORA #$30 ; convert to ASCII
JMP $FFD2 ; print character

```

Function Name: monitor

Purpose: Enter the machine language monitor

Call address: \$FECC

Communication registers: None

Preparatory routines: None

Error returns: Does not return

Stack requirements: Does not return

Registers affected: Does not return

Description: This routine switches from BASIC to machine language monitor mode. It does not return to the caller. When the user quits the monitor, it will restart BASIC.

How to Use:

1. Call this routine.

EXAMPLE:

```
JMP monitor
```

Function Name: enter_basic

Purpose: Enter BASIC

Call address: \$FF47

Communication registers: .C

Preparatory routines: None

Error returns: Does not return

Description: Call this to enter BASIC mode, either through a cold start (.C=1) or a warm start (.C=0).

EXAMPLE:

```

CLC
JMP enter_basic ; returns to the "READY." prompt

```

Function Name: screen_mode

Purpose: Get/Set the screen mode

Call address: \$FF5F

Communication registers: .A, .X, .Y, .C

Preparatory routines: None

Error returns: .C = 1 in case of error

Stack requirements: 4

Registers affected: .A, .X, .Y

Description: If .C is set, a call to this routine gets the current screen mode in .A, the width (in tiles) of the screen in .X, and the height (in tiles) of the screen in .Y. If .C is clear, it sets the current screen mode to the value in .A. For a list of possible values, see the basic statement `SCREEN`. If the mode is unsupported, .C will be set, otherwise cleared.

EXAMPLE:

```
LDA #$80
CLC
JSR screen_mode ; SET 320x200@256C MODE
BCS FAILURE
```

Function Name: screen_set_charset

Purpose: Activate a 8x8 text mode charset

Call address: \$FF62

Communication registers: .A, .X, .Y

Preparatory routines: None

Stack requirements: [?]

Registers affected: .A, .X, .Y

Description: A call to this routine uploads a character set to the video hardware and activates it. The value of .A decides what charset to upload:

Value	Description
0	use pointer in .X/.Y
1	ISO
2	PET upper/graph
3	PET upper/lower
4	PET upper/graph (thin)
5	PET upper/lower (thin)
6	ISO (thin)

If .A is zero, .X (lo) and .Y (hi) contain a pointer to a 2 KB RAM area that gets uploaded as the new 8x8 character set. The data has to consist of 256 characters of 8 bytes each, top to bottom, with the MSB on the left and set bits representing the foreground color.

EXAMPLE:

```
LDA #0
LDX #<MY_CHARSET
LDY #>MY_CHARSET
JSR screen_set_charset ; UPLOAD CUSTOM CHARSET "MY_CHARSET"
```

Function Name: JSRFAR

Purpose: Execute a routine on another RAM or ROM bank

Call address: \$FF6E

Communication registers: None

Preparatory routines: None

Error returns: None

Stack requirements: 4

Registers affected: None

Description: The routine `JSRFAR` enables code to execute some other code located on a specific RAM or ROM bank. This works independently of which RAM or ROM bank the currently executing code is residing in. The 16 bit address and the 8 bit bank number have to follow the instruction stream. The `JSRFAR` routine will switch both the ROM and the RAM bank to the specified bank and restore it after the routine's `RTS`. Execution resumes after the 3 byte arguments. **Note:** The C128 also has a `JSRFAR` function at \$FF6E, but it is incompatible with the X16 version.

How to Use:

1. Call this routine.

EXAMPLE:

```
JSR JSRFAR
.WORD $C000 ; ADDRESS
.BYTE 1      ; BANK
```

Chapter 5: Math Library

The Commander X16 contains a floating point Math library with a precision of 40 bits, which corresponds to 9 decimal digits. It is a stand-alone derivative of the library contained in Microsoft BASIC. Except for the different base address, it is compatible with the C128 and C65 libraries.

The full documentation of these functions can be found in the book [C128 Developers Package for Commodore 6502 Development](#). The Math Library documentation starts in Chapter 13. (PDF page 257)

The following functions are available from machine language code after setting the ROM bank to 4, which is the default.

Format Conversions

Address	Symbol	Description
\$FE00	AYINT	convert floating point to integer (signed word)
\$FE03	GIVAYF	convert integer (signed word) to floating point
\$FE06	FOUT	convert floating point to ASCII string
\$FE09	VAL_1	convert ASCII string to floating point <i>[not yet implemented]</i>
\$FE0C	GETADR	convert floating point to an address (unsigned word)
\$FE0F	FLOATC	convert address (unsigned word) to floating point

X16 Additions

The following calls are new to the X16 and were not part of the C128 math library API:

Address	Symbol	Description
\$FE87	FLOAT	FAC = (s8).A convert signed byte to float
\$FE8A	FLOATS	FAC = (s16)facho+1:facho
\$FE8D	QINT	facho:facho+1:facho+2:facho+3 = u32(FAC)
\$FE93	FOUTC	Convert FAC to ASCII string at fbuffr - 1 + .Y

Movement

Address	Symbol	Description
\$FE5A	CONUPK	move RAM MEM to ARG
\$FE5D	ROMUPK	move ROM/RAM MEM to ARG (use CONUPK)
\$FE60	MOVFRM	move RAM MEM to FACC (use MOVFM)
\$FE63	MOVFM	move ROM/RAM MEM to FACC
\$FE66	MOVMF	move FACC to RAM MEM
\$FE69	MOVFA	move ARG to FACC

\$FE6C	MOVAF	move FACC to ARG
--------	-------	------------------

X16 Additions

The following calls are new to the X16 and were not part of the C128 math library API:

Address	Symbol	Description
\$FE81	MOVEF	ARG = FAC (just use MOVAF)

Math Functions

Address	Symbol	Description
\$FE12	FSUB	FACC = MEM - FACC
\$FE15	FSUBT	FACC = ARG - FACC
\$FE18	FADD	FACC = MEM + FACC
\$FE1B	FADDT	FACC = ARG + FACC
\$FE1E	FMULT	FACC = MEM * FACC
\$FE21	FMULTT	FACC = ARG * FACC
\$FE24	FDIV	FACC = MEM / FACC
\$FE27	FDIVT	FACC = ARG / FACC
\$FE2A	LOG	FACC = natural log of FACC
\$FE2D	INT	FACC = INT() truncate of FACC
\$FE30	SQR	FACC = square root of FACC
\$FE33	NEGOP	negate FACC (switch sign)
\$FE36	FPWR	FACC = raise ARG to the MEM power
\$FE39	FPWRT	FACC = raise ARG to the FACC power
\$FE3C	EXP	FACC = EXP of FACC
\$FE3F	COS	FACC = COS of FACC
\$FE42	SIN	FACC = SIN of FACC
\$FE45	TAN	FACC = TAN of FACC
\$FE48	ATN	FACC = ATN of FACC
\$FE4B	ROUND	FACC = round FACC
\$FE4E	ABS	FACC = absolute value of FACC
\$FE51	SIGN	.A = test sign of FACC
\$FE54	FCOMP	.A = compare FACC with MEM
\$FE57	RND_0	FACC = random floating point number

X16 Additions to math functions

The following calls are new to the X16 and were not part of the C128 math library API:

Address	Symbol	Description
\$FE6F	FADDH	FACC += .5
\$FE72	ZEROFAC	FACC = 0
\$FE75	NORMAL	Normalize FACC
\$FE78	NEGFAC	FACC = -FACC (just use NEGOP)
\$FE7B	MUL10	FACC *= 10
\$FE7E	DIV10	FACC /= 10
\$FE84	SGN	FACC = sgn(FACC)
\$FE90	FINLOG	FACC += (s8).A add signed byte to float
\$FE96	POLYX	Polynomial Evaluation 1 (SIN/COS/ATN/LOG)
\$FE99	POLY	Polynomial Evaluation 2 (EXP)

How to use the routines

Concepts:

- **FACC** (sometimes abbreviated to FAC): the floating point accumulator. You can compare this to the 6502 CPU's .A register, which is the accumulator for most integer operations performed by the CPU. FACC is the primary floating point register. Calculations are done on the value in this register, usually combined with ARG. After the operation, usually the original value in FACC has been replaced by the result of the calculation.
- **ARG**: the second floating point register, used in most calculation functions. Often the value in this register will be lost after a calculation.
- **MEM**: means a floating point value stored in system memory somewhere. The format is [40 bits \(5 bytes\) Microsoft binary format](#). To be able to work with given values in calculations, they need to be stored in memory somewhere in this format. To do this you'll likely need to use a separate program to pre-convert floating point numbers to this format, unless you are using a compiler that directly supports it.

Note that FACC and ARG are just a bunch of zero page locations. This means you can poke around in them. But that's not good practice because their locations aren't guaranteed/public, and the format is slightly different than how the 5-byte floats are normally stored into memory. Just use one of the Movement routines to copy values into or out of FACC and ARG.

To perform a floating point calculation, follow the following pattern:

1. load a value into FACC. You can convert an integer, or move a MEM float number into FACC.
2. do the same but for ARG, the second floating point register.
3. call the required floating point calculation routine that will perform a calculation on FACC with ARG.
4. repeat the previous 2 steps if required.
5. the result is in FACC, move it into MEM somewhere or convert it to another type or string.

An example program that calculates and prints the distance an object has fallen over a certain period using the formula $d = \frac{1}{2} g t^2$

```
; calculate how far an object has fallen: d = 1/2 * g * t^2.
; we set g = 9.81 m/sec^2, time = 5 sec -> d = 122.625 m.
```



```

CHROUT = $ffd2
FOUT   = $fe06
FMULTT = $fe21
FDIV    = $fe24
CONUPK  = $fe5a
MOVFM   = $fe63

    lda #4
    sta $01      ; rom bank 4 (BASIC) contains the fp routines.
    lda #<flt_two
    ldy #>flt_two
    jsr MOVFM
    lda #<flt_g
    ldy #>flt_g
    jsr FDIV      ; FACC= g/2
    lda #<flt_time
    ldy #>flt_time
    jsr CONUPK    ; ARG = time
    jsr FMULTT    ; FACC = g/2 * time
    lda #<flt_time
    ldy #>flt_time
    jsr CONUPK    ; again ARG = time
    jsr FMULTT    ; FACC = g/2 * time * time
    jsr FOUT      ; to string
    ; print string in AY
    sta $02
    sty $03
    ldy #0
loop:
    lda ($02),y
    beq done
    jsr CHROUT
    iny
    bne loop
done:
    rts

flt_g:      .byte $84, $1c, $f5, $c2, $8f ; float 9.81
flt_time:   .byte $83, $20, $00, $00, $00 ; float 5.0
flt_two:    .byte $82, $00, $00, $00, $00 ; float 2.0

```

Notes

- `RND_0` : For `.Z=1`, the X16 behaves differently than the C128 and C65 versions. The X16 version takes entropy from `.A/.X/.Y` instead of from a CIA timer. So in order to get a "real" random number, you would use code like this:

```

LDA #$00
PHP
JSR entropy_get ; KERNAL call to get entropy into .A/.X/.Y
PLP             ; restore .Z=1
JSR RND_0

```

- The calls `FADDT` , `FMULTT` , `FDIVT` and `FPWRT` were broken on the C128/C65. They are fixed on the X16.

- For more information on the additional calls, refer to [Mapping the Commodore 64](#) by Sheldon Leemon, ISBN 0-942386-23-X, but note these errata:
 - `FMULT` adds mem to FAC, not ARG to FAC

Chapter 6: Machine Language Monitor

The built-in machine language monitor can be started with the `MON` BASIC command. It is based on the monitor of the Final Cartridge III and supports most of its features.

If you invoke the monitor by mistake, you can exit with by typing `X`, followed by the `RETURN` key.

Some features specific to this monitor are:

- The `I` command prints a PETSCII/ISO-encoded memory dump.
- The `EC` command prints a binary memory dump. This is also useful for character sets.
- Scrolling the screen with the cursor keys or F3/F5 will continue memory dumps and disassemblies, and even disassemble backwards.

The following commands are used to dump memory contents in various formats:

Dump	Prefix	description
M	:	8 hex bytes
I	'	32 PETSCII/ISO characters
EC	[1 binary byte (character data)
ES]	3 binary bytes (sprite data)
D	,	disassemble
R	;	registers

Except for `R`, these commands take a start address and an optional end address (inclusive). The dumps are prefixed with one of the "Prefix" characters in the table above, so they can be edited by navigating the cursor over a printed line, changing the data and pressing `RETURN`.

Note that editing a disassembled line (prefix `,`) only allows changing the 1-3 opcode bytes. To edit the assembly, change the prefix to `A` (see below).

These are the remaining commands:

Command	Syntax	Description
F	<i>start end byte</i>	fill
H	<i>start end byte [byte...]</i>	hunt
C	<i>start end start</i>	compare
T	<i>start end start</i>	transfer
A	<i>address instruction</i>	assemble
G	[<i>address</i>]	run code
J	[<i>address</i>]	run subroutine
\$	<i>value</i>	convert hex to decimal
#	<i>value</i>	convert decimal to hex
X		exit monitor

0	<i>bank</i>	set ROM bank
K	<i>bank</i>	set RAM/VRAM bank/I2C
L	<code>["filename",dev[,start]]</code>	load file
S	<code>"filename",dev,start,end</code>	save file
@	<i>command</i>	send drive command

- All addresses have to be 4 digits.
- All bytes have to be 2 digits (including device numbers).
- The end address of S is exclusive.
- The bank argument for K is
 - 00 - FF : switch to main RAM, set RAM bank
 - V0 - V1 : switch to Video RAM, set VRAM bank
 - I : switch to the I2C address space
- The bank argument for 0 is
 - 00 - FF : set ROM bank
- @ takes:
 - 8 , 9 to change the default drive (also for L)
 - \$ to display the disk directory
 - anything else as a disk command

Chapter 7: Memory Map

The Commander X16 has 512 KB of ROM and 2,088 KB (2 MB [^1](#) + 40 KB) of RAM with up to 3.5MB of RAM or ROM available to cartridges.

Some of the ROM/RAM is always visible at certain address ranges, while the remaining ROM/RAM is banked into one of two address windows.

This is an overview of the X16 memory map:

Addresses	Description
\$0000-\$9EFF	Fixed RAM (40 KB minus 256 bytes)
\$9F00-\$9FFF	I/O Area (256 bytes)
\$A000-\$BFFF	Banked RAM (8 KB window into one of 256 banks for a total of 2 MB)
\$C000-\$FFFF	Banked System ROM and Cartridge ROM/RAM (16 KB window into one of 256 banks, see below)

Banked Memory

Writing to the following zero-page addresses sets the desired RAM or ROM bank:

Address	Description
\$0000	Current RAM bank (0-255)
\$0001	Current ROM/Cartridge bank (ROM is 0-31, Cartridge is 32-255)

The currently set banks can also be read back from the respective memory locations. Both settings default to 0 on RESET.

ROM Allocations

Here is the ROM/Cartridge bank allocation:

Bank	Name	Description
0	KERNAL	KERNAL operating system and drivers
1	KEYBD	Keyboard layout tables
2	CBDOS	The computer-based CBM-DOS for FAT32 SD cards
3	FAT32	The FAT32 driver itself
4	BASIC	BASIC interpreter
5	MONITOR	Machine Language Monitor
6	CHARSET	PETSCII and ISO character sets (uploaded into VRAM)
7	CODEX	CodeX16 Interactive Assembly Environment / Monitor
8	GRAPH	Kernal graph and font routines
9	DEMO	Demo routines

10	AUDIO	Audio API routines
11	UTIL	System Configuration (Date/Time, Display Preferences)
12	BANNEX	BASIC Annex (code for some added BASIC functions)
13-14	X16EDIT	The built-in text editor
13-31	-	<i>[Currently unused]</i>
32-255	-	Cartridge RAM/ROM

Important: The layout of the banks may still change.

Cartridge Allocation

Cartridges can use the remaining 32-255 banks in any combination of ROM, RAM, Memory-Mapped IO, etc. See Kevin's reference cartridge design for ideas on how this may be used. This provides up to 3.5MB of additional RAM or ROM.

Important: The layout of the banks is not yet final.

RAM Contents

This is the allocation of fixed RAM in the KERNAL/BASIC environment.

Addresses	Description
\$0000-\$00FF	Zero page
\$0100-\$01FF	CPU stack
\$0200-\$03FF	KERNAL and BASIC variables, vectors
\$0400-\$07FF	Available for machine code programs or custom data storage
\$0800-\$9EFF	BASIC program/variables; available to the user

The \$0400-\$07FF can be seen as the equivalent of \$C000-\$CFFF on a C64. A typical use would be for helper machine code called by BASIC.

Zero Page

Addresses	Description
\$0000-\$0001	Banking registers
\$0002-\$0021	16 bit registers r0-r15 for KERNAL API
\$0022-\$007F	Available to the user
\$0080-\$009C	Used by KERNAL and DOS
\$009D-\$00A8	Reserved for DOS/BASIC
\$00A9-\$00D3	Used by the Math library (and BASIC)
\$00D4-\$00FF	Used by BASIC

Machine code applications are free to reuse the BASIC area, and if they don't use the Math library, also that area.

Banking

This is the allocation of banked RAM in the KERNAL/BASIC environment.

Bank	Description
0	Used for KERNAL/CBDOS variables and buffers
1-255	Available to the user

(On systems with only 512 KB RAM, banks 64-255 are unavailable.)

During startup, the KERNAL activates RAM bank 1 as the default for the user.

I/O Area

This is the memory map of the I/O Area:

Addresses	Description	Speed
\$9F00-\$9F0F	VIA I/O controller #1	8 MHz
\$9F10-\$9F1F	VIA I/O controller #2	8 MHz
\$9F20-\$9F3F	VERA video controller	8 MHz
\$9F40-\$9F41	YM2151 audio controller	2 MHz
\$9F42-\$9F5F	Unavailable	---
\$9F60-\$9F7F	Expansion Card Memory Mapped IO3	8 MHz
\$9F80-\$9F9F	Expansion Card Memory Mapped IO4	8 MHz
\$9FA0-\$9FBF	Expansion Card Memory Mapped IO5	2 MHz
\$9FC0-\$9FDF	Expansion Card Memory Mapped IO6	2 MHz
\$9FE0-\$9FFF	Cartidge/Expansion Memory Mapped IO7	2 MHz

Expansion Cards & Cartridges

Expansion cards can be accessed via memory-mapped I/O (MMIO), as well as I2C. Cartridges are essentially expansion cards which are housed in an external enclosure and may contain RAM, ROM and an I2C EEPROM (for save data). Internal expansion cards may also use the RAM/ROM space, though this could cause conflicts.

While they may be uncommon, since cartridges are essentially external expansion cards in a shell, that means they can also use MMIO. This is only necessary when a cartridge includes some sort of hardware expansion and MMIO was desired (as opposed to using the I2C bus). In that case, it is recommended cartridges use the IO7 range and that range should be the last option used by expansion cards in the system. **MMIO is unneeded for cartridges which simply have RAM/ROM.**

For more information, consult the [Hardware](#) section of the manual.

¹: Current development systems have 2 MB of bankable RAM. Actual hardware is currently planned to have an option of either 512 KB or 2 MB of RAM.

Chapter 8: Video Programming

The VERA video chip supports resolutions up to 640x480 with up to 256 colors from a palette of 4096, two layers of either a bitmap or tiles, 128 sprites of up to 64x64 pixels in size. It can output VGA as well as a 525 line interlaced signal, either as NTSC or as RGB (Amiga-style).

See the [VERA Programmer's Reference](#) for the complete reference.

The X16 KERNAL uses the following video memory layout:

Addresses	Description
\$00000-\$12BFF	320x240@256c Bitmap
\$12C00-\$12FFF	<i>unused</i>
\$13000-\$1AFFF	Sprite Image Data (up to \$1000 per sprite at 64x64 8-bit)
\$1B000-\$1EBFF	Text Mode
\$1EC00-\$1EFFF	<i>unused</i>
\$1F000-\$1F7FF	Charset
\$1F800-\$1F9BF	<i>unused</i>
\$1F9C0-\$1F9FF	VERA PSG Registers (16 x 4 bytes)
\$1FA00-\$1FBFF	VERA Color Palette (256 x 2 bytes)
\$1FC00-\$1FFFF	VERA Sprite Attributes (128 x 8 bytes)

Application software is free to use any part of video RAM if it does not use the corresponding KERNAL functionality. To restore text mode, call `CINT` (\$FF81).

Chapter 9: Sound Programming

Audio bank API

The Commander X16 provides many convenience routines for controlling the YM2151 and VERA PSG. These are called similarly to how KERNAL API calls are done in machine language.

In order to gain access to these routines, you must either use `jsrfar` from the KERNAL API:

```
AUDIO_BANK = $0A

jsr jsrfar ; $FF6E
.word ym_init ; $C063
.byte AUDIO_BANK
```

or switch to ROM bank `$0A` directly:

```
lda #$0A ; Audio bank number
sta $01 ; ROM bank register
```

Conveniently, the KERNAL API still exists in this bank, and calling a KERNAL API routine will automatically switch your ROM bank back to the KERNAL bank to perform the routine and then switch back right before returning, so there's usually no need for your audio-centric program to switch away from the audio bank to perform the occasional KERNAL API call.

Audio API routines

For the audio chips, some of the documentation uses the words *channel* and *voice* interchangeably. This table of API routines uses *channel* for the 8 on the YM2151, and *voice* for the 16 on the PSG.

Label	Address	Class	Description	Inputs	Returns	Preserves
audio_init	\$C09F	-	Wrapper routine that calls both <code>psg_init</code> and <code>ym_init</code> followed by <code>ym_loaddefpatches</code> . This is the routine called by the KERNAL at reset.	none	none	none
bas_fmchordstring	\$C08D	BASIC	Starts playing all of notes specified in a string. This uses the same parser as <code>bas_fmplaystring</code> but instead of playing the notes in sequence, it starts playback of each note in the string, on many channels as is	.A = string length .X .Y = pointer to string	none	none

			necessary, then returns to the caller without delay. The first FM channel that is used is the one specified by calling <code>bas_playstringvoice</code> prior to calling this routine. The string pointer must point to low RAM (\$0000-\$9EFF).			
<code>bas_fmfreq</code>	\$C000	BASIC	Plays a note specified in Hz on an FM channel	.A = channel .X .Y = 16-bit frequency in Hz .C clear = normal .C set = no retrigger	.C clear = success .C set = error	none
<code>bas_fmnote</code>	\$C003	BASIC	Plays a note specified in BASIC format on an FM channel	.A = channel .X = note (BASIC format) .Y = fractional semitone .C clear = normal .C set = no retrigger	.C clear = success .C set = error	none
<code>bas_fmplaystring</code>	\$C006	BASIC	Plays a note script using the FM channel which was specified on a previous call to <code>bas_playstringvoice</code> . This string pointer must point to low RAM (\$0000-\$9EFF). This routine depends on interrupts being enabled. In particular, it uses <code>WAI</code> as a delay for timing, so it expects IRQ to be asserted and acknowledged	.A = string length .X .Y = pointer to string	none	none

			once per video frame, which is the case by default on the system. Stops playback and returns control if the STOP key is pressed.			
bas_fm vib	\$C009	BASIC	Sets the LFO speed and both amplitude and frequency depth based on inputs. Also sets the LFO waveform to triangle.	.A = speed .X = PMD/AMD depth	.C clear = success .C set = error	none
bas_playstringvoice	\$C00C	BASIC	Preparatory routine for bas_fmplaystring and bas_psgplaystring to set the voice/channel number for playback	.A = PSG/YM voice/channel	none	.A .X
bas_psgchordstring	\$C090	BASIC	Starts playing all of notes specified in a string. This uses the same parser as bas_psgplaystring but instead of playing the notes in sequence, it starts playback of each note in the string, on many voices as is necessary, then returns to the caller without delay. The first PSG voice that is used is the one specified by calling bas_playstringvoice prior to calling this routine. The string pointer must point	.A = string length .X .Y = pointer to string	none	none

			to low RAM (\$0000-\$9EFF).			
bas_psgfreq	\$C00F	BASIC	Plays a note specified in Hz on a PSG voice	.A = voice .X .Y = 16-bit frequency	.C clear = success .C set = error	none
bas_psgnote	\$C012	BASIC	Plays a note specified in BASIC format on a PSG voice	.A = voice .X = note (BASIC format) .Y = fractional semitone	.C clear = success .C set = error	none
bas_psgwav	\$C015	BASIC	Sets a waveform and duty cycle for a PSG voice	.A = voice .X 0-63 = Pulse, 1/128 - 64/128 duty cycle .X 64-127 = Sawtooth .X 128-191 = Triangle .X 192-255 = Noise	.C clear = success .C set = error	none
bas_psgplaystring	\$C018	BASIC	Plays a note script using the PSG voice which was specified on a previous call to <code>bas_playstringvoice</code> . This string pointer must point to low RAM (\$0000-\$9EFF). This routine depends on interrupts being enabled. In particular, it uses <code>WAI</code> as a delay for timing, so it expects IRQ to be asserted and acknowledged once per video frame, which is the case by default on the system. Stops playback and returns control if	.A = string length .X .Y = pointer to string	none	none

			the STOP key is pressed.			
notecon_bas2fm	\$C01B	Conversion	Convert a note in BASIC format to a YM2151 KC code	.X = note (BASIC format)	.X = note (YM2151 KC) .C clear = success .C set = error	.Y
notecon_bas2midi	\$C01E	Conversion	Convert a note in BASIC format to a MIDI note number	.X = note (BASIC format)	.X = MIDI note .C clear = success .C set = error	.Y
notecon_bas2psg	\$C021	Conversion	Convert a note in BASIC format to a PSG frequency	.X = note (BASIC format) .Y = fractional semitone	.X .Y = PSG frequency .C clear = success .C set = error	none
notecon_fm2bas	\$C024	Conversion	Convert a note in YM2151 KC format to a note in BASIC format	.X = YM2151 KC	.X = note (BASIC format) .C clear = success .C set = error	.Y
notecon_fm2midi	\$C027	Conversion	Convert a note in YM2151 KC format to a MIDI note number	.X = YM2151 KC	.X = MIDI note .C clear = success .C set = error	.Y
notecon_fm2psg	\$C02A	Conversion	Convert a note in YM2151 KC format to a PSG frequency	.X = YM2151 KC .Y = fractional semitone	.X .Y = PSG frequency .C clear = success .C set = error	none
notecon_freq2bas	\$C02D	Conversion	Convert a frequency in Hz to a note in BASIC format and a fractional semitone	.X .Y = 16-bit frequency in Hz	.X = note (BASIC format) .Y = fractional semitone .C clear = success .C set = error	none

notecon_freq2fm	\$C030	Conversion	Convert a frequency in Hz to YM2151 KC and a fractional semitone (YM2151 KF)	.X .Y = 16-bit frequency in Hz	.X = YM2151 KC .Y = fractional semitone (YM2151 KF) .C clear = success .C set = error	none
notecon_freq2midi	\$C033	Conversion	Convert a frequency in Hz to a MIDI note and a fractional semitone	.X .Y = 16-bit frequency in Hz	.X = MIDI note .Y = fractional semitone .C clear = success .C set = error	none
notecon_freq2psg	\$C036	Conversion	Convert a frequency in Hz to a VERA PSG frequency	.X .Y = 16-bit frequency in Hz	.X .Y = 16-bit frequency in VERA PSG format .C clear = success .C set = error	none
notecon_midi2bas	\$C039	Conversion	Convert a MIDI note to a note in BASIC format	.X = MIDI note	.X = note (BASIC format) .C clear = success .C set = error	.Y
notecon_midi2fm	\$C03C	Conversion	Convert a MIDI note to a YM2151 KC. Fractional semitone is unneeded as it is identical to KF already.	.X = MIDI note.	.X = YM2151 KC .C clear = success .C set = error	.Y
notecon_midi2psg	\$C03F	Conversion	Convert a MIDI note and fractional semitone to a PSG frequency	.X = MIDI note .Y = fractional semitone	.X .Y = 16-bit frequency in VERA PSG format .C clear = success	none

					.C set = error	
notecon_psg2bas	\$C042	Conversion	Convert a frequency in VERA PSG format to a note in BASIC format and a fractional semitone	.X .Y = 16-bit frequency in VERA PSG format	.X = note (BASIC format) .Y = fractional semitone .C clear = success .C set = error	none
notecon_psg2fm	\$C045	Conversion	Convert a frequency in VERA PSG format to YM2151 KC and a fractional semitone (YM2151 KF)	.X .Y = 16-bit frequency in VERA PSG format	.X = YM2151 KC .Y = fractional semitone (YM2151 KF) .C clear = success .C set = error	none
notecon_psg2midi	\$C048	Conversion	Convert a frequency in VERA PSG format to a MIDI note and a fractional semitone	.X .Y = 16-bit frequency in VERA PSG format	.X = MIDI note .Y = fractional semitone .C clear = success .C set = error	none
psg_getatten	\$C093	VERA PSG	Retrieve the attenuation value for a voice previously set by psg_setatten	.A = voice	.X = attenuation value	.A
psg_getpan	\$C096	VERA PSG	Retrieve the simple panning value that is currently set for a voice.	.A = voice	.X = pan value	.A
psg_init	\$C04B	VERA PSG	Initialize the state of the PSG. Silence all voices. Reset the attenuation levels to 0. Set "playstring" defaults including 04, T120, S1, and L4. Set all PSG voices	none	none	none

			to the pulse waveform at 50% duty with panning set to both L+R			
psg_playfreq	\$C04E	VERA PSG	Turn on a PSG voice at full volume (factoring in attenuation) and set its frequency	.A = voice .X .Y = 16 bit frequency in VERA PSG format	none	none
psg_read	\$C051	VERA PSG	Read a value from one of the VERA PSG registers. If the selected register is a volume register, return either the cooked value (attenuation applied) or the raw value (as received by psg_write or psg_setvol, or as set by psg_playfreq) depending on the state of the carry flag	.X = PSG register address (offset from \$1F9C0) .C clear = if volume, return raw .C set = if volume, return cooked	.A = register value	.X
psg_setatten	\$C054	VERA PSG	Set the attenuation value for a PSG voice. The valid range is from \$00 (full volume) to \$3F (fully muted). API routines which affect volume will deduct the attenuation value from the intended volume before setting it. Calls to this routine while a note is playing will change the output volume of the voice immediately. This control can be considered a "master volume" for the voice.	.A = voice .X = attenuation	none	none

psg_setfreq	\$C057	VERA PSG	Set the frequency of a PSG voice without changing any other attributes of the voice	.A = voice .X .Y = 16 bit frequency in VERA PSG format	none	none
psg_setpan	\$C05A	VERA PSG	Set the simple panning for the voice. A value of 0 will silence the voice entirely until another pan value is set.	.A = voice .X 0 = none .X 1 = left .X 2 = right .X 3 = both	none	none
psg_setvol	\$C05D	VERA PSG	Set the volume for the voice. The volume that's written to the VERA has attenuation applied. Valid volumes range from \$00 to \$3F inclusive	.A = voice .X = volume	none	none
psg_write	\$C060	VERA PSG	Write a value to one of the VERA PSG registers. If the selected register is a volume register, attenuation will be applied before the value is written to the VERA	.A = value .X = PSG register address (offset from \$1F9C0)	none	.A .X
psg_write_fast	\$C0A2	VERA PSG	Same effect as psg_write but does not preserve the state of the VERA CTRL and ADDR registers. It also assumes VERA_CTRL bit 0 is clear, VERA_ADDR0_H = \$01 (auto increment 0 recommended), and VERA_ADDR0_M = \$F9. This routine is meant for use by sound engines	.A = value .X = PSG register address (offset from \$1F9C0)	none	.A .X

			that typically write out multiple PSG registers in a loop.			
ym_getatten	\$C099	YM2151	Retrieve the attenuation value for a channel previously set by ym_setatten	.A = channel	.X = attenuation value	.A
ym_getpan	\$C09C	YM2151	Retrieve the simple panning value that is currently set for a channel.	.A = channel	.X = pan value	.A
ym_init	\$C063	YM2151	Initialize the state of the YM chip. Silence all channels by setting the release part of the ADSR envelope to max and then setting all channels to released. Reset all attenuation levels to 0. Set "playstring" defaults including 04, T120, S1, and L4. Set panning for all channels set to both L+R. Reset LFO state. Set all of the other registers to \$00	none	.C clear = success .C set = error	none
ym_loaddefpatches	\$C066	YM2151	Load a default set of patches into the 8 channels. C0: Piano (0) C1: E. Piano (5) C2: Vibraphone (11) C3: Fretless (35) C4: Violin (40) C5: Trumpet (56) C6: Blown Bottle (76) C7: Fantasia (88)	none	.C clear = success .C set = error	none
ym_loadpatch	\$C069	YM2151	Load into a channel a patch preset by number (0-161) from the audio bank, or	.A = channel .C clear = .X .Y = patch address .C set = .X =	.C clear = success .C set = error	none

			from an arbitrary memory location. High RAM addresses (\$A000-\$BFFF) are accepted in this mode.	patch number		
ym_loadpatch1fn	\$C06C	YM2151	Load patch into a channel by way of an open logical file number. This routine will read 26 bytes from the open file, or possibly fewer bytes if there's an error condition. The routine will leave the file open on return. On return if .C is set, check .A for the error code.	.A = channel .X = Logical File Number	.C clear = success .C set .A=0 = YM error .C set .A&3=2 = read timeout .C set .A&3=3 = file not open .C set .A&64=64 = EOF .C set .A&128=128 = device not present	none
ym_playdrum	\$C06F	YM2151	Load a patch associated with a MIDI drum note number and trigger it on a channel. Valid drum note numbers mirror the General MIDI percussion standard and range from 25 (Snare Roll) through 87 (Open Surdo). Note 0 will release the note. After the drum is played, the channel will still contain the patch for the drum sound and thus may not sound musical if you attempt to play notes on it before	.A = channel .X = drum note	.C clear = success .C set = error	none

			loading another instrument patch.			
ym_playnote	\$C072	YM2151	Set a KC/KF on a channel and optionally trigger it.	.A = channel .X = KC .Y = KF (fractional semitone) .C clear = trigger .C set = no trigger	.C clear = success .C set = error	none
ym_setatten	\$C075	YM2151	Set the attenuation value for a channel. The valid range is from \$00 (full volume) to \$7F (fully muted). API routines which affect TL or CON will add the attenuation value to the intended TL on operators that are carriers before setting it. Calls to this routine will change the TL of the channel's carriers immediately. This control can be considered a "master volume" for the channel.	.A = channel .X = attenuation	.C clear = success .C set = error	.A .X
ym_setdrum	\$C078	YM2151	Load a patch associated with a MIDI drum note number and set the KC/KF for it on a channel. Called by ym_playdrum.	.A = channel .X = drum note	.C clear = success .C set = error	none
ym_setnote	\$C07B	YM2151	Set a KC/KF on a channel. Called by ym_playnote.	.A = channel .X = KC .Y = KF (fractional semitone)	.C clear = success .C set = error	none

ym_setpan	\$C07E	YM2151	Set the simple panning for the channel. A value of 0 will silence the channel entirely until another pan value is set.	.A = channel .X 0 = none .X 1 = left .X 2 = right .X 3 = both	.C clear = success .C set = error	none
ym_read	\$C081	YM2151	Read a value from the in-RAM shadow of one of the YM2151 registers. The YM2151's internal registers cannot be read from, but this API keeps state of what was written, so this routine will be able to retrieve chip values for you. If the selected register is a TL register, return either the cooked value (attenuation applied) or the raw value (as received by ym_write) depending on the state of the carry flag	.X = YM2151 register address .C clear = if TL, return raw .C set = if TL, return cooked	.A = register value .C clear = success .C set = error	.X
ym_release	\$C084	YM2151	Release a note on a channel. If a note is not playing, this routine has no tangible effect	.A = channel	.C clear = success .C set = error	none
ym_trigger	\$C087	YM2151	Trigger the currently configured note on a channel, optionally releasing the channel first depending on the state of the carry flag.	.A = channel .C clear = release first .C set = no release	.C clear = success .C set = error	none
ym_write	\$C08A	YM2151	Write a value to one of the YM2151 registers and to	.A = value .X = YM	.C clear = success	.A .X

			the in-RAM shadow copy. If the selected register is a TL register, attenuation will be applied before the value is written. Writes which affect which operators are carriers will have TL values for that channel appropriately recalculated and rewritten	register address	.C set = error	
--	--	--	--	------------------	----------------	--

A note on semitones (get it?)

It may be advantageous to consider storing note data internally as the MIDI representation with a fractional component if you want things like pitch slides to behave the same way between the PSG and YM2151.

Essentially, it can be thought of as an 8.8 fixed point 16-bit number.

The YM2151 handles semitones differently than the PSG and requires converting the MIDI note to the appropriate KC value using `notecon_midi2fm`. KF is the fractional semitone (albeit with only the top 6-bits used) and requires no conversion.

The PSG, by contrast, operates with linear pitch which is why `notecon_midi2psg` also takes the fractional component (y) as input.

Thus, if you manage all your pitch slides using MIDI notes along with a fractional component, you can then convert this directly over to PSG or YM2151 as required and end up with the same pitch (or close enough to it).

Direct communication with the YM2151 and VERA PSG vs API

Use of the API routines above is not required to access the capabilities of the sound chips. However, mixing raw writes to a chip and API access for the same chip is not recommended, particularly where PSG volumes and YM2151 TL and RLFBCON registers are concerned. The API processes volumes, calculating attenuation and adjusting the output volume accordingly, and the API will be oblivious to direct manipulation of the sound chips.

The sections below describe how to do raw access to the sound chips outside of the API.

VERA PSG and PCM Programming

- For VERA PSG and PCM, refer to the [VERA Programmer's Reference](#).

YM2151 (OPM) FM Synthesis

The Yamaha YM2151 (OPM) sound chip is an FM synthesizer ASIC in the Commander X16. It is connected to the system bus at I/O address `0x9F40` (address register) and at `0x9F41` (data register). It has 8 independent voices with 4 FM operators each. Each voice is capable of left/right/both audio channel output. The four operators of each channel may be connected in one of 8 pre-defined "connection algorithms" in order to produce a wide variety of timbres.

YM2151 Communication:

There are 3 basic operations to communicate with the YM chip: Reading its status, address select, and data write. These are performed by reading from or writing to one of the two I/O addresses as follows:

Address	Name	Read Action	Write Action
0x9F40	YM_address	Undefined (returns ?)	Selects the internal register address where data is written.
0x9F41	YM_data	Returns the YM_status byte	Writes the value into the currently-selected internal address.

The values stored in the YM's internal registers are write-only. If you need to know the values in the registers, you must store a copy of the values somewhere in memory as you write updates to the YM.

YM Write Procedure

1. Ensure YM is not busy (see Write Timing below).
2. Select the desired internal register address by writing it into `YM_address`.
3. Write the new value for this register into `YM_data`.

Note: You may write into the same register multiple times without repeating a write to `YM_address`. The same register will be updated with each data write.

Write Timing:

The YM2151 is sensitive to the speed at which you write data into it. If you make writes when it is not ready to receive them, they will be dropped and the sound output will be corrupted.

You must include a delay between writes to the address select register (\$9F40) and the subsequent data write. 10 CPU cycles is the recommended minimum delay.

The YM becomes `BUSY` for approximately 150 CPU cycles' (at 8Mhz) whenever it receives a data write. *Any writes into YM_data during this BUSY period will be ignored!*

In order to avoid this, you can use the `BUSY` flag which is bit 7 of the `YM_status` byte. Read the status byte from `YM_data` (0x9F41). If the top bit (7) is set, the YM may not be written into at this time. Note that it is not *required* that you read `YM_status`, only that writes occur no less than ~150 CPU cycles apart. For instance, BASIC executes slowly enough that you are in no danger of writing into the YM too quickly, so BASIC programs may skip checking `YM_status`.

Lastly, the `BUSY` flag sometimes takes a (very) short period before it goes high. This has only been observed when IMMEDIATELY polling the flag after a write into `YM_data`. As long as your code does not do so, this quirk should not be an issue.

Example Code:

Assembly Language:

```
check_busy:
    BIT YM_data      ; check busy flag
    BMI check_busy   ; wait until busy flag is clear
    LDA #$08         ; Select YM register $08 (Key-Off/On)
    STA YM_addr      ;
    NOP              ;<--+
    NOP              ; |
    NOP              ; +---slight pause before writing data
    NOP              ; |
    NOP              ;<--+
    LDA #$04         ; Write $04 (Release note on channel 4).
```

```
STA YM_data
RTS
```

BASIC:

```
10 YA=$9F40      : REM YM_ADDRESS
20 YD=$9F41      : REM YM_DATA
30 POKE YA,$29    : REM CHANNEL 1 NOTE SELECT
40 POKE YD,$4A    : REM SET NOTE = CONCERT A
50 POKE YA,$08    : REM SELECT THE KEY ON/OFF REGISTER
60 POKE YD,$00+1 : REM RELEASE ANY NOTE ALREADY PLAYING ON CHANNEL 1
70 POKE YD,$78+1 : REM KEY-ON VOICE 1 TO PLAY THE NOTE
80 FOR I=1 TO 100 : NEXT I : REM DELAY WHILE NOTE PLAYS
90 POKE YD,$00+1 : REM RELEASE THE NOTE
```

YM2151 Internal Addressing

The YM register address space can be thought of as being divided into 3 ranges:

Range	Type	Description
00 .. 1F	Global Values	Affect individual global parameters such as LFO frequency, noise enable, etc.
20 .. 3F	Channel CFG	Parameters in groups of 8, one per channel. These affect the whole channel.
40 .. FF	Operator CFG	Parameters in groups of 32 - these map to individual operators of each voice.

YM2151 Register Map

Global Settings:

Addr	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
\$01	Test Register	!	!	!	!	!	!	LR	!	Bit 1 is the LFO reset bit. Setting it disables the LFO and holds the oscillator at 0. Clearing it enables the LFO. All other bits control various test functions and should not be written into.
\$08	Key Control	.	C2	M2	C1	M1	CHA			Starts and Releases notes on the 8 channels. Setting/Clearing bits for M1,C1,M2,C2 controls the key state for those operators on channel CHA. NOTE: The operator order is different than the order they appear in the Operator configuration registers!
\$0F	Noise Control	NE	.	.	NFRQ					NE = Noise Enable NFRQ = Noise Frequency

									When eabled, C2 of channel 7 will use a noise waveform instead of a sine waveform.	
\$10	Ta High	CLKA1							Top 8 bits of Timer A period setting	
\$11	Ta Low	CLKA2	Bottom 2 bits of Timer A period setting	
\$12	Timer B	CLKB							Timer B period setting	
\$14	IRQ Control	CSM	.	Clock ACK		IRQ EN		Clock Start	CSM: When a timer expires, trigger note key-on for all channels. For the other 3 fields, lower bit = Timer A, upper bit = Timer B. Clock ACK: clears the timer's bit in the YM_status byte and acknowledges the IRQ.	
\$18	LFO Freq.	LFRQ							Sets LFO frequency. \$00 = ~0.008Hz \$FF = ~32.6Hz	
\$19	LFO Amplitude	0	AMD							AMD = Amplitude Modulation Depth PMD = Phase Modulation (vibrato) Depth Bit 7 determines which parameter is being set when writing into this register.
		1	PMD							
\$1B	CT / LFO Waveform	CT		W	CT: sets output pins CT1 and CT1 high or low. (not connected to anything in X16) W: LFO Waveform: 0-4 = Saw, Square, Triange, Noise For sawtooth: PM->//// AM->\\	

Channel CFG Registers:

Register Range	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
\$20 + channel	RL		FB			CON			RL Right/Left Output Enable FB M1 Feedback Level
\$28 + channel	.	KC							CON Operator connection algorithm KC Key Code
\$30 + channel	KF						.	.	KF Key Fraction

						PMS
\$38 + channel	.	PMS	.	.	AMS	AMS
						Phase Modulation Sensitivity
						Amplitude Modulation Sensitivity

Operator CFG Registers:

Register Range	Operator	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
\$40	M1: \$40+channel	.	DT1			MUL				DT1 Detune Amount (fine)
	M2: \$48+channel									MUL Frequency Multiplier
	C1: \$50+channel									
	C2: \$58+channel									
\$60	M1: \$60+channel	.	TL							TL Total Level (volume attenuation) (0=max, \$7F=min)
	M2: \$68+channel									
	C1: \$70+channel									
	C2: \$78+channel									
\$80	M1: \$80+channel	KS		.	AR					KS Key Scaling (ADSR rate scaling)
	M2: \$88+channel									AR Attack Rate
	C1: \$90+channel									
	C2: \$98+channel									
\$A0	M1: \$A0+channel	A M E n a	.	.	D1R					AM-Ena Amplitude Modulation Enable
	M2: \$A8+channel									D1R Decay Rate 1 (From peak down to sustain level)
	C1: \$B0+channel									
	C2: \$B8+channel									

\$C0	M1: \$C0+channel	DT2	.	D2R	DT2 Detune Amount (coarse) D2R Decay Rate 2 (During sustain phase)
	M2: \$C8+channel				
	C1: \$D0+channel				
	C2: \$D8+channel				
\$E0	M1: \$E0+channel	D1L		RR	D1L Decay 1 Level (Sustain level) Level at which decay switches from D1R to D2R RR Release Rate
	M2: \$E8+channel				
	C1: \$F0+channel				
	C2: \$F8+channel				

YM2151 Register Details

Global Parameters:

LR (LFO Reset)

Register \$01, bit 1

Setting this bit will disable the LFO and hold it at level 0. Clearing this bit allows the LFO to operate as normal. (See LFRQ for further info)

KON (KeyON)

Register \$08

- Bits 0-2: Channel_Number
- Bits 3-6: Operator M1, C1, M2, C2 control bits:
 - 0: Releases note on operator
 - 0->1: Triggers note attack on operator
 - 1->1: No effect

Use this register to start/stop notes. Typically, all 4 operators are triggered/released together at once. Writing a value of \$78+channel_number will start a note on all 4 OPs, and writing a value of \$00+channel_number will stop a note on all 4 OPs.

NE (Noise Enable)

Register \$0F, Bit 7

When set, the C2 operator of channel 7 will use a noise waveform instead of a sine.

NFRQ (Noise Frequency)

Register \$0F, Bits 0-4

Sets the noise frequency, \$00 is the lowest and \$1F is the highest. NE bit must be set in order for this to have any effect. Only affects operator C2 on channel 7.

CLKA1 (Clock A, high order bits)

Register \$10, Bits 0-7

This is the high-order value for Clock A (a 10-bit value).

CLKA2 (Clock A, low order bits)

Register \$11, Bits 0-1

Sets the 2 low-order bits for Clock A (a 10-bit value).

Timer A's period is Computed as $(64 * (1024 - \text{CLKA})) / \text{PhiM}$ ms. (PhiM = 3579.545Khz)

CLKB (Clock B)

Register \$12, Bits 0-7

Sets the Clock B period. The period for Timer B is computed as $(1024 * (256 - \text{CLKB})) / \text{PhiM}$ ms. (PhiM = 3579.545Khz)

CSM

Register \$14, Bit 7

When set, the YM2151 will generate a KeyON attack on all 8 channels whenever TimerA overflows.

Clock ACK

Register \$14, Bits 4-5

Clear (acknowledge) IRQ status generated by TimerA and TimerB (respectively).

IRQ EN

Register \$14, Bits 2-3

When set, enables IRQ generation when TimerA or TimerB (respectively) overflow. The IRQ status of the two timers is checked by reading from the YM2151_STATUS byte. Bit 0 = Timer A IRQ status, and Bit 1 = Timer B IRQ status. Note that these status bits are only active if the timer has overflowed AND has its IRQ_EN bit set.

Clock Start

Register \$14, Bits 0-1

When set, these bits clear the TimerA and TimerB (respectively) counters and starts it running.

LFRQ (LFO Frequency)

Register \$18, Bits 0-7

Sets the LFO frequency.

- \$00 = ~0.008Hz
- \$FF = ~32.6Hz

Note that even setting the value zero here results in a positive LFO frequency. Any channels sensitive to the LFO will still be affected by the LFO unless the **LR** bit is set in register \$01 to completely disable it.

AMD (Amplitude Modulation Depth)

Register \$19 Bits 0-6, Bit 7 clear

Sets the peak strength of the LFO's Amplitude Modulation effect. Note that bit 7 of the value written into \$19 must be clear in order to set the AMD. If bit 7 is set, the write will be interpreted as PMD.

PMD (Phase Modulation Depth)

Register \$19 Bits 0-6, Bit 7 set

Sets the peak strength of the LFO's Phase Modulation effect. Note that bit 7 of the value written into \$19 must be set in order to set the PMD. If bit 7 is clear, the value is interpreted as AMD.

CT (Control pins)

Register \$1B, Bits 6-7

These bits set the electrical state of the two CT pins to on/off. These pins are not connected to anything in the X16 and have no effect.

W (LFO Waveform)

Register \$1B, Bits 0-1

Sets the LFO waveform: 0: Sawtooth, 1: Square (50% duty cycle), 2: Triangle, 3: Noise

Channel Control Parameters:

RL (Right/Left output enable)

Register \$20 (+ channel), Bits 6-7

Setting/Clearing these bits enables/disables audio output for the selected channel. (bit6=left, bit7=right)

FB (M1 Self-Feedback)

Register \$20 (+ channel), bits 3-5

Sets the amount of self feedback on operator M1 for the selected channel. 0=none, 7=max

CON (Connection Algorithm)

Register \$20 (+ channel), bits 0-2

Sets the selected channel to connect the 4 operators in one of 8 arrangements.

[insert picture here]

KC (Key Code - Note selection)

Register \$28 + channel, bits 0-6

Sets the octave and semitone for the selected channel. Bits 4-6 specify the octave (0-7) and bits 0-3 specify the semitone:

0	1	2	4	5	6	8	9	A	C	D	E
C#	D	D#	E	F	F#	G	G#	A	A#	B	C

Note that natural C is at the TOP of the selected octave, and that each 4th value is skipped. Thus if concert A (A-4, 440hz) is KC=\$4A, then middle C is KC=\$3E

KF (Key Fraction)

Register \$30 + channel, Bits 2-7

Raises the pitch by 1/64th of a semitone * the KF value.

PMS (Phase Modulation Sensitivity)

Register \$38 + channel, Bits 4-6

Sets the Phase Modulation (vibrato) sensitivity of the selected channel. The resulting vibrato depth is determined by the combination of the global PMD setting (see above) modified by each channel's PMS.

Sensitivity values: (+/- cents)

0	1	2	3	4	5	6	7
0	5	10	20	50	100	400	700

AMS (Amplitude Modulation Sensitivity)

Register \$38 + channel, Bits 0-1

Sets the Amplitude Modulation sensitivity of the selected channel. Note that each operator may individually enable or disable this effect on its output by setting/clearing the AMS-Ena bit (see below). Operators acting as outputs will exhibit a tremolo effect (varying volume) and operators acting as modulators will vary their effectiveness on the timbre when enabled for amplitude modulation.

Sensitivity values: (dB)

0	1	2	3
0	23.90625	47.8125	95.625

Operator Control Parameters:

Operators are arranged as follows:

name	M1	M2	C1	C2
index	0	1	2	3

These are the names used throughout this document for consistency, but they may function as either modulators or carriers, depending on which `CON` ALG is used.

The Operator Control parameters are mapped to channels/operators as follows: Register + 8*op + channel. You may also choose to think of these register addresses as using bits 0-2 = channel, bits 3-4 = operator, and bits 5-7 = parameter. This reference will refer to them using the address range, e.g. \$60-\$7F = TL. To set TL for channel 2, operator 1, the register address would be \$6A (\$60 + 1*8 + 2).

DT1 (Detune 1 - fine detune)

Registers \$40-\$5F, Bits 4-6

Detunes the operator from the channel's main pitch. Values 0 and 4=no detuning. Values 1-3=detune up, 5-7 = detune down.

The amount of detuning varies with pitch. It decreases as the channel's pitch increases.

MUL (Frequency Multiplier)

Registers \$40-\$5F, Bits 0-3

If MUL=0, it multiplies the operator's frequency by 0.5

Otherwise, the frequency is multiplied by the value in MUL (1,2,3...etc)

TL (Total Level - attenuation)

Registers \$60-\$7F, Bits 0-6

This is essentially "volume control" - It is an attenuation value, so \$00 = maximum level and \$7F is minimum level. On output operators, this is the volume output by that operator. On modulating operators, this affects the amount of modulation done to other operators.

KS (Key Scaling)

Registers \$80-\$9F, Bits 6-7

Controls the speed of the ADSR progression. The KS value sets four different levels of scaling. Key scaling increases along with the pitch set in KC. 0=min, 3=max

AR (Attack Rate)

Registers \$80-\$9F, Bits 0-4

Sets the attack rate of the ADSR envelope. 0=slowest, \$1F=fastest

AMS-Enable (Amplitude Modulation Sensitivity Enable)

Registers \$A0-\$BF, Bit 7

If set, the operator's output level will be affected by the LFO according to the channel's AMS setting. If clear, the operator will not be affected.

D1R (Decay Rate 1)

Registers \$A0-\$BF, Bits 0-4

Controls the rate at which the level falls from peak down to the sustain level (D1L). 0=none, \$1F=fastest.

DT2 (Detune 2 - coarse)

Registers \$C0-\$DF, Bits 6-7

Sets a strong detune amount to the operator's frequency. Yamaha suggests that this is most useful for sound effects. 0=off,

D2R (Decay Rate 2)

Registers \$C0-\$DF, Bits 0-4

Sets the Decay2 rate, which takes effect once the level has fallen from peak down to the sustain level (D1L). This rate continues until the level reaches zero or until the note is released.

0=none, \$1F=fastest

D1L

Registers \$E0-\$FF, Bits 4-7

Sets the level at which the ADSR envelope changes decay rates from D1R to D2R. 0=minimum (no D2R), \$0F=maximum (immediately at peak, which effectively disables D1R)

RR

Registers \$E0-\$FF, Bitst 0-3

Sets the rate at which the level drops to zero when a note is released. 0=none, \$0F=fastest

Getting sound out of the YM2151 (a brief tutorial)

While there is a large number of parameters that affect the sound of the YM2151, its operation can be thought of in simplified terms if you consider that there are basically three components to deal with: Instrument configuration (patch), voice pitch selection, and "pressing/releasing" the "key" to trigger (begin) and release (end) notes. It's essentially the same as using a music keyboard. Pressing an instrument button (e.g. Marimba) makes the keyboard sound like a Marimba. Once this is done, you press a key on the keyboard to play a note, and release it to stop the note. With the YM, loading a patch (pressing the Marimba button) entails setting all of the various operators' registers on the voice(s) you want the instrument to be used on. On the music keyboard, pitch and note stop/start are done with a single piano key. In the YM2151, these are two distinct actions.

For this tutorial, we will start with the simplest operation, (triggering notes) and proceed to note selection, and finally patch configuration.

Triggering and Releasing Notes:

Key On/Off (KON) Register (\$08):

This is probably the most important single register in the YM2151. It is used to trigger and release notes. It controls the key on/off state for all 8 channels. A note is triggered whenever its key state changes from off to on, and is released whenever the state changes from on to off. Repeated writes of the same state (off->off or on->on) have no effect.

Whenever an operator is triggered, it progresses through the states of attack, decay1, and sustain/decay2. Whenever an active note is released, it enters the release state where the volume decreases until reaching zero. It then remains silent until the next time the operator is triggered. If you are familiar with the C64 SID chip, this is the same behavior as the "gate" bit on that chip.

Key state and voice selection are both contained in the value written into the KON register as follows:

- Key ON = \$78 + channel number
- Key OFF = \$0 + channel number

Simple Examples:

To release the note in channel 4: write \$08 to `YM_address` (\$9F40) and then write \$04 (\$00+4) to `YM_data` (\$9F41).

To begin a note on channel 7, write \$08 into `YM_address` to select the KON register. Then write \$7F (\$78+7) into `YM_data`

If the current key state of a channel is not known, you can write key off and then key on immediately (after waiting for the YM busy period to end, of course):

```
POKE $9F40,$08 : REM SELECT KEY ON/OFF REGISTER
POKE $9F41,$07 : REM KEY OFF FOR VOICE 7
POKE $9F41,$7F : REM KEY ON FOR VOICE 7
```

Remember: BASIC is slow enough that you do not need to poll the `YM_status` byte, but assembly and other languages will need to do so.

The ADSR parameters will be discussed in more detail later.

Advanced:

Each channel (voice) of the YM2151 uses 4 operators which can be gated together or independently. Independent triggering gives lots of advanced possibilities. To trigger and release operators independently, you use different values than \$78 or \$00. These values are composed by 4 bits which signal the on/off state for each operator.

Suppose a note is playing on channel 2 with all 4 operators active. You can release only the M1 operator by writing \$72 into register \$08.

The KON value format:

7	6	5	4	3	2	1	0
-	C2	M2	C1	M1	Channel		

Pitch Control

YM Registers:

- `KC` = \$28 + channel number
- `KF` = \$30 + channel number

For note selection, each voice has two parameters: `KC` (Key Code) and `KF` (Key Fraction). These are set in register ranges \$28 and \$30, respectively. The KC codes correspond directly to the notes of the chromatic scale. Each value maps to a specific octave & semitone. The `KF` value can even be ignored for basic musical playback. It is mostly useful for vibrato or pitch bend effects. `KF` raises the pitch selected in `KC` in 1/64th increments of the way up to the next semitone.

Like all registers in the YM, whenever a channel's `KC` or `KF` value is written, it takes effect immediately. If a note is playing, its pitch immediately changes. When triggering new notes, it is not important whether you write the pitch or key the note first. This happens quickly in real-time and you will not hear any real difference. Changing the pitch without re-triggering the ADSR envelope is how to achieve pitch slides or a legato effect.

Key Code (KC):

`KC` codes are "conveniently" arranged so that the upper nybble is the octave (0-7) and the lower nybble is the pitch. The pitches are arranged as follows within an octave:

Note	C#	D	D#	E	F	F#	G	G#	A	A#	B	C
Low Nybble (hex)	0	1	2	4	5	6	8	9	A	C	D	E

(Note that every 4th value is skipped.)

Combine the above with an octave to get a note's `KC` value. For instance: concert A (440hz) is (by sheer coincidence) \$4A. Middle C is \$3E, and so forth.

Key Fraction (KF):

`KF` values are written into the top 6 bits of the voice's `KF` register. Basically the value is 0, 1<<2, 2<<2, .. 63<<2

Loading a patch

The patch configuration is by far the most complicated aspect of using the YM. If you take as given that a voice has a patch loaded, then playing notes on it is fairly straightforward. For the moment, we will assume a pre-patched voice.

To get started quickly, here is some BASIC code to patch voice 0 with a marimba tone:

```

5 YA=$9F40 : YD=$9F41 : V=0
10 REM: MARIMBA PATCH FOR YM VOICE 0 (SET V=0..7 FOR OTHER VOICES)
20 DATA $DC, $00, $1B, $67, $61, $31, $21, $17, $1F, $0A, $DF, $5F, $DE
30 DATA $DE, $0E, $10, $09, $07, $00, $05, $07, $04, $FF, $A0, $16, $17
40 READ D
50 POKE YA, $20+V : POKE YD, D
60 FOR A=$38 TO $F8 STEP 8
```

```
70 READ D : POKE YA,A+V : POKE YD,D
80 NEXT A
```

Once a voice has been patched as above, you can now POKE notes into it with very few commands for each note.

Patches consist mostly of ADSR envelope parameters. A complete patch contains values for the \$20 range register (LR|FB|CON), for the \$38 range register (AMS|PMS), and 4 values for each of the parameter ranges starting at \$40. (4 operators per voice means 4 values per parameter). Since this is a huge amount of flexibility, it is recommended to experiment with instrument creation in an application such as a chip tracker or VST, as the creative process of instrument design is very hands-on and subjective.

Using the LFO

There is a single global LFO in the YM2151 which can affect the level (volume) and/or pitch of all 8 channels simultaneously. It has a single frequency and waveform setting which must be shared among all channels, and shared between both phase and amplitude modulation. The global parameters AMD and PMD act as modifiers to the sensitivity settings of the channels. While the frequency and waveform of the LFO pattern must be shared, the depths of the two types of modulation are independent of each other.

You can re-trigger the LFO by setting and then clearing the LR bit in the test register (\$01).

Vibrato:

Use Phase Modulation on the desired channels. The PMS parameter for each channel allows them to vary their vibrato depths individually. Channels with PMS set to zero will have no vibrato. The values given earlier in the PMS parameter description represent their maximum amount of affect. These values are modified by the global PMD. A PMD value of \$7F means 100% effectiveness, \$40 means all channels' vibrato depths will be reduced by half, etc.

The vibrato speed is global, depending solely on the value set to LFRQ.

Amplitude Modulation:

Amplitude modulation works similarly to phase modulation, except that the intensity is a combination of the per-channel AMS value modified by the global AMD value. Additionally, within channels having non-zero amplitude modulation sensitivity, individual operators must have their AMS-en bit enabled in order to be affected by the modulation.

If the active operators are acting as carriers (generating output directly), then amplitude modulation will vary the volume of the sound being produced by that operator. This can be described as a "tremolo" effect. If the operators are acting as modulators, then the timbre of the voice will vary as the output level of the affected operators increases and decreases. You may simultaneously enable amplitude modulation on both types of operators.

The amplitude modulation speed is global, depending solely on the value set to LFRQ.

Chapter 10: Commander X16 Programmer's Reference Guide

I/O Programming

There are two 65C22 "Versatile Interface Adapter" (VIA) I/O controllers in the system, VIA#1 at address \$9F00 and VIA#2 at address \$9F10. The IRQ out lines of VIA#1 is connected to the CPU's NMI line, while the IRQ out line of VIA#2 is connected to the CPU's IRQ line.

The following tables describe the connections of the I/O pins:

VIA#1

Pin	Name	Description
PA0	I2CDATA	I2C Data
PA1	I2CCLK	I2C Clock
PA2	NESLATCH	NES LATCH (for all controllers)
PA3	NESCLK	NES CLK (for all controllers)
PA4	NESDAT3	NES DATA (controller 3)
PA5	NESDAT2	NES DATA (controller 2)
PA6	NESDAT1	NES DATA (controller 1)
PA7	NESDAT0	NES DATA (controller 0)
PB0	<i>Unused</i>	
PB1	<i>Unused</i>	
PB2	<i>Unused</i>	
PB3	SERATNO	Serial ATN out
PB4	SERCLKO	Serial CLK out
PB5	SERDATAO	Serial DATA out
PB6	SERCLKI	Serial CLK in
PB7	SERDATAI	Serial DATA in
CA1	<i>Unused</i>	
CA2	<i>Unused</i>	
CB1	IECSRQ	
CB2	<i>Unused</i>	

The KERNAL uses Timer 2 for timing transmissions on the Serial Bus.

VIA#2

The second VIA is completely unused by the system. All its 16 GPIOs and 4 handshake I/Os can be freely used.

I2C Bus

The Commander X16 contains an I2C bus, which is implemented through two pins of VIA#1. The system management controller (SMC) and the real-time clock (RTC) are connected through this bus. The KERNAL APIs `i2c_read_byte` and `i2c_write_byte` allow talking to these devices.

System Management Controller

The system management controller (SMC) is device \$42 on the I2C bus. It controls the power and activity LEDs, and can be used to power down the system or inject RESET and NMI signals. It also handles communication with the PS/2 keyboard and mouse.

Register	Value	Description
\$01	\$00	Power off
\$01	\$01	Hard reboot
\$02	\$00	Inject RESET
\$03	\$00	Inject NMI
\$04	\$00..\$FF	Power LED brightness
\$05	\$00..\$FF	Activity LED brightness
\$07	-	Read from keyboard buffer
\$08	\$00..\$FF	Echo
\$18	-	Read ps2 status
\$19	\$00..\$FF	Send ps2 command
\$1A	\$0000..\$FFFF	Send ps2 command (2 bytes)
\$20	\$00	Set mouse device ID, standard mouse
\$20	\$03	Set mouse device ID, Intellimouse with scroll wheel
\$20	\$04	Set mouse device ID, Intellimouse with scroll wheel+extra buttons
\$21	-	Read from mouse buffer
\$22	-	Get mouse device ID
\$30	-	Get SMC firmware version, major
\$31	-	Get SMC firmware version, minor
\$32	-	Get SMC firmare version, patch
\$8F	\$31	Start bootloader, if present

Real-Time-Clock

The Commander X16 contains a battery-backed Microchip MCP7940N real-time-clock (RTC) chip as device \$6F. It provide a real-time clock/calendar, two alarms and 64 bytes of RAM.

Register	Description
----------	-------------

\$00	Clock seconds
\$01	Clock minutes
\$02	Clock hours
\$03	Clock weekday
\$04	Clock day
\$05	Clock month
\$06	Clock year
\$07	Control
\$08	Oscillator trim
\$09	<i>reserved</i>
\$0A	Alarm 0 seconds
\$0B	Alarm 0 minutes
\$0C	Alarm 0 hours
\$0D	Alarm 0 weekday
\$0E	Alarm 0 day
\$0F	Alarm 0 month
\$10	<i>reserved</i>
\$11	Alarm 1 seconds
\$12	Alarm 1 minutes
\$13	Alarm 1 hours
\$14	Alarm 1 weekday
\$15	Alarm 1 day
\$16	Alarm 1 month
\$17	<i>reserved</i>
\$18	Power-fail minutes
\$19	Power-fail hours
\$1A	Power-fail day
\$1B	Power-fail month
\$1C	Power-up minutes
\$1D	Power-up hours
\$1E	Power-up day
\$1F	Power-up month

\$20-\$5F	64 Bytes SRAM
-----------	---------------

The second half of the RTC's SRAM (NVRAM) is reserved for use by the KERNAL. \$20-\$3F is available for use by user programs.

For more information, please refer to this device's datasheet.

Chapter 11: Working With CMDR-DOS

This manual describes Commodore DOS on FAT32, aka CMDR-DOS.

CMDR-DOS

Commander X16 duplicates and extends the programming interface used by Commodore's line of disk drives, including the famous (or infamous) VIC-1541. CMDR-DOS uses the industry-standard FAT-32 format. Partitions can be 32MB up to (in theory) 2TB and supports CMD-style partitions, subdirectories, timestamps and filenames up to 255 characters. It is the DOS built into the [Commander X16](#).

There are three basic interfaces for CMDR-DOS: the binary interface (LOAD, SAVE, etc.), the data file interface (OPEN, PRINT#, INPUT#, GET#), and the command interface. We will give a brief summary of BASIC commands here, but please refer to the [BASIC chapter](#) for full syntax of each command.

If you are familiar with the SD2IEC or the CMD hard drive, navigating partitions and subdirectories is similar, with "CD", "MD", and "RD" commands to navigate directories.

Binary Load/Save

The primary use of the binary interface is loading and saving program files and loading binary files into RAM.

Your binary commands are LOAD, SAVE, BLOAD, VLOAD, BVLOAD, VERIFY, and BVERIFY.

This is a brief summary of the LOAD and SAVE commands. For full documentation, refer to [Chapter 3: BASIC Programming](#).

LOAD

```
LOAD <filename> [,device][,secondary_address]  LOAD <filename> [,device][,ram_bank,start_address]
```

This reads a program file from disk. The first two bytes of the file are the memory location to which the file will be loaded, with the low byte first. BASIC programs will start with \$01 \$08, which translates to \$0801, the start of BASIC memory. The device number should be 8 for reading from the SD card.

If using the first form, secondary_address has multiple meanings:

- 0 or not present: load the data to address \$0801, regardless of the address header.
- 1: load to the address specified in the file's header
- 2: load the file headerless to the location \$0801.

If using the second form, ram_bank sets the bank for the load, and start_address is the location to read your data into.

The value of the ram_bank argument only affects the load when the start_address is set in the range of \$A000-\$BFFF.

Examples:

```
LOAD "ROBOTS.PRG", 8, 1  loads the program "ROBOTS.PRG" into memory at the address encoded in the file.
```

```
LOAD "HELLO", 8  loads a program to the start of BASIC at $0801.
```

LOAD "*", 8, 1 loads the first program in the current directory. See the section below on wildcards for more information about using * and ? to access files of a certain type or with unprintable characters.

LOAD "DATA.BIN", 8, 1, \$A000 loads a file into banked RAM, RAM bank 1, starting at \$A000. The first two bytes of the file are skipped. To avoid skipping the first two bytes, use the BLOAD command instead.

SAVE

```
SAVE <filename>[,device]
```

Saves a file from the computer to the SD card. SAVE always reads from the beginning of BASIC memory at \$0801, up to the end of the BASIC program. Device is optional and defaults to 8 (the SD card, or an IEC disk drive, if one is plugged in.)

One word of caution: CMDR-DOS will not let you overwrite a file by default. To overwrite a file, you need to prefix the filename with @:, like this:

```
SAVE "@:DEMO.PRG"
```

BSAVE

```
BSAVE <filename>,<device>,<ram_bank>,<start_address>,<end_address>
```

Saves an arbitrary region of memory to a file without a two-byte header. To allow concatenating multiple regions of RAM into a single file with multiple successive calls to BSAVE, BSAVE allows the use of append mode in the filename string. To make use of this option, the first call to BSAVE can be called normally, which creates the file anew, while subsequent calls should be in append mode to the same file.

Another way to save arbitrary binary data from arbitrary locations is to use the S command in the MONITOR: [Chapter 6: Machine Language Monitor](#).

```
S "filename",8,<start_address>,<end_address>
```

Where and are a 16-bit hexadecimal address.

After a SAVE or BSAVE, the DOS command is implicitly run to show the drive status. The Commodore file I/O model does not report certain failures back to BASIC, so you should double-check the result after a write operation.

```
00, OK,00,00
```

```
READY.
```

An OK reply means the file saved correctly. Any other result is an error that should be addressed:

```
63, FILE EXISTS,00,00
```

CMDR-DOS does not allow files to be overwritten without special handling. If you get FILE EXISTS, either change your file's name or save it with the @: prefix, like this:

```
SAVE "@:HELLO"
```

BLOAD

BLOAD loads a file *without an address header* to an arbitrary location in memory. Usage is similar to LOAD. However, BLOAD does not require or use the 2-byte header. The first byte in the file is the first byte loaded into memory.

```
BLOAD "filename",8,<ram_bank>,<start_address>
```

VLOAD

Read binary data into VERA. VLOAD skips the 2-byte address header and starts reading at the third byte of the file.

```
VLOAD "filename",8,<vram_bank>,<start_address>
```

BVLOAD

Read binary data into VERA without a header. This works like BLOAD, but into VERA RAM.


```
BVLOAD "filename",8,<vram_bank>,<start_address>
```

Sequential Files

Sequential files have two basic modes: read and write. The OPEN command opens a file for reading or writing. The PRINT# command writes to a file, and the GET# and INPUT# commands read from the file.

todo: examples

Command Channel

The command channel allows you to send commands to the CMDR-DOS interface. You can open and write to the command channel using the OPEN command, or you can use the DOS command to issue commands and read the status. While DOS can be used in immediate mode or in a program, only the combination of OPEN/INPUT# can read the command response back into a variable for later processing.

In either case, the ST psuedo-variable will allow you to quickly check the status. A status of 64 is "okay", and any other value should be checked by reading the error channel (shown below.)

To open the command channel, you can use the OPEN command with secondary address 15.

```
10 OPEN 15,8,15
```

If you want to issue a command immediately, add your command string at the end of the OPEN statement:

```
10 OPEN 15,8,15, "CD:/"
```

This example changes to the root directory of your SD card.

To know whether the OPEN command succeeded, you must open the command channel and read the result. To read the command channel (and clear the error status if an error occurred), you need to read four values:

```
20 INPUT#15,A,B$,C,D
```

A is the error number. B\$ is the error message. C and D are unused in CMDR-DOS for most responses, but will return the track and sector when used with a disk drive on the IEC connector.

```
30 PRINT A;B$;C;D
40 CLOSE 15
```

So the entire program looks like:

```
10 OPEN 15,8,15, "CD:/"
20 INPUT#15,A,B$,C,D
30 PRINT A;B$;C;D
40 CLOSE 15
```

If the error number (A) is less than 20, no error occurred. Usually this result is 0 (or 00) for OK.

You can also use the DOS command to send a command to CMDR-DOS. Entering DOS by itself will print the drive's status on the screen. Entering a command in quotes or a string variable will execute the command. We will talk more about the status variable and DOS status message in the next section.

```
DOS
00, 0K, 00, 00
```

```
READY.
DOS "CD:/"
```

The special case of `DOS "$"` will print a directory listing.

```
DOS "$"
```

You can also read the name of the current directory with `DOS "$=C"`

```
DOS "$=C"
```

DOS Features

This is the base features set compared to other Commodore DOS devices:

Feature	1541	1571/1581	CMD HD/FD	SD2IEC	CMDR-DOS
Sequential files	yes	yes	yes	yes	yes
Relative files	yes	yes	yes	yes	not yet
Block access	yes	yes	yes	yes	not yet
Code execution	yes	yes	yes	no	yes
Burst commands	no	yes	yes	no	no
Timestamps	no	no	yes	yes	yes
Time API	no	no	yes	yes	not yet
Partitions	no	no	yes	yes	yes
Subdirectories	no	no	yes	yes	yes

It consists of the following components:

- Commodore DOS interface
 - `dos/main.s` : TALK/LISTEN dispatching
 - `dos/parser.s` : filename/path parsing
 - `dos/cmdch.s` : command channel parsing, status messages
 - `dos/file.s` : file read/write
- FAT32 interface
 - `dos/match.s` : FAT32 character set conversion, wildcard matching
 - `dos/dir.s` : FAT32 directory listing
 - `dos/function.s` : command implementations for FAT32
- FAT32 implementation
 - `fat32/*` : [FAT32 for 65c02 library](#)

All currently unsupported commands are decoded in `cmdch.s` anyway, but hooked into `31, SYNTAX ERROR, 00, 00`, so adding features should be as easy as adding the implementation.

CMDR-DOS implements the TALK/LISTEN layer (Commodore Peripheral Bus layer 3), it can therefore be directly hooked up to the Commodore IEEE KERNAL API (`talk`, `tksa`, `untilk`, `listn`, `secnd`, `unlsn`, `acptr`, `ciout`) and be used as a computer-based DOS, like on the C65 and the X16.

CMDR-DOS does not contain a layer 2 implementation, i.e. IEEE-488 (PET) or Commodore Serial (C64, C128, ...). By adding a Commodore Serial (aka "IEC") implementation, CMDR-DOS could be adapted for use as the system software of a standalone

65c02-based Serial device for Commodore computers, similar to an sd2iec device.

The Commodore DOS side and the FAT32 side are well separated, so a lot of code could be reused for a DOS that uses a different filesystem.

Or the core feature set, these are the supported functions:

Feature	Syntax	Supported	Comment
Reading	,?,R	yes	
Writing	,?,W	yes	
Appending	,?,A	yes	
Modifying	,?,M	yes	
Types	,S/,P/,U/,L	yes	ignored on FAT32
Overwriting	@:	yes	
Magic channels 0/1		yes	
Channel 15 command	<i>command:args...</i>	yes	
Channel 15 status	<i>code,string,a,b</i>	yes	
CMD partition syntax	0:/1:/...	yes	
CMD subdirectory syntax	//DIR//://DIR/:	yes	
Directory listing	\$	yes	
Dir with name filtering	\$:FIL*	yes	
Dir with name and type filtering	\$:*P/\$:*D/\$:*A	yes	
Dir with timestamps	\$=T	yes	with ISO-8601 times
Dir with time filtering	\$=T</\$=T>	not yet	
Dir long listing	\$=L	yes	shows human readable file size instead of blocks, time in ISO-8601 syntax, attribute byte, and exact file size in hexadecimal
Partition listing	\$=P	yes	
Partition filtering	\$:NAME*=P	no	
Current Working Directory	\$=C	yes	

And this table shows which of the standard commands are supported:

Name	Syntax	Description	Supported
------	--------	-------------	-----------

BLOCK-ALLOCATE	B-A <i>medium medium track sector</i>	Allocate a block in the BAM	no ¹
BLOCK-EXECUTE	B-E <i>channel medium track sector</i>	Load and execute a block	not yet
BLOCK-FREE	B-F <i>medium medium track sector</i>	Free a block in the BAM	no ¹
BLOCK-READ	B-R <i>channel medium track sector</i>	Read block	no ¹
BLOCK-STATUS	B-S <i>channel medium track sector</i>	Check if block is allocated	no ¹
BLOCK-WRITE	B-W <i>channel medium track sector</i>	Write block	no ¹
BUFFER-POINTER	B-P <i>channel index</i>	Set r/w pointer within buffer	not yet
CHANGE DIRECTORY	cd[path]:name	Change the current sub-directory	yes
CHANGE DIRECTORY	cd[medium]:-	Change sub-directory up	yes
CHANGE PARTITION	cp num	Make a partition the default	yes
COPY	c[path_a]:target_name=[path_b]:source_name[, ...]	Copy/concatenate files	yes
COPY	cdst_medium=src_medium	Copy all files between disk	no ¹
DUPLICATE	d:dst_medium=src_medium	Duplicate disk	no ¹
FILE LOCK	F-L[path]:name[, ...]	Enable file write-protect	yes
FILE RESTORE	F-R[path]:name[, ...]	Restore a deleted file	not yet
FILE UNLOCK	F-U[path]:name[, ...]	Disable file write-protect	yes
GET DISKCHANGE	G-D	Query disk change	yes
GET PARTITION	G-P[num]	Get information about partition	yes
INITIALIZE	i[medium]	Re-mount filesystem	yes
LOCK	L[path]:name	Toggle file write protect	yes
MAKE DIRECTORY	md[path]:name	Create a sub-directory	yes
MEMORY-EXECUTE	M-E addr_lo addr_hi	Execute code	yes
MEMORY-READ	M-R addr_lo addr_hi [count]	Read RAM	yes
MEMORY-WRITE	M-W addr_lo addr_hi count data	Write RAM	yes
NEW	n[medium]:name,id,FAT32	File system creation	yes ³
PARTITION	/[medium]:name]	Select 1581 partition	no
PARTITION	/[medium]:name,track sector count_lo count_hi ,c	Create 1581 partition	no
POSITION	p channel record_lo record_hi offset	Set record index in REL file	not yet

REMOVE DIRECTORY	<i>rd[path]:name</i>	Delete a sub-directory	yes
RENAME	<i>r[path]:new_name=old_name</i>	Rename file	yes
RENAME-HEADER	<i>r-h[medium]:new_name</i>	Rename a filesystem	yes
RENAME- PARTITION	<i>r-p:new_name=old_name</i>	Rename a partition	no ¹
SCRATCH	<i>s[path]:pattern[, ...]</i>	Delete files	yes
SWAP	<i>s-{8 9 D}</i>	Change primary address	yes
TIME READ ASCII	<i>T-RA</i>	Read Time/Date (ASCII)	no ⁴
TIME READ BCD	<i>T-RB</i>	Read Time/Date (BCD)	no ⁴
TIME READ DECIMAL	<i>T-RD</i>	Read Time/Date (Decimal)	no ⁴
TIME READ ISO	<i>T-RI</i>	Read Time/Date (ISO)	no ⁴
TIME WRITE ASCII	<i>T-WA dow mo/da/yr hr:mi:se ampm</i>	Write Time/Date (ASCII)	no ⁴
TIME WRITE BCD	<i>T-WB b0 b1 b2 b3 b4 b5 b6 b7 b8</i>	Write Time/Date (BCD)	no ⁴
TIME WRITE DECIMAL	<i>T-WD b0 b1 b2 b3 b4 b5 b6 b7</i>	Write Time/Date (Decimal)	no ⁴
TIME WRITE ISO	<i>T-WI yyyy-mm-ddthh:mm:ss dow</i>	Write Time/Date (ISO)	no ⁴
U1/UA	<i>u1 channel medium track sector</i>	Raw read of a block	not yet
U2/UB	<i>u2 channel medium track sector</i>	Raw write of a block	not yet
U3-U8/UC-UH	<i>U3 - U8</i>	Execute in user buffer	not yet
U9/UI	<i>UI</i>	Soft RESET	yes
U:/UJ	<i>UJ</i>	Hard RESET	yes
USER	<i>U0> pa</i>	Set unit primary address	yes
USER	<i>U0>B flag</i>	Enable/disable Fast Serial	no
USER	<i>U0>Dval</i>	Set directory sector interleave	no ¹
USER	<i>U0>H number</i>	Select head 0/1	no ¹
USER	<i>U0>Lflag</i>	Large REL file support on/off	no
USER	<i>U0>M flag</i>	Enable/disable 1541 emulation mode	no ¹
USER	<i>U0>R num</i>	Set number fo retries	no ¹
USER	<i>U0>S val</i>	Set sector interleave	no ¹
USER	<i>U0>T</i>	Test ROM checksum	no ⁵

USER	<code>u0>v flag</code>	Enable/disable verify	no ¹
USER	<code>u0> pa</code>	Set unit primary address	yes
USER	<code>uI{+ -}</code>	Use C64/VIC-20 Serial protocol	no ¹
UTILITY LOADER	<code>&[[path]:]name</code>	Load and execute program	no ¹
VALIDATE	<code>v[medium]</code>	Filesystem check	no ²
WRITE PROTECT	<code>w-{0 1}</code>	Set/unset device write protect	yes

- ¹: outdated API, not useful, or can't be supported on FAT32
- ²: is a no-op, returns `00, 0K, 00, 00`
- ³: third argument `FAT32` has to be passed
- ⁴: CMDR-DOS was architected to run on the main computer, so it shouldn't be DOS that keeps track of the time
- ⁵: Instead of testing the ROM, this command currently verifies that no buffers are allocated, otherwise it halts. This is used by unit tests to detect leaks.

The following special file syntax and `OPEN` options are specific to CMDR-DOS:

Feature	Syntax	Description
Open for Read & Write	<code>,?,M</code>	Allows arbitrarily reading, writing and setting the position (<i>p</i>) ¹
Get current working directory	<code>\$=C</code>	Produces a directory listing containing the name of the current working directory followed by all parent directory names all the way up to /

- ¹: once the EOF has been reached while reading, no further reads or writes are possible.

The following added command channel features are specific to CMDR-DOS:

Feature	Syntax	Description
POSITION	<code>p channel p0 p1 p2 p3</code>	Set position within file (like <code>sd2iec</code>); all args binary

To use the POSITION command, you need to open two channels: a data channel and the command channel. The *channel* argument should be the same as the secondary address of the data channel.

Example

```
OPEN 1,8,2,"LEVEL.DAT,S,R"
OPEN 15,8,15,"P"+CHR$(2)+CHR$(0)+CHR$(1)+CHR$(0)+CHR$(0)
```

This opens LEVEL.DAT for reading and positions the read/write pointer at byte 256.

```
OPEN 2,8,5,"LEVEL.DAT,S,R"
OPEN 15,8,15,"P"+CHR$(5)+CHR$(128)+CHR$(0)+CHR$(0)+CHR$(0)
```

This time, the secondary address is 5, and the pointer is at byte 128.

Current Working Directory

The `$=C` command will list the current working directory and its parent path. The current directory will be at the top of the listing, with each parent directory beneath, with `/` at the bottom.

```
DOS"$=C"

0 "/TEST          "
0  "TEST"         DIR
0  "/"           DIR
65535 BLOCKS FREE.
```

License

Copyright 2020, 2023 Michael Steil < mist64@mac.com >, et al.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 12: Hardware Pinouts

This chapter covers pinout for the I/O ports and headers.

Port and Socket Listing

- VERA Connectors
- SNES Controller Ports (x2)
- IEC Port
- PS/2 Keyboard and mouse
- Expansion Slots (x4 in Gen1)
- User Port Header
- ATX Power Supply
- Front Panel

Chip sockets are not listed; pinouts are available on their respective data sheets.

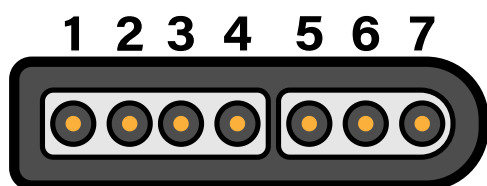
Disclaimer

The instructions and information in this document are the best available information at the time of writing. This information is subject to change, and no warranty is implied. We are not liable for damage or injury caused by use or misuse of this information, including damage caused by inaccurate information. Interfacing and modifying your Commander X16 is done solely at your own risk.

If you attempt to upgrade your firmware and the process fails, one of our community members may be able to help. Please visit the forums or the Discord community, both of which can be reached through <https://commanderx16.com>.

SNES Ports

The computer contains two SNES style ports and will work with Super Nintendo compatible game pads. An on-board pin header is accessible to connect two additional SNES ports.



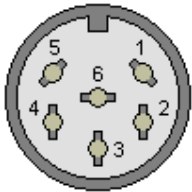
Pin #	Description	Wire Color
1	+5v	White
2	Data Clock	Yellow/Red
3	Data Latch	Orange
4	Serial Data	Red/Yellow
5	N/C	-
6	N/C	-
7	Ground	Brown

The Data Clock and Data Latch are generated by the computer and are shared across all SNES ports. The Serial Data line is unique per controller.

Thanks to [Console Mods Wiki](#)

IEC Port

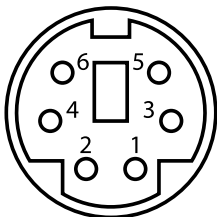
The IEC port is a female 6 pin DIN 45322 connector. The pinout and specifications are the same as the Commodore 128 computer, with the required lines for Fast IEC, as used by the 1571 and 1581 diskette drives. 1541 drives are also compatible, using standard IEC mode at 400-600 bytes/sec.



Pin	Description	Signal Direction	Remark
1	SERIAL SRQ	IN	Serial Service Request In, at the C128 "Fast Serial Clock"
2	GND	-	Ground, signal ground (0V)
3	SERIAL ATN	OUT	Attention, for the selection of a device at beginning/end of a transmission
4	SERIAL CLK	IN/OUT	Clock (for data transmission)
5	SERIAL DATA	IN/OUT	Data
6	SERIAL RESET	OUT(/IN)	Reset

The IEC protocol is beyond the scope of this document. Please see [Wikipedia](#) for more information.

PS/2 Keyboard and Mouse



Pin	Name	Description
1	+DATA	Data
2	NC	Not connected
3	GND	Ground
4	Vcc	+5 VDC
5	+CLK	Clock
6	NC	Not Connected

Expansion Cards / Cartridges

The expansion slots can be used for I/O modules and RAM/ROM cartridges and expose the full CPU address and data bus, plus the ROM bank select lines, stereo audio, and 5 IO select lines.

The expansion/cartridge port is a 60-pin edge connector with 2.54mm pitch. Pin 1 is in the rear-left corner.

Desc	Pin		Pin	Desc
-12V	1	[]	2	+12V
GND	3	[]	4	+5V
AUDIO_L	5	[]	6	GND
AUDIO_R	7	[]	8	ROMB7
IO3	9	[]	10	ROMB0
IO4	11	[]	12	ROMB1
IO7	13	[]	14	ROMB6
IO5	15	[]	16	ROMB2
IO6	17	[]	18	ROMB5
RESB	19	[]	20	ROMB3
RDY	21	[]	22	ROMB4
IRQB	23	[]	24	PHI2
BE	25	[]	26	RWB
NMIB	27	[]	28	MLB
SYNC	29	[]	30	D0
A0	31	[]	32	D1
A1	33	[]	34	D2
A2	35	[]	36	D3
A3	37	[]	38	D4
A4	39	[]	40	D5
A5	41	[]	42	D6
A6	43	[]	44	D7
A7	45	[]	46	A15
A8	47	[]	48	A14
A9	49	[]	50	A13
A10	51	[]	52	A12
A11	53	[]	54	SDA

GND	55	[]	56	SCL
+5V	57	[]	58	GND
+12V	59	[]	60	-12V

To simplify address decoding, pins IO3-IO7 are active for specific, 32-byte memory mapped IO (MMIO) address ranges.

Address	Usage	Speed
\$9F60-\$9F7F	Expansion Card Memory Mapped IO3	8 MHz
\$9F80-\$9F9F	Expansion Card Memory Mapped IO4	8 MHz
\$9FA0-\$9FBF	Expansion Card Memory Mapped IO5	2 MHz
\$9FC0-\$9FDF	Expansion Card Memory Mapped IO6	2 MHz
\$9FE0-\$9FFF	Cartidge/Expansion Memory Mapped IO7	2 MHz

Expansion cards can use the IO3-IO6 lines as enable lines to provide their IO address range (s), or decode the address from the address bus directly. To prevent conflicts with other devices, expansion boards should allow the user to select their desired I/O bank with jumpers or DIP switches. IO7 is given priority to external cartridges that use MMIO and should be only used by an expansion card if there are no other MMIO ranges available. Doing so may cause a bus conflict with cartridges that make use of MMIO (such as those with expansion hardware). See below for more information on cartridges.

ROMB0-ROMB7 are connected to the ROM bank latch at address \$01. Values 0-31 (\$00 - \$1F) address the on-board ROM chips, and 32-255 are intended for expansion ROM or RAM chips (typically used by cartridges, see below). This allows for a total of 3.5MB of address space in the \C000-\FFFF address range.

SCL and SDA pins are shared with the i2c connector on J9 and can be used to access i2c peripherals on cartridges or expansion cards.

AUDIO_L and AUDIO_R are routed to J10, the audio option header.

The other pins are connected to the system bus and directly to the 65C02 processor.

Cartridges

Cartridges are essentially an expansion card housed in an external enclosure. Typically they are used for applications (e.g. games) with the X16 being able to boot directly from a cartridge at power on. Typically they contain a mix of banked ROM and/or RAM and an optional I2C EEPROM (for storing game save states).

They can also function as an expansion card which means they can also use MMIO. Similarly an internal expansion card could contain RAM/ROM as well.

Because of this, while developers are free to use the hardware as they please, to avoid conflicts, the banked ROM/RAM space is suggested to be used only by cartridges and cartridges should avoid using MMIO IO3-IO6. Instead, IO7 should be the default option for cartridges and the last option for expansion cards (only used if there are no other IO ranges available).

This helps avoid bus conflicts and an otherwise bad user experience given a cartridge should be simple to use from the standpoint of the user ("insert game -> play game").

These are soft guidelines. There is nothing physically preventing an expansion card from using banked ROM/RAM or a cartridge using any of the MMIO addresses. Doing so risks conflicts and compatibility issues.

Cartridges with additional hardware would be similar to expansion chips found on some NES and SNES cartridges (think VRC6, Super FX, etc.) and could be used for really anything, such as having a MIDI input for a cartridge that is meant as a music maker; some sort of hardware accelerator FPGA; network support, etc.

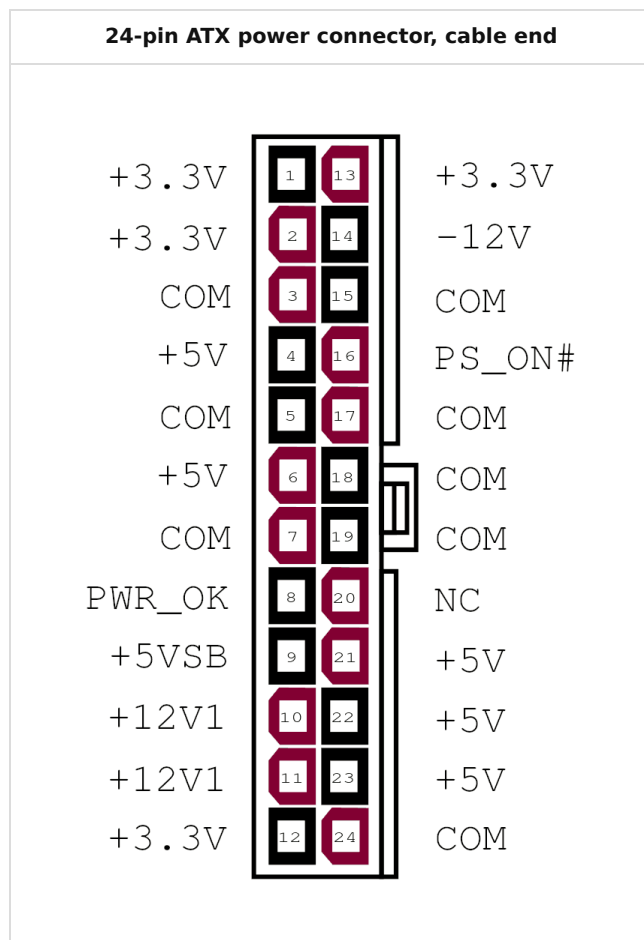
For more information about the memory map visit the [Memory Map](#) section of the manual.

Booting from Cartridges

After the X16 finishes it's hardware initialization, the kernel checks bank 32 for the signature "CX16" at \$C000 . If found, it then jumps to \$C004 and leaves interrupts disabled.

ATX Power Supply

The Commander X16 has a socket for an industry standard 24-pin ATX power supply connector. Either a 24-pin or 20-pin PSU connector can be plugged in, though only the pins for the older 20-pin standard are used by the computer. You don't need an expensive power supply, but it must supply the -12v rail. Not all do, so check your unit to make sure. If you can't tell from the label, you can check Pin 12 and COM. If the clip side is facing away from you, pin 14 will be the second pin on the left on the clip side. For a 20-pin cable, -12v is on pin 12, but at the same relative position — the second pin on the left on the clip side.



By CalvinTheMan - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=50881708>

The Commander X16 does not use the 4-pin CPU power, GPU power, 4-pin drive power, or SATA power connectors.

To save space, when running a bare motherboard, we recommend a "Pico PSU" power supply, which derives all of the necessary power lines from a single 12V source.

J1 ROM Write Protect

Remove J1 to write protect system ROM. With J1 installed, users can program the system ROM using an appropriate ROM flash program.

J2 NMI

Connect a button here to generate an Non Maskable Interrupt (NMI) on the CPU. This will execute a BASIC warm start, which will stop any existing program, clear the screen, and print the READY prompt.

J3 (Unknown)

Connect J8 for LPT Compat. (TODO: Is this the Centronics parallel port mode Lorin hinted at early on?)

J4 Extra 65C22 Pins

Desc	Pin		Pin	Desc
CA1	1	..	2	CA2
PB0	3	..	4	PB1
PB2	5	..	6	CB2

These pins are connected to VIA 1 at \$9F00-\$9F0F.

J5 Program Microcontroller

Remove jumpers from J5 to program microcontroller.

J6 System Speed

Pin	Desc
1 - 2	8 MHz
3 - 4	4 MHz
5 - 6	2 MHz

J7 SNES 3/4

Desc	Pin		Pin	Desc
CLC	1	..	2	VCC
LATCH	3	..	4	DAT4
DAT3	5	..	6	GND

These pins will allow for two additional SNES controllers, for a total of four controllers on the system.

J8 Front Panel

Desc	Pin		Pin	Desc
HDD LED+	1	..	2	POW LED +
HDD LED-	3	..	4	POW LED -
RESET BUT	5	..	6	POW BUT
RESET BUT	7	..	7	POW BUT

+5VDC	9	..	10	NC
-------	---	----	----	----

This pinout is compatible with newer ATX style motherboards. AT motherboards and older ATX cases may still have a 3-pin power LED connector (with a blank pin in the middle.) You will need to move the + (red) wire on the power LED connector to the center pin, if this is the case. Or you can use two Male-Female breadboard cables to jumper the header to your power LED connector.

There is no on-board speaker header. Instead, all audio is routed to the rear panel headphone jack via the Audio Option header.

J9 I2C/SMC Header

Desc	Pin		Pin	Desc
SMC MOSI/I2C SDA	1	..	2	5V STANDBY
RTC MFP	3	..	4	SMC TX
SMC Reset	5	..	6	SMC RX
SMC SCK/I2C SCL	7	..	8	GND
SMC MISO	9	..	10	GND

J10 Audio Option

Desc	Pin		Pin	Desc
SDA	1	..	2	RESB
SCL	3	..	4	VCC
	5	..	6	
+12V	7	..	8	-12V
	9	..	10	
VERA_L	11	..	12	BUS_L
	13	..	14	
VERA_R	15	..	16	BUS_R
	17	..	18	
YM_L	19	..	20	OUT_L
	21	..	22	
YM_R	23	..	24	OUT_R

5,6,9,10,13,14,17,18,21,22 - GND

Next to the audio header is a set of jumper pads, JP1-JP6. Cutting these traces allows you to extract isolated audio from each of the system devices or build a mixer to adjust the relative balance of the audio devices.

In order to avoid ground loop and power supply noise, we recommend installing a ground loop isolator when using an external mixer. 2 or 3 isolators will be required (one for each stereo pair.) (TODO: measure noise and test with pro audio gear.)

J12 User Port

Desc	Pin		Pin	Desc
PB0	1	..	2	PB4
PA0	3	..	4	PB5
PA1	5	..	6	PB6/CB1
PA2	7	..	8	PB7/CB2
PA3	9	..	10	GND
PA4	11	..	12	GND
PA5	13	..	14	GND
PA6	15	..	16	GND
PA7	17	..	18	GND
CA1	19	..	20	GND
PB1	21	..	22	GND
PB2	23	..	24	GND
PB3/CA2	25	..	16	VCC

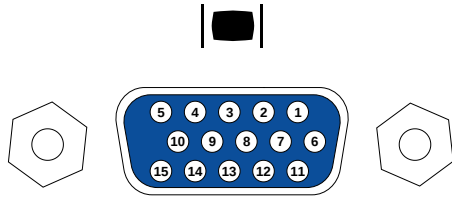
User port is connected to VIA 2 at address \$9F10-\$9F1F. This can be used for serial or parallel port I/O. Commander X16 does not have support for a serial port device in the KERNAL.

VERA Video Header

Desc	Pin		Pin	Desc
VCC	1	..	2	GND
D7	3	..	4	D6
D5	5	..	6	D4
D3	7	..	8	D2
D1	9	..	10	D0
/IO1	11	..	12	RESB
/MEMWE	13	..	14	IRQB
A4	15	..	16	/MEMOE
A2	17	..	18	A3
A0	19	..	20	A1
GND	21	..	22	GND
VERA_L	23	..	24	VERA_R

VERA is connected to I/O ports at \$9F20-\$9F3F. See [VERA Programmer's Reference](#) for details.

VGA Connector



Pin	Desc
1	RED
2	GREEN
3	BLUE
4	
5	GND
6	RED_RTN
7	GREEN_RTN
8	BLUE_RTN
9	
10	GND
11	
12	
13	HSync
14	VSyn
15	

The VGA connector is a female [DE-15](#) jack.

The video resolution is 640x480 59.5FPS progressive scan, RGB color, and separated H/V sync.

In interlace mode, both horizontal and vertical sync pulses will appear on the HSync pin. (TODO: Test with OSSC).

VERA does not use the ID/DDC lines.

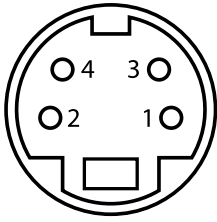
Composite Connector

The Composite video is a standard RCA connector. Center pin carries signal. Shield is signal ground.

The signal is NTSC Composite baseband video.

The video is 480 lines 59.97Hz interlaced. Composite is not available when VGA is running at 59.5Hz progressive scan.

S-Video Connector



Pin	Desc	
1	GND (Y)	
2	GND (C)	
3	Y	Intensity (Luminance)
4	C	Color (Chrominance)

The connector is a 4-pin Mini-DIN connector. While the same size as a PS/2 connector, the PS/2 connector has a plastic key at the bottom. Do not attempt to plug a keyboard or mouse into the S-Video port, or bent pins will occur.

The signal is NTSC baseband Y/C separated video. S-Video provides better resolution than composite, since the color and intensity are provided on separate pins. you can use a splitter cable to separate the Y and C signals to drive a Commodore 1702 or compatible monitor.

The video is 480 lines 59.97Hz interlaced. Composite is not available when VGA is running at 59.5Hz progressive scan.

J2 VERA Programming Interface

Pin	Desc
1	+5V
2	FPGA_CDONE
3	FPGA_CRESET_B
4	SPI_MISO
5	SPI_MOSI
6	SPI_SCK
7	SPI_SSEL_N
8	GND

VERA J7 Remote SD Card Option

Pin	Desc
1	CS
2	SCK
3	MOSI
4	MISO

5	+5V
6	GND

This requires an EEPROM programmer and an interface board to program. See [chapter 13](#) for the programming adapter and instructions.

Chapter 13: Upgrade Guide

This chapter provides tips for running upgrades on the various programmable chips.

WARNING: flashing any of these components has a risk of leading to an unbootable system. At the current time, doing hardware flash updates requires skill and knowledge beyond that of an ordinary end user and is not recommended without guidance from the community on the Commander X16 Discord.

Under the headings of each component is a matrix which indicates which software tools can be used to perform the flash of that component, depending on which flashing hardware you have access to and the operating system of the computer you have the device connected to. Some components of the Commander X16 can be self-flashed, but the risk of a failed flash rendering your X16 unbootable is high, in which case an external programmer must be used to flash the component and thus "unbrick" the system.

Flashable components

- System ROM
- SMC (PS/2 and Power controller)
- VERA

System ROM

Official community system ROMs will be posted as releases at [X16Community/x16-emulator](#) inside the distribution for the Emulator.

TODO: link to instructions for each solution in the matrix

↓ Hardware / OS →	Windows	Linux	Mac OS	Commander X16
Commander X16	-	-	-	x16-flash
XGecu TL866II+	Xgpro	minipro	minipro	-
XGecu TL866-3G / T48	Xgpro	-	-	-

SMC

Official community SMC ROMs will be posted as releases at [X16Community/x16-smc](#).

TODO: link to instructions for each solution in the matrix

↓ Hardware / OS →	Windows	Linux	Mac OS	Commander X16
Commander X16	-	-	-	-
USBtinyISP	arduino	arduino	arduino	-
XGecu TL866II+	Xgpro	-	-	-
XGecu TL866-3G / T48	Xgpro	-	-	-

VERA

TODO: link to instructions for each solution in the matrix

Official community VERA bitstreams will be posted as releases at [X16Community/vera-module](#)

↓ Hardware / OS →	Windows	Linux	Mac OS	Commander X16
Commander X16	-	-	-	flashvera
XGecu TL866II+	Xgpro	minipro	minipro	-
XGecu TL866-3G / T48	Xgpro	-	-	-

Appendix A: Sound

FM instrument patch presets

#	Instrument Name	#	Instrument Name
0	Acoustic Grand Piano	64	Soprano Sax †
1	Bright Acoustic Piano	65	Alto Sax †
2	Electric Grand Piano	66	Tenor Sax †
3	Honky-tonk Piano	67	Baritone Sax
4	Electric Piano 1	68	Oboe †
5	Electric Piano 2	69	English Horn †
6	Harpsichord	70	Bassoon
7	Clavinet	71	Clarinet †
8	Celesta	72	Piccolo
9	Glockenspiel	73	Flute †
10	Music Box	74	Recorder
11	Vibraphone †	75	Pan Flute
12	Marimba	76	Blown Bottle
13	Xylophone	77	Shakuhachi
14	Tubular Bells	78	Whistle †
15	Dulcimer	79	Ocarina
16	Drawbar Organ †	80	Lead 1 (Square) †
17	Percussive Organ †	81	Lead 2 (Sawtooth) †
18	Rock Organ †	82	Lead 3 (Triangle) †
19	Church Organ	83	Lead 4 (Chiff+Sine) †
20	Reed Organ	84	Lead 5 (Charang) †
21	Accordion	85	Lead 6 (Voice) †
22	Harmonica	86	Lead 7 (Fifths) †
23	Bandoneon	87	Lead 8 (Solo) †
24	Acoustic Guitar (Nylon)	88	Pad 1 (Fantasia) †
25	Acoustic Guitar (Steel)	89	Pad 2 (Warm) †
26	Electric Guitar (Jazz)	90	Pad 3 (Polysynth) †
27	Electric Guitar (Clean)	91	Pad 4 (Choir) †

28	Electric Guitar (Muted)	92	Pad 5 (Bowed)
29	Electric Guitar (Overdriven)	93	Pad 6 (Metallic)
30	Electric Guitar (Distortion)	94	Pad 7 (Halo) †
31	Electric Guitar (Harmonics)	95	Pad 8 (Sweep) †
32	Acoustic Bass	96	FX 1 (Raindrop)
33	Electric Bass (finger)	97	FX 2 (Soundtrack) †
34	Electric Bass (picked)	98	FX 3 (Crystal)
35	Fretless Bass	99	FX 4 (Atmosphere) †
36	Slap Bass 1	100	FX 5 (Brightness) †
37	Slap Bass 2	101	FX 6 (Goblin)
38	Synth Bass 1	102	FX 7 (Echo)
39	Synth Bass 2	103	FX 8 (Sci-Fi) †
40	Violin †	104	Sitar
41	Viola †	105	Banjo
42	Cello †	106	Shamisen
43	Contrabass †	107	Koto
44	Tremolo Strings †	108	Kalimba
45	Pizzicato Strings	109	Bagpipe
46	Orchestral Harp	110	Fiddle †
47	Timpani	111	Shanai †
48	String Ensemble 1 †	112	Tinkle Bell
49	String Ensemble 2 †	113	Agogo
50	Synth Strings 1 †	114	Steel Drum
51	Synth Strings 2 †	115	Woodblock
52	Choir Aahs †	116	Taiko Drum
53	Voice Doos	117	Melodic Tom
54	Synth Voice †	118	Synth Drum
55	Orchestra Hit	119	Reverse Cymbal
56	Trumpet †	120	Fret Noise
57	Trombone	121	Breath Noise
58	Tuba	122	Seashore †
59	Muted Trumpet †	123	Bird Tweet

60	French Horn		124	Telephone Ring
61	Brass Section		125	Helicopter
62	Synth Brass 1		126	Applause †
63	Synth Brass 2		127	Gunshot

† Instrument is affected by the LFO, giving it a vibrato or tremolo effect.

FM extended instrument patch presets

These presets exist mainly to support playback of drum sounds, and many of them only work correctly or sound musical at certain pitches or within a small range of pitches.

#	Instrument Name		#	Instrument Name
128	Silent		146	Vibraslap
129	Snare Roll		147	Bongo
130	Snap		148	Maracas
131	High Q		149	Short Whistle
132	Scratch		150	Long Whistle
133	Square Click		151	Short Guiro
134	Kick		152	Long Guiro
135	Rim		153	Mute Cuica
136	Snare		154	Open Cuica
137	Clap		155	Mute Triangle
138	Tom		156	Open Triangle
139	Closed Hi-Hat		157	Jingle Bell
140	Pedal Hi-Hat		158	Bell Tree
141	Open Hi-Hat		159	Mute Surdo
142	Crash		160	Pure Sine
143	Ride Cymbal		161	Timbale
144	Splash Cymbal		162	Open Surdo
145	Tambourine			

Drum presets

These are the percussion instrument mappings for the drum number argument of the `ym_playdrum` and `ym_setdrum` API calls, and the `FMDRUM` BASIC command.

#	Instrument Name		#	Instrument Name
---	-----------------	--	---	-----------------

			56	Cowbell
25	Snare Roll		57	Crash Cymbal 2
26	Finger Snap		58	Vibraslap
27	High Q		59	Ride Cymbal 2
28	Slap		60	High Bongo
29	Scratch Pull		61	Low Bongo
30	Scratch Push		62	Mute High Conga
31	Sticks		63	Open High Conga
32	Square Click		64	Low Conga
33	Metronome Bell		65	High Timbale
34	Metronome Click		66	Low Timbale
35	Acoustic Bass Drum		67	High Agogo
36	Electric Bass Drum		68	Low Agogo
37	Side Stick		69	Cabasa
38	Acoustic Snare		70	Maracas
39	Hand Clap		71	Short Whistle
40	Electric Snare		72	Long Whistle
41	Low Floor Tom		73	Short Guiro
42	Closed Hi-Hat		74	Long Guiro
43	High Floor Tom		75	Claves
44	Pedal Hi-Hat		76	High Woodblock
45	Low Tom		77	Low Woodblock
46	Open Hi-Hat		78	Mute Cuica
47	Low-Mid Tom		79	Open Cuica
48	High-Mid Tom		80	Mute Triangle
49	Crash Cymbal 1		81	Open Triangle
50	High Tom		82	Shaker
51	Ride Cymbal 1		83	Jingle Bell
52	Chinese Cymbal		84	Belltree
53	Ride Bell		85	Castanets
54	Tambourine		86	Mute Surdo
55	Splash Cymbal		87	Open Surdo

BASIC FMPLAY and PSGPLAY string macros

Overview

The play commands use a string of tokens to define sequences of notes to be played on a single voice of the corresponding sound chip. Tokens cause various effects to happen, such as triggering notes, changing the playback speed, etc. In order to minimize the amount of text required to specify a sequence of sound, the player maintains an internal state for most note parameters.

Stateful Player Behavior:

Playback parameters such as tempo, octave, volume, note duration, etc do not need to be specified for each note. These states are global between all voices of both the FM and PSG sound chips. The player maintains parameter state during and after playback. For instance, setting the octave to 5 in an `FMPLAY` command will result in subsequent `FMPLAY` and `PSGPLAY` statements beginning with the octave set to 5.

The player state is reset to default values whenever `FMINIT` or `PSGINIT` are used.

Parameter	Default	Equivalent Token
Tempo	120	T120
Octave	4	O4
Length	4	L4
Note Spacing	1	S1

Using Tokens:

The valid tokens are: `A-G, I, K, L, O, P, R, S, T, V, <, > .`

Each token may be followed by optional modifiers such as numbers or symbols. Options to a token must be given in the order they are expected, and must have no spacing between them. Tokens may have spaces between them as desired. Any unknown characters are ignored.

Example:

```
FMPLAY 0, "L4"      : REM DEFAULT LENGTH = QUARTER NOTE
FMPLAY 0, "A2. C+." : REM VALID
FMPLAY 0, "A.2 C.+ " : REM INVALID
```

The valid command plays A as a dotted half, followed by C# as a dotted quarter.

The invalid example would play A as a dotted quarter (not half) because length must come before dots. Next, it would ignore the 2 as garbage. Then it would play natural C (not sharp) as a dotted quarter. Finally, it would ignore the + as garbage, because sharp/flat must precede length and dot.

Token definitions:

Musical notes

- Synopsis: Play a musical note, optionally setting the length.
- Syntax: `<A-G>[<+/->][<length>][.]`

Example:

```
FMPLAY 0, "A+2A4C.G-8."
```

On the YM2151 using channel 0, plays in the current octave an **A**[#] [half note](#)[?] followed by an **A** [quarter note](#)[?], followed by **C** dotted quarter note, followed by **G**_b dotted [eighth note](#)[?].

Lengths and dots after the note name or rest set the length just for the current note or rest. To set the default length for subsequent notes and rests, use the `L` macro.

Rests

- Synopsis: Wait for a period of silence equal to the length of a note, optionally setting the length.
- Syntax: `R[<length>][.]`

Example:

```
PSGPLAY 0, "CR2DRE"
```

On the VERA PSG using voice 0, plays in the current octave a **C** quarter note, followed by a half rest (silence), followed by a quarter **D**, followed by a quarter rest (silence), and finally a quarter **E**.

The numeral `2` in `R2` sets the length for the `R` itself but does not alter the default note length (assumed as 4 - quarter notes in this example).

Note Length

- Synopsis: Set the default length for notes and rests that follow
- Syntax: `L[<length>][.]`

Example values:

- L4 = quarter note (crotchet)
- L16 = sixteenth note (semiquaver)
- L12 = 8th note triplets (quaver triplet)
- L4. = dotted quarter note (1.5x the length)
- L4.. = double-dotted quarter note (1.75x the length)

Example program:

```
10 FMPLAY 0, "L4"
20 FOR I=1 TO 2
30 FMPLAY 0, "CDECL8"
40 NEXT
```

On the YM2151 using channel 0, this program, when RUN, plays in the current octave the sequence **C D E C** first as quarter notes, then as eighth notes the second time around.

Articulation

- Synopsis: Set the spacing between notes, from legato to extreme staccato
- Syntax: `S<0-7>`

`S0` indicates legato. For FMPLAY, this also means that notes after the first in a phrase don't implicitly retrigger.

`S1` is the default value, which plays a note for 7/8 of the duration of the note, and releases the note for the remaining 1/8 of the note's duration.

You can think of `S` is, out of 8, how much space is put between the notes.

Example:

```
FMPLAY 0, "L4S1CDES0CDES4CDE"
```

On the YM2151 using channel 0, plays in the current octave the sequence **C D E** three times, first with normal articulation, next with legato (notes all run together and without retriggering), and finally with a moderate staccato.

Explicit retrigger

- Synopsis: on the YM2151, when using `S0` legato, retrigger on the next note.
- Syntax: `K`

Example:

```
FMPLAY 0, "S0CDEKFGA"
```

On the YM2151 using channel 0, plays in the current octave the sequence **C D E** using legato, only triggering on the first note, then the sequence **F G A** the same way. The note **F** is triggered without needing to release the previous note early.

Octave

- Synopsis: Explicitly set the octave number for notes that follow
- Syntax: `0<0-7>`

Example:

```
PSGPLAY 0, "04A02A06CDE"
```

On the VERA PSG using voice 0, changes to octave 4 and plays **A** (440Hz), then switches to octave 2, and plays **A** (110Hz), then switches to octave 6 and plays the sequence **C D E**

Octave Up

- Synopsis: Increases the octave by 1
- Syntax: `>`

If the octave would go above 7, this macro has no effect.

Example:

```
PSGPLAY 0, "04AB>C+DE"
```

On the VERA PSG using voice 0, changes to octave 4 and plays the first five notes of the **A major** scale by switching to octave 5 starting at the **C#**

Octave Down

- Synopsis: Decreases the octave by 1
- Syntax: `<`

If the octave would go below 0, this macro has no effect. Example:

```
PSGPLAY 0, "05GF+EDC<BAG"
```

On the VERA PSG using voice 0, changes to octave 5 and plays the **G major** scale from the top down by switching to octave 4 starting at the **B**

Tempo

- Synopsis: Sets the BPM, the number of quarter notes per minute
- Syntax: `T<1-255>`

High tempo values and short notes tend to have inaccurate lengths due to quantization error. Delays within a string do keep track of fractional frames so the overall playback length should be relatively consistent.

Low tempo values that cause delays (lengths) to exceed 255 frames will also end up being inaccurate. For very long notes, it may be better to use legato to string several together.

Example:

```
10 FMPLAY 0,"T120C4CGGAAGR"
20 FMPLAY 0,"T180C4CGGAAGR"
```

On the YM2151 using channel 0, plays in the current octave the first 7 notes of *Twinkle Twinkle Little Star*, first at 120 beats per minute, then again 1.5 times as fast at 180 beats per minute.

Volume

- Synopsis: Set the channel or voice volume
- Syntax: `V<0-63>`

This macro mirrors the `PSGVOL` and `FMVOL` BASIC commands for setting a channel or voice's volume. 0 is silent, 63 is maximum volume.

Example:

```
FMPLAY 0,"V40ECV45ECV50ECV55ECV60ECV63EC"
```

On the YM2151 using channel 0, starting at a moderate volume, plays the sequence **E C**, repeatedly, increasing the volume steadily each time.

Panning

- Synopsis: Sets the stereo output of a channel or voice to left, right, or both.
- Syntax: `P<1-3>`

1 = Left
2 = Right
3 = Both

Example:

```
10 FOR I=1 TO 4
20 PSGPLAY 0,"P1CP2B+"
30 NEXT I
40 PSGPLAY 0,"P3C"
```

On the VERA PSG using voice 0, in the current octave, repeatedly plays a **C** out of the left speaker, then a **B#** (effectively a **C** one octave higher) out of the right speaker. After 4 such loops, it plays a **C** out of both speakers.

Instrument change

- Synopsis: Sets the FM instrument (like `FMINST`) or PSG waveform (like `PSGWAV`)
- Syntax: `I<0-255>` (0-162 for FM)

Note: This macro is available starting in ROM version R43.

Example:

```
10 FMINIT  
20 FMVIB 200,15  
30 FMCHORD 0,"I11CI11EI11G"
```

This program sets up appropriate vibrato/tremolo and plays a C major chord with the vibraphone patch across FM channels 0, 1, and 2.