APPENDIX I

KERNAL Jump Table

USING THE KERNAL JUMP TABLE

The *KERNAL* is the centerpiece of the MEGA65 operating system. It is responsible for setting up the operating system when the computer is switched on or reset, interfacing with peripherals, invoking disk operations, managing input and output streams, and driving key features of the screen editor and BASIC. While it is running, the KERNAL owns the CPU memory map and interrupt handlers, and reserves regions of memory to support its features.

The KERNAL provides an *Application Programming Interface* (API) so that machine code programs can take advantage of its features, such as to read a file from disk, by calling subroutines. A program accesses these subroutines by calling addresses in a *jump table*, a region of ROM code that is guaranteed by the API to always access the same subroutines in the same way, even across different versions of the ROM code. The program uses `jsr` or `jmp` (as required by the routine) using the jump table address as an absolute target. Memory at the jump table address contains a `jmp` instruction to the appropriate internal address.

For example, to write a character to the output stream (the screen terminal, by default), a program sets the accumulator (A) register to the PETSCII code to write, then calls the CHROUT subroutine with the jump table address $FFD2:

```
lda #$41
jsr $ffd2
```

Prerequisites of Jump Table Routines

All KERNAL jump table routines expect the following conditions to be true when called:

1. The 16-bit address range $E000 – $FFFF must be mapped to the KERNAL ROM at 3.E000 – 3.FFFF.
2. The 16-bit address range $0000 – $1FFF must be mapped to KERNAL internal variable memory at 0.0000 – 0.1FFF.
3. The CPU base page register (B) must be $00.

Some jump table routines have additional prerequisites. See the reference below.

The KERNAL BASIC and SYS memory maps maintain these conditions by default. If a machine code program does not change the MAP or B registers, it can call KERNAL jump table routines without additional set-up.

The KERNAL makes use of other regions of memory. See *the MEGA65 Book*, **??** (**??**).

Unless otherwise noted, a program must assume that a call to a KERNAL routine overwrites data in all CPU registers and value-related status flags. Implications for the stack and mode-related status flags (interrupts) are noted explicitly.

History and Compatibility

The MEGA65 KERNAL jump table is based on, and compatible with, the Commodore 65 jump table. This API is derived from, but *not* entirely compatible with, the KERNAL APIs of the Commodore 64 and Commodore 128.

This documentation adopts the names of the routines from Commodore 65 documentation where possible, and invents unique names for routines new to the MEGA65. A routine with a name similar to C64 or C128 documentation may *not* be API compatible. Differences are noted in this reference, where known.

The MEGA65's "GO64" mode uses the C64 KERNAL. Refer to a C64 programming reference for information on how to use the C64 KERNAL.

For a complete history of the KERNAL jump table across all Commodore computers, see: `https://www.pagetable.com/?p=926`

MEGA65 KERNAL JUMP TABLE QUICK REFERENCE

The following is a quick reference for the MEGA65 KERNAL jump table, in address order. See the full reference for details and examples.

Address	Category	Name	Description
$FF41	I/O	GETIO	Read the current input and output devices
$FF44	Files	GETLFS	Read file, device, secondary address
$FF47	Editor	KEYLOCKS	Read/set keyboard locks
$FF4A	Editor	ADDKEY	Add a character to the soft keyboard input buffer
$FF4D	Reserved	SPIN_SPOUT	Set up fast serial ports
$FF50	Files	CLOSE_ALL	Close all files on a device
$FF53	OS	C64MODE	Reset to GO64 mode
$FF56	OS	MonitorCall	Invoke monitor
$FF59	OS	BOOT_SYS	Boot an alternate system from disk
$FF5C	OS	PHOENIX	Call cartridge cold start or disk boot loader
$FF5F	Files	LKUPLA	Search for logical file number in use
$FF62	Files	LKUPSA	Search for secondary address in use
$FF65	Editor	SWAPPER	Toggle between 40x25 and 80x25 text modes
$FF68	Editor	PFKEY	Program an editor function key
$FF6B	Files	SETBNK	Set bank for I/O and filename memory
$FF6E	Memory	JSRFAR	Call subroutine in any bank
$FF71	Memory	JMPFAR	Jump to address in any bank
$FF74	Memory	LDA_FAR	Read a byte from an address in any bank
$FF77	Memory	STA_FAR	Store a byte to an address in any bank
$FF7A	Memory	CMP_FAR	Compare a byte with an address in any bank
$FF7D	Tools	PRIMM	Print an inline null-terminated short string
$FF81	Editor	CINT	Initialize screen editor
$FF84	OS	IOINIT	Initialize I/O devices
$FF87	OS	RAMTAS	Initialize RAM and buffers
$FF8A	OS	RESTOR	Initialize KERNAL vector table
$FF8D	OS	VECTOR	Read/set KERNAL vector table
$FF90	OS	SETMSG	Enable/disable KERNAL messages
$FF93	Serial	SECND	Send secondary address to listener
$FF96	Serial	TKSA	Send secondary address to talker

continued ...

...continued

Address	Category	Name	Description
$FF99	Reserved	MEMTOP	RESERVED, DO NOT USE
$FF9C	Reserved	MEMBOT	RESERVED, DO NOT USE
$FF9F	Tools	KEY	Scan keyboard
$FFA2	OS	MONEXIT	Monitor's exit to BASIC
$FFA5	Serial	ACPTR	Accept a byte from talker
$FFA8	Serial	CIOUT	Send a byte to listener
$FFAB	Serial	UNTLK	Send "untalk" command
$FFAE	Serial	UNLSN	Send "unlisten" command
$FFB1	Serial	LISTN	Send "listen" command
$FFB4	Serial	TALK	Send "talk" command
$FFB7	I/O	READSS	Get status of last I/O operation
$FFBA	Files	SETLFS	Set file, device, secondary address
$FFBD	Files	SETNAM	Set filename pointers
$FFC0	Files	OPEN	Open logical file
$FFC3	Files	CLOSE	Close logical file
$FFC6	I/O	CHKIN	Set input channel
$FFC9	I/O	CKOUT	Set output channel
$FFCC	I/O	CLRCH	Restore default channels
$FFCF	I/O	BASIN	Read a character from input device
$FFD2	I/O	BSOUT	Write a character to output device
$FFD5	Files	LOAD	Load/verify from file
$FFD8	Files	SAVE	Save to file
$FFDB	Tools	SETTIM	Set CIA1 24-hour clock
$FFDE	Tools	RDTIM	Read CIA1 24-hour clock
$FFE1	Tools	STOP	Report Stop key (see ScanStopKey)
$FFE4	I/O	GETIN	Read a character from input device, without waiting
$FFE7	Files	CLALL	Close all files and channels
$FFEA	Tools	ScanStopKey	Scan Stop key
$FFED	Editor	SCRORG	Get current screen window size
$FFF0	Editor	PLOT	Read/set cursor position
$FFF3	Reserved	IOBASE	RESERVED, DO NOT USE

MEGA65 KERNAL JUMP TABLE REFERENCE

ACPTR

Address: JSR $FFA5

Description:

Accept a byte from talker

Outputs: **A** The accepted byte

Remarks: This is a low-level serial routine. Most programs will prefer the higher level I/O routines (OPEN et al.).

After sending the TALK command to a serial device to tell it to provide output, and optionally a secondary address with TKSA, a program calls ACPTR to read data from the device, one byte at a time. See TALK.

This behavior can be overridden or extended with the IACPTR vector.

Example:

```
acptr = $ffa5

    jsr $ffa5
    sta data
```

ADDKEY

Address: JSR $FF4A

Description:
Add a character to the keyboard input buffer

Inputs: **A** The PETSCII code

Outputs: **C flag** 0 on success, 1 on failure

Remarks: The MEGA65 KERNAL reads keyboard events from three sources: the active function key macro, a keyboard buffer managed by the KERNAL, and the hardware typing event queue. When GETIN reads from the keyboard, it checks each of these sources, in order.

ADDKEY adds a PETSCII code to the keyboard buffer. This buffer is processed by GETIN when no function key macro is active. It supersedes the hardware typing event queue.

The keyboard buffer has a fixed size. If the buffer is full, the key will not be added, and the C flag will be set.

NOTE: For the MEGA65, the buffer is consumed by calls to GETIN, and not by the KERNAL IRQ as on a C64. A program must return to the screen editor for GETIN to be called. If a program needs to enqueue more than a few characters, one option is to use an IIRQ vector to feed characters into the buffer gradually.

Example:

```
addkey = $ff4a

    ; Cause RUN and a carriage return to be typed
    ; once control returns to the screen editor.
    lda #'R'
    jsr addkey
    bcs err
    lda #'U'
    jsr addkey
    bcs err
    lda #'N'
    jsr addkey
    bcs err
    lda #13
    jsr addkey
    bcs err
```

BASIN

Address: JSR $FFCF

Description:
Read a character from input device

Outputs:
A The character read
C flag Set on error

Remarks: The default input device is the keyboard. A program can change the input device with OPEN and CHKIN.

BASIN treats the input device as buffered and line oriented, with each line ending in a carriage return (PETSCII 13).

If the device is RS-232 or serial and the buffer is empty, BASIN waits for the next character. If the device is in an error state, BASIN skips input and returns a carriage return.

If the device is the keyboard, the cursor is activated and BASIN waits for the user to type a logical line and press Return. Each call to BASIN returns a character from the line, up to and including the carriage return. This also advances the cursor position to the last character on the logical line—but not after it. (Writing a character at this point, such as with BSOUT, overwrites the last character on the line. It is usually appropriate to write a single carriage return to the screen terminal.)

If the device is the screen, BASIN behaves as if the user pressed Return with keyboard input, and reads from the current cursor position up to the end of the logical line. The final character of a logical line is a carriage return.

To attempt to read a character without waiting on an empty buffer, use GETIN.

This routine can be overridden or extended using the IBASIN vector.

Example:

```
basin = $ffcf

    ; Read a line into memory, up to
    ; 255 characters
    ldy #0
read_more:
    jsr basin
    sta data,y
    iny
    beq out_of_memory
    cmp #$0d   ; carriage return?
    bne read_more
```

BOOT_SYS

Address: JMP $FF59

Description:

Boot an alternate system from disk

Remarks: The Commodore 65 intended a feature where a floppy disk in the internal 3-1/2" disk drive could bootstrap software using up to 512 bytes of machine code in a "home" sector, at track 0, sector 1.

When the BOOT_SYS KERNAL routine is called, or the user performs the **BOOT SYS** command, or the user holds the Alt key during start-up, the KERNAL reads the home sector of the disk. If the first byte is $4C, the KERNAL loads the sector into memory at 0.0400, then invokes it with JMP. If no disk is present or the first byte is not $4C, no action is taken.

The boot sector feature is distinct from the BASIC 10 **BOOT** command that loads and runs a file named `AUTOBOOT.C65`. It is not compatible with C128-style boot sectors.

This routine does not return. Only use JMP with this routine.

NOTE: For the MEGA65, this feature is untested.

Example:

```
boot_sys = $ff59

        jmp boot_sys
```

BSOUT

Address: JSR $FFD2

Description:
Write a character to output device

Inputs: **A** Character to output

Outputs: **C flag** Set on error
A Error code, if any

Remarks: The default output device is the screen. A program can change the output device with OPEN and CKOUT.

BSOUT treats the output device as buffered.

If the device is RS-232 and the output buffer is full, BSOUT waits until there is room in the buffer.

If the device is serial, a single character buffer is used with CIOUT and the previously buffered character is sent.

If the device is the screen, the character is written to screen memory at the current cursor position.

This routine can be overridden or extended using the IBSOUT vector.

Example:

```
bsout = $ffd2

    lda #$0d
    jsr bsout
```

C64MODE

Address: JMP $FF53

Description:

Reset to GO64 mode

Remarks: This switches to the GO64 mode memory map, resets the VIC modes, and jumps to the GO64 start routine.

This routine does not return. Only use JMP with this routine.

Example:

```
c64mode = $ff53

        jmp c64mode
```

CHKIN

Address: JSR $FFC6

Description:
Set input channel

Inputs: **X** The logical address of the input device

Outputs: **C flag** Set on error
A Error code, if any

Remarks: This changes the input channel to use a given device identified by its logical address. Call CHKIN after a call to OPEN. BASIN and GETIN read from the current input channel.

The default input device is the keyboard. A program can call CLRCH to restore default channel settings.

CHKIN performs device-specific tasks, such as sending a TALK command to a serial channel. Setting a non-input device results in an error.

This routine can be overridden or extended using the ICHKIN vector.

Example:

```
chkin = $ffc6

    ldx #8
    jsr chkin
```

CINT

Address: JSR $FF81

Description:
Initialize screen editor

Remarks: CINT resets all properties of the screen editor, including indirect vectors, function key macros, VIC registers, and SID registers. It also clears the screen.

CINT is typically called along with IOINIT, which resets additional properties of the screen editor.

Because it updates indirect vectors, it must be called with IRQs disabled.

Example:

```
cint = $ff81

    sei
    jsr cint
    cli
```

CIOUT

Address: JSR $FFA8

Description:
Send a byte to listener

Inputs: **A** The byte to send

Remarks: This is a low-level serial routine. Most programs will prefer the higher level I/O routines (OPEN et al.).

After sending the LISTN command to a serial device to tell it to listen for input, and optionally a secondary address with SECND, a program uses CIOUT to send data to the device, one byte at a time.

This behavior can be overridden or extended with the ICIOUT vector.

Example:

```
ciout = $ffa8

        lda  data
        jsr  ciout
```

CKOUT

Address: JSR $FFC9

Description:
Set output channel

Inputs: **X** The logical address of the output device

Outputs: **C flag** Set on error
A Error code, if any

Remarks: This changes the output channel to use a given device identified by its logical address. Call CKOUT after a call to OPEN. BSOUT writes to the current output channel.

The default output device is the screen terminal. A program can call CLRCH to restore default channel settings.

CKOUT performs device-specific tasks, such as sending a LISTN command to a serial channel. Setting a non-output device results in an error.

This routine can be overridden or extended using the ICKOUT vector.

Example:

```
ckout = $ffc9

        ldx #8
        jsr chkout
```

CLALL

Address: JSR $FFE7

Description:
Close all files and channels

Remarks: All open logical files are closed, as if closed with CLOSE. This includes performing device-specific closing tasks, as well as restoring the default input and output channels (as if via CLRCH).

This behavior can be overridden or extended with the ICLALL vector.

Example:

```
clall = $ffe7

      jsr clall
```

CLOSE

Address: JSR $FFC3

Description:
Close logical file

Inputs: **A** The logical file to close
C flag Set to delete logical file without sending close command

Outputs: **C flag** Set on error
A Error code, if error

Remarks: This performs device-specific closing tasks for the given open logical file number.

If the file is already closed, or the logical file number refers to the keyboard or screen, CLOSE returns without an error.

If the device is RS-232, this waits for the write buffer to empty before closing.

If the device is serial, any buffered output character is sent, a "close" command is sent if appropriate, and the device is told to UNLSTN.

As a special case, if the C flag is set, the device is a disk (FA is 8 or greater), and the secondary address is the command channel (15), no "close" command is sent, and the logical file is removed from the file list. This is necessary to prevent the disk from closing other opened files when closing the command channel. See OPEN for a complete example.

NOTE: When CLOSE is used on a file that was set as the input or output channel by CHKIN or CKOUT, it does *not* reset the input/output channel, and a subsequent read/write may misbehave. Use CLRCH after CLOSE to reset the input/output channels.

This behavior can be overridden or extended with the ICLOSE vector.

Example:

```
close = $ffc3
clrch = $ffcc

    clc           ; normal close
    lda #$01
    jsr close
    jsr clrch     ; clear I/O channels after
                  ; closing the file
```

CLOSE_ALL

Address: JSR $FF50

Description:
Close all files on a device

Inputs: **A** The device number (0-31)

Remarks: This calls CLOSE for each open file for the given device (FA).

If one of the closed files is the current input or output channel, the default is restored.

Example:

```
close_all = $ff50

    lda #$08
    jsr close_all
```

CLRCH

Address: JSR $FFCC

Description:
Restore default channels

Remarks: This clears all open channels and restores the default system I/O channels, including the keyboard as the input channel and the screen terminal as the output channel.

If the current input or output channels are to serial devices, CLRCH sends UNTLK or UNLSN, as appropriate.

This routine can be overridden or extended using the ICLRCH vector.

Example:

```
clrch = $ffcc

        jsr clrch
```

CMP_FAR

Address: JSR $FF7A

Description:

Compare a byte with an address in any bank

Inputs:
A Data to compare
X Base page pointer
Y Index from pointed address
Z Bank

Outputs: CPU status flags based on the comparison

Remarks: Conceptually, this is the equivalent of `cmp (X),y` in bank Z.

This can access any address in the first 1MB of memory. Unlike JMPFAR, this does not change the memory map. It uses a DMA job to access the long address.

For the MEGA65, the 45GS02 supports an instruction that can do this with a 32-bit address stored on the base page, without the need for a KERNAL routine: `cmp [zp4],z`

Example:

```
cmp_far = $ff7a

    ; Compare memory at address 4.4502
    ; to the accumulator
    lda #$00
    sta $fe
    lda #$45
    sta $ff
    ldx #$fe   ; base page pointer -> $4500
    ldy #$02   ; Y index
    ldz #$04   ; bank
    lda #$99   ; data to compare
    jsr cmp_far
    beq they_are_equal  ; ...
```

GETIN

Address: JSR $FFE4

Description:

Read a character from input device, without waiting

Outputs: **A** The character read
C flag Set on error

Remarks: The default input device is the keyboard. A program can change the input device with OPEN and CHKIN.

GETIN treats the input device as buffered, but does not wait for input. If the input buffer for the device is empty, GETIN returns $00.

For the MEGA65, if the input device is the keyboard, GETIN uses several keyboard input buffers: the KERNAL keyboard buffer, the function key macro buffer, and the MEGA65 hardware keyboard buffer. Unlike the C65, the MEGA65 KERNAL does not populate the KERNAL keyboard buffer during the IRQ handler. Instead, GETIN consumes one typing event from the MEGA65 hardware keyboard buffer directly. The KERNAL keyboard buffer is still honored to support software features that inject keystrokes.

If the input device is serial or the screen, it behaves like BASIN.

This routine can be overridden or extended using the IGETIN vector.

Example:

```
getin = $ffe4

get_a_key:
    jsr getin
    beq get_a_key
    sta data
```

GETIO

Address: JSR $FF41

Description:

Read the current input and output devices

Outputs: **X** current input device (FA)
Y current output device (FA)

Remarks: An input device of 0 represents the keyboard.

An output device of 3 represents the screen.

These settings are modified by CHKIN and CKOUT.

Example:

```
getio = $ff41

    jsr getio
    cpy #3
    beq is_screen_output
```

GETLFS

Address: JSR $FF44

Description:

Read file, device, secondary address

Outputs:
A The logical file number (LA)
X The device number (FA)
Y The secondary address, or $FF (SA)

Remarks: These settings are modified by SETLFS, as well as other KERNAL calls and BASIC commands that look up information about open files via the logical file number.

A program can use this to determine from which disk drive the program was loaded by calling it just after starting, before performing other disk I/O.

Example:

```
getlfs = $ff44

    jsr getlfs
```

IOBASE

Address: JSR $FFF3

Description:

RESERVED, DO NOT USE

Remarks: Historically, this routine returns the start address of the I/O registers. On the C65, this is always $D000.

IOINIT

Address: JSR $FF84

Description:

Initialize I/O devices

Remarks: IOINIT resets the CIA chips, the 45GS02 port, the VIC chip, the UART, and the DOS.

It must be called with IRQs disabled.

Example:

```
ioinit = $ff84

    sei
    jsr ioinit
    cli
```

JMPFAR

Address: JMP $FF71

Description:

Jump to address in any bank

Inputs:

$02 Address bank (0-5)
$03 Address high
$04 Address low
$05 CPU status register
$06 CPU A register
$07 CPU X register
$08 CPU Y register
$09 CPU Z register

Remarks: This updates the CPU memory map to make a requested address visible as a 16-bit address, then JMPs to it. It does not return.

This routine has similar restrictions as the **SYS** BASIC command, and cannot reach every address. It can only jump to addresses in non-zero banks from X.2000 – X.7FFF.

The revised memory map keeps $0000-$1FFF and $8000-$DFFF unmapped (so they refer to bank 0 and I/O registers), and $E000-$FFFF mapped to KERNAL ROM.

A program sets parameters to JMPFAR by writing to the KERNAL's base page. Parameters include the three address bytes, and the state of the CPU registers expected on entry to the new location.

This routine does not return. Only use JMP to call this routine.

Example:

```
jmpfar = $ff71

    ; MAP $2000-$7FFF to bank 4, then JMP
    ; to 4.4500
    lda #$04
    sta $02
    lda #$45
    sta $03
    lda #$00
    sta $04
    lda #$04   ; Start routine with
               ; interrupts disabled
    sta $05
    jmp jmpfar
```

JSRFAR

Address: JSR $FF6E

Description:
Call subroutine in any bank

Inputs:
$02 Address bank (0-5)
$03 Address high
$04 Address low
$05 CPU status register
$06 CPU A register
$07 CPU X register
$08 CPU Y register
$09 CPU Z register

Outputs: CPU status, A, X, Y, and Z registers are as they were at the end of the subroutine.

Remarks: This updates the CPU memory map to make a requested address visible as a 16-bit address, then JSRs to it. On return, it restores the previous KERNAL-managed memory map (e.g. the SYS map).

This routine has similar restrictions as the **SYS** BASIC command, and cannot reach every address. It can only jump to addresses in non-zero banks from X.2000 – X.7FFF.

The revised memory map keeps $0000-$1FFF and $8000-$DFFF unmapped (so they refer to bank 0 and I/O registers), and $E000-$FFFF mapped to KERNAL ROM.

A program sets parameters to JSRFAR by writing to the KERNAL's base page. Parameters include the three address bytes, and the state of the CPU registers expected on entry to the new location.

Only use JSR to call this routine.

Example:

```
jsrfar = $ff6e

        ; MAP $2000-$7FFF to bank 4, then
        ; JSR to 4.4500
        lda #$04
        sta $02
        lda #$45
        sta $03
        lda #$00
        sta $04
        lda #$04   ; Start routine with
                   ; interrupts disabled
        sta $05
        jsr jsrfar
```

```
; Program continues...
```

KEY

Address: JSR $FF9F

Description:

Perform a keyboard scan.

Remarks: For the MEGA65, typing events are managed by a hardware buffer mechanism and not the KERNAL. A program that wants to consume a typing event can call GETIN, or access the typing event buffer registers directly, without calling KEY.

KEY manages special keyboard behaviors, including Mega + Shift, No Scroll, and Ctrl-S. The MEGA65 KERNAL calls KEY in the KERNAL IRQ handler many times per second, similar to other Commodores. A program that disables the KERNAL IRQ handler but wants to maintain these behaviors can call KEY from a custom handler.

NOTE: For the MEGA65, KEY diverges from the C65 and does not offer vector hooks. Keyboard intercept vectors are a work in progress.

Example:

```
key = $ff9f

    jsr key
```

KEYLOCKS

Address: JSR $FF47

Description:
Read/set keyboard locks

Inputs: To set:
C flag = 0
A The new locks value

To read:
C flag = 1

Outputs: When reading: **A** The locks value

Remarks: Locks bits: **Bit 5:** Function keys (0=macros, 1=raw keys)
Bit 6: No Scroll locked (0=unlocked, 1=locked)
Bit 7: Mega+Shift locked (0=unlocked, 1=locked)

Bits 0–4 are reserved for future use. To maintain future compatibility, always read the locks value and preserve the values of these bits.

Example:

```
keylocks = $ff47

    sec
    jsr keylocks
    ora #%00100000   ; Disable function key macros
    clc
    jsr keylocks
```

LDA_FAR

Address: JSR $FF74

Description:

Read a byte from an address in any bank

Inputs:
X Base page pointer
Y Index from pointed address
Z Bank

Outputs: **A** The value at the address

Remarks: Conceptually, this is the equivalent of `lda (X),y` in bank Z.

This can access any address in the first 1MB of memory. Unlike JMPFAR, this does not change the memory map. It uses a DMA job to access the long address.

For the MEGA65, the 45GS02 supports an instruction that can do this with a 32-bit address stored on the base page, without the need for a KERNAL routine: `lda [zp4],z`

Example:

```
lda_far = $ff74

    ; Load memory at address 4.4502
    ; into the accumulator
    lda #$00
    sta $fe
    lda #$45
    sta $ff
    ldx #$fe   ; base page pointer -> $4500
    ldy #$02   ; Y index
    ldz #$04   ; bank
    jsr lda_far
```

LISTN

Address: JSR $FFB1

Description:
Send “listen” command

Inputs: **A** The device number (FA, 0-31)

Remarks: This is a low-level serial routine. Most programs will prefer the higher level I/O routines (OPEN et al.).

With the low-level serial protocol, the computer is expected to coordinate all device communication on the serial bus. Each device has a hard-coded device number; some devices, like disk drives, may have switches for changing its recognized device number. All devices see all traffic on the bus, and are expected to change their mode when they see a command with their device number.

When the computer wants to send data to a device, the computer sends the device the “listen” command. Typically, the computer is only sending data to one device at a time. To change listeners, the computer broadcasts an “untalk” command to all devices, then sends another “listen” command with a different device number.

The computer gives one device at a time permission to “talk” on the serial bus. To change talkers, the computer broadcasts an “untalk” command to all devices, then sends another “talk” command.

Use UNLSN to tell all devices to stop talking.

This behavior can be overridden or extended with the ILISTEN vector.

Example:

```
listn = $ffb1

    lda #8
    jsr listn
```

LKUPLA

Address: JSR $FF5F

Description:

Search for logical file number in use

Inputs: **A** The logical file number (LA)

Outputs: If found:
C flag = 0
A The logical file number (LA)
X The device number (FA)
Y The secondary address (SA), or $FF if not applicable

If not found:
C flag = 1

Remarks: This is used primarily by BASIC to locate an unused logical file number. In practice, most programs have complete control over the logical file numbers they use, and no look-up is necessary.

This can also be used to identify the device for an open file by the logical file number.

Example:

```
lkupla = $ff5f

    lda #$01
    jsr lkupla
    bcs not_in_use
```

LKUPSA

Address: JSR $FF62

Description:
Search for secondary address in use

Inputs: **Y** The secondary address (SA)

Outputs: If found:
C flag = 0
A The logical file number (LA)
X The device number (FA)
Y The secondary address (SA), or $FF if not applicable

If not found:
C flag = 1

Remarks: This is used primarily by BASIC to locate an unused secondary address. In practice, most programs have complete control over the secondary addresses they use, and no look-up is necessary.

Example:

```
lkupsa = $ff62

    ldy #$60
    jsr lkupsa
    bcs not_in_use
```

LOAD

Address: JSR $FFD5

Description:
Load/verify from file

Inputs:
A = 0 to load, 1 to verify; set bit 6 for "raw" mode
X Load address, low
Y Load address, high

Outputs:
C flag Set on error
A Error code, if error
X Ending address, low
Y Ending address, high

Remarks: Before calling LOAD, the program must call SETBNK, SETLFS, and SETNAM to set parameters for the file to load. LOAD cannot be used with RS-232 devices, the screen, or the keyboard. The logical file number is not used.

When used with a disk drive, LOAD only loads a file of type PRG. To load a file of another type, use SETNAM with the type mentioned in the name string (such as `"MYFILE,S"` for a SEQ file), along with OPEN, CHKIN, BASIN, and CLOSE. See OPEN for an example.

If A bit 0 is clear, LOAD loads the file from the input device and writes it to memory.

If A bit 0 is set, LOAD verifies the file: it loads the file and compares it to the contents of memory. If the file does not match memory, LOAD returns an error status. The memory is not changed.

If A bit 6 is set ("raw" mode), LOAD treats the first two bytes of the file as data. When clear, the first two bytes of the file are expected to be the stored load address, as is typical for a PRG file. A PRG loaded with raw mode must still contain at least two bytes, or a File Not Found error is reported.

If the secondary address (SA) is $00, the provided X/Y is used as the load address, and the first two bytes of the file are skipped if not in "raw" mode. If SA is not $00, X/Y are ignored, and the actual load address is taken from the stored load address in the first two bytes of the file. This is the same behavior as this BASIC command: `LOAD "FILE",8,1`

The complete LOAD operation must fit within a single bank. It cannot cross bank boundaries.

LOAD performs all of the tasks needed to open the file, read it, then close it. It does not leave any logical files open (as OPEN does).

On success, the address of the byte last byte loaded (or verified) *plus one* is returned in the X and Y registers.

SAVE sets the I/O status as appropriate. This can be read with READSS.

This behavior can be overridden or extended with the ILOAD vector.

Example:

```
load = $ffd5
setbnk = $ff6b
setlfs = $ffba
setnam = $ffbd

    lda #$00    ; memory in bank 0
    ldx #$00    ; filename in bank 0
    jsr setbnk

    lda #$00    ; (not used by load)
    ldx #$08    ; device number
    ldy #$00    ; load
    jsr setlfs

    lda #filename_end-filename
    ldx #<filename
    ldy #>filename
    jsr setnam

    lda #0      ; load
    ldx #$00    ; to 0.4500
    ldy #$45
    jsr load

    rts

filename:
    !pet "myfile"
filename_end:
```

MEMBOT

Address: JSR $FF9C

Description:
RESERVED, DO NOT USE

Remarks: Historically, MEMBOT and MEMTOP refer to memory allocation parameters used by the KERNAL. The C65 KERNAL does not use this facility. These routines are mentioned in C65 documentation but do not do anything.

MEMTOP

Address: JSR $FF99

Description:
RESERVED, DO NOT USE

Remarks: Historically, MEMBOT and MEMTOP refer to memory allocation parameters used by the KERNAL. The C65 KERNAL does not use this facility. These routines are mentioned in C65 documentation but do not do anything.

MONEXIT

Address: JMP $FFA2

Description:
Monitor's exit to BASIC

Remarks: This is only used by the monitor, and is not useful to programs.

Example:
```
monexit = $ffa2

     jmp monexit
```

MonitorCall

Address: JMP $FF56

Description:

Invoke monitor

Remarks: This stops the program, then invokes the monitor.

This routine does not return. Only use JMP to call this routine.

When the monitor exits, it returns to the `READY.` prompt.

Example:

```
monitor_call = $ff56

        jmp monitor_call
```

OPEN

Address: JSR $FFC0

Description:
Open logical file

Outputs: **C flag** Set on error
A Error code, if error

Remarks: Before calling OPEN, the program must call SETBNK, SETLFS, and SETNAM to set parameters for the file to open.

The requested logical file number (LA) must be currently un-opened. Use LKUPLA to test for unused logical file numbers, if necessary. There can be up to ten logical files opened simultaneously.

OPEN performs device-specific opening tasks for serial, RS-232, and keyboard and screen devices, including clearing the previous status and transmitting any given filename or command string supplied to SETNAM and SETBNK.

OPEN sets the I/O status as appropriate. This can be read with READSS.

An open file can be made the current input device with CHKIN, or the current output device with CKOUT. BASIN and GETIN read from the input device, and BSOUT writes to the output device.

A program must close an open file when it is done using it. To close a specific file, use CLOSE. To close all files for a device, use CLOSE_ALL. To close all open files, use CLALL. When closing a disk command channel, set the Carry flag before calling CLOSE; see CLOSE for more information.

This behavior can be overridden or extended with the IOPEN vector.

Examples: Disk drives accept arguments in the filename string set with SETNAM, following the filename and delimited with commas. The following example opens a file of type SEQ for writing (`"...,s,w"`), then writes the bytes 0 to 255. (Error checking is elided for brevity.)

```
open = $ffc0
close = $ffc3
ckout = $ffc9
bsout = $ffd2
setbnk = $ff6b
setlfs = $ffba
setnam = $ffbd

    lda #$00    ; memory in bank 0
    ldx #$00    ; filename in bank 0
    jsr setbnk
    lda #$01     ; logical file number
    ldx #$08     ; device number
    ldy #$02     ; secondary address
                 ; (not 0, 1, or 15)
    jsr setlfs

    lda #filename_end-filename
    ldx #<filename
    ldy #>filename
    jsr setnam

    jsr open
    ldx #$01
    jsr ckout

    lda #0
-   jsr bsout
    inc
    bne -

    clc
    lda #$01
    jsr close

    rts

filename:
    !pet "myseqfile,s,w"
filename_end:
```

Disk drives have a *command channel* that can be accessed with the secondary address 15. When a logical file is opened to the drive's SA of 15, the filename is the text of the command to perform. The command is performed immediately. The following example is similar to the BASIC command `OPEN 1,8,15,"I0"`, which sends the initialization command to disk drive device 8.

```
open = $ffc0
close = $ffc3
setbnk = $ff6b
setlfs = $ffba
setnam = $ffbd

    lda #cmd_initialize_end-cmd_initialize
    ldx #<cmd_initialize
    ldy #>cmd_initialize
    jsr setnam
    ldx #0
    jsr setbnk

    lda #1
    ldx #8
    ldy #15
    jsr setlfs

    jsr open     ; perform I0 command
    bcs error

    lda #1
    sec          ; "special close" (see CLOSE)
    jsr close

    ; ...

cmd_initialize:
    !pet "i0"
cmd_initialize_end:
```

PFKEY

Address: JSR $FF68

Description:

Program an editor function key

Inputs:

A Base page pointer to string address (lo/hi/bank)
X Function key number (1-16)
Y String length

Outputs: **C flag** Set if the update failed

Remarks: The screen editor supports user-defined macros associated with function keys. The PFKEY call updates a macro's definition, similar to the **KEY** BASIC command.

As input, the program sets the accumulator to a base page address that contains the 24-bit address of the macro string, in little-endian format. The Y register is the string's length, up to 240 characters.

The X register is the key to define: 1 – 14 for the corresponding numbered function key, 15 for the Help key, and 16 for the Stop (Shift + Run/Stop) key.

The string contains the PETSCII codes that the macro will type when executed. This is unlike the BASIC string expression that you would use with the **KEY** command: special characters like double-quotes must appear as PETSCII codes in the string used with PFKEY. (See the example below.)

If successful, PFKEY copies the string into macro memory, and does not need the string at the provided address to persist.

Example:

```
pfkey = $ff68

    ; Store string address in base page $fd-$ff
    lda #<macro
    sta $fd
    lda #>macro
    sta $fe
    lda #0
    sta $ff

    ; Install macro string for F6 key
    lda #$fd
    ldy #macro_end-macro
    ldx #6
    jsr pfkey

    rts
```

```
macro:
    !pet "print ",34,"it works!",34,13
macro_end:
```

PHOENIX

Address: JMP $FF5C

Description:
Call disk boot loader

Remarks: See BOOT_SYS for a description of boot disks.

Some early Commodore 65 prototype ROMs replaced this routine with one that performed diagnostics and printed results. Later versions replaced this with a call to the boot sequence.

This routine does not return. Only use JMP with this routine.

NOTE: For the MEGA65, this feature is untested.

Example:

```
phoenix = $ff5c

        jmp phoenix
```

PLOT

Address: JSR $FFF0

Description:
Read/set cursor position

Inputs:
To set:
C flag = 0
X Cursor line
Y Cursor column

To read:
C flag = 1

Outputs:
When setting:
C flag Set on error

When reading:
X Cursor line
Y Cursor column

Remarks: The cursor position is relative to the active window. Use SCRORG to read the current window size.

When setting, if the requested position is outside the active window, the routine returns with the Carry flag set.

Example:

```
plot = $fff0
primm = $ff7d

    ; Move the cursor to the 78th column
    ; of the 3rd row
    clc
    ldy #77
    ldx #2
    jsr plot

    jsr primm
    !pet "message",0

    ; Read the new cursor position
    sec
    jsr plot
    stx $1600
    sty $1601
```

PRIMM

Address: JSR $FF7D

Description:

Print an inline null-terminated short string.

Remarks: The string to print immediately follows the JSR instruction in memory.

The string must end in a null ($00) byte, and must be at most 255 bytes in length including the null byte.

Characters are written to the output stream, as with BSOUT.

Only use JSR to call PRIMM.

PRIMM preserves registers. This makes it suitable for dropping in the middle of a program temporarily for troubleshooting purposes.

Example:

```
primm = $ff7d

    jsr primm
    !pet "this is a petscii string to print.",0

    ; Program continues...
```

RAMTAS

Address: JSR $FF87

Description:

Initialize RAM and buffers

Remarks: RAMTAS resets KERNAL base page variables, and resets pointers to the top and bottom of system RAM.

This is primarily for use by the system start-up and reset sequences. A machine code program executed from a BASIC bootstrap program cannot cleanly return to BASIC after calling RAMTAS, nor can RAMTAS be called from the monitor.

Example:

```
ramtas = $ff87

      jsr ramtas
```

RDTIM

Address: JSR $FFDE

Description:
Read CIA1 24-hour clock

Outputs:
Y Hours, 0-23, in BCD
X Minutes, 0-59, in BCD
A Seconds, 0-59, in BCD
Z Tenths of a second, 0-9

Remarks: This reads the time-of-day (TOD) clock feature of the CIA1 device. This clock counts tenths of seconds, and maintains hours, minutes, seconds, and tenths of seconds.

Use SETTIM to set the value of the TOD clock.

For the MEGA65, the CIA1 TOD clock has no relationship to the battery-backed Real-Time Clock (RTC). The value returned by RDTIM is the CIA1 TOD value, not the RTC value. BASIC 65's TI$ and DT$ special variables use the RTC, and not the TOD.

Values are in Binary Coded Decimal format, where each nibble represents a decimal digit. For example, when RDTIM returns the hexadecimal value $59 in the accumulator, this represents a value of 59 (decimal) seconds.

Example:

```
rdtim = $ffde
bsout = $ffd2

    ; Print hours and minutes, as HH:MM
    jsr rdtim
    tya
    jsr print_bcd
    lda #':'
    jsr bsout
    txa
    jsr print_bcd
    rts

print_bcd:
    pha
    lsr
    lsr
    lsr
    lsr
    adc #'0'
    jsr bsout
    pla
```

```
and #$0f
adc #'0'
jsr bsout
rts
```

READSS

Address: JSR $FFB7

Description:
Get status of last I/O operation

Outputs: **A** The status byte

Remarks: This returns the status of the last I/O operation performed for a serial or RS-232 device. This only works with high-level I/O operations such as OPEN, LOAD, and SAVE.

In the case of an RS-232 device, READSS clears the status byte and returns its previous value.

Example:

```
readss = $ffb7

        jsr readss
```

RESTOR

Address: JSR $FF8A

Description:
Initialize KERNAL vector table

Remarks: This resets all KERNAL-related vectors to their default values, pointing to routines in ROM.

It does not reset vectors used by the screen editor and BASIC.

It must be called with interrupts disabled.

See VECTOR.

Example:

```
restor = $ff8a

    sei
    jsr restor
    cli
```

SAVE

Address: JSR $FFD8

Description:
Save to file

Inputs:
A Base page pointer to start address
X End address + 1, low
Y End address + 1, high

Outputs:
C flag Set on error
A Error code, if error

Remarks: Before calling SAVE, the program must call SETBNK, SETLFS, and SETNAM to set parameters for the file to load. SAVE cannot be used with RS-232 devices, the screen, or the keyboard. The logical file number is not used.

When used with a disk drive, SAVE saves a file of type PRG by default. To save a file of another type, use SETNAM with the type mentioned in the name string (such as `"MYFILE,S"` for a SEQ file), along with OPEN, CKOUT, BSOUT, and CLOSE. See OPEN for an example.

By default, saving a file with the name of an existing file is an error, and does not change the file. To overwrite the file if it exists, prefix the filename string with: `@:` For example: `"@:MYFILE"`

SAVE writes a region of memory to a file on a serial device. The start address of the memory region must be stored as a 16-bit address, little-endian, on the base page. The program provides the base page address of this pointer in the A register.

The program provides the ending address of the memory region as a 16-bit value *plus one* in the X and Y registers.

The complete memory region must fit within a single bank. It cannot cross bank boundaries.

SAVE performs all of the tasks needed to open the file, write it, then close it. It does not leave any logical files open (as OPEN does).

SAVE sets the I/O status as appropriate. This can be read with READSS.

This behavior can be overridden or extended with the ISAVE vector.

Example:

```
save = $ffd8
setbnk = $ff6b
setlfs = $ffba
setnam = $ffbd

    ; fill memory with data to test saving
    ldx #0
```

```
-   txa
    sta $4500,x
    inx
    bne -

    lda #$00    ; memory in bank 0
    ldx #$00    ; filename in bank 0
    jsr setbnk

    lda #$00    ; (not used by save)
    ldx #$08    ; device number
    ldy #$01    ; save
    jsr setlfs

    lda #filename_end-filename
    ldx #<filename
    ldy #>filename
    jsr setnam

    ; Save 0.4500 to 0.45FF
    lda #$00
    sta $fe
    lda #$45
    sta $ff
    lda #$fe
    ldx #$00
    ldy #$46

    jsr save
    bcs error

    ; ...

filename:
    !pet "myfile"
filename_end:
```

ScanStopKey

Address: JSR $FFEA

Description:

Scan Stop key

Outputs: **A** $7F if the Stop key is pressed, $FF otherwise
N flag Clear if the Stop key is pressed, set otherwise

Remarks: Unlike STOP, ScanStopKey does not invoke the ISTOP vector, and does not close channels or flush the keyboard buffer.

Example:

```
scan_stop_key = $ffea

        ; Wait for Stop key
-       jsr scan_stop_key
        bmi -
```

SCRORG

Address: JSR $FFED

Description:
Get current screen window size

Outputs: **C flag** Text screen width: 0=80, 1=40
X Window width
Y Window height
A Window top-left screen memory address, low
Z Window top-left screen memory address, high

Remarks: The screen editor maintains a "window," a rectangle on the text screen, where terminal output is written and input is accepted. The window size and position defaults to the full screen, and can be adjusted with the **WINDOW** BASIC command.

SCRORG returns properties of this window that can be used to plot characters in screen memory.

Example:

```
scrorg = $ffed

    ; Plot a spade glyph in the top left corner
    ; of the active window.
    jsr scrorg
    sta $fe
    stz $ff
    lda #65
    ldy #0
    sta ($fe),y
```

SECND

Address: JSR $FF93

Description:
Send secondary address to listener

Inputs: **A** Secondary address (SA)

Remarks: This is a low-level serial routine. Most programs will prefer the higher level I/O routines (OPEN et al.).

After sending the LISTN command to a serial device to tell it to accept input, SECND sends a secondary address to the device. This is typically used to configure the device prior to sending it data. The behavior depends on the device.

To send a secondary address to a TALKing device, use TKSA.

This behavior can be overridden or extended with the ISECOND vector.

Example:

```
listn = $ffb1
secnd = $ff93

    lda #8
    jsr listn
    lda #15
    jsr secnd
```

SETBNK

Address: JSR $FF6B

Description:
Set megabyte and bank for I/O and filename memory address

Inputs: For banks 0 – 5:
A The memory bank
X The filename bank

For 28-bit addresses:
A Bit 7 set, bits 0 – 3 = bits 24 – 27 of memory address
Y Bits 16 – 23 of memory address
X Bit 7 set, bits 0 – 3 = bits 24 – 27 of filename address
Z Bits 16 – 23 of filename address

Remarks: SETBNK must be used for any memory I/O operation, along with SETLFS and SETNAM for opening files. The lower two bytes of each address are arguments to other calls.

For backwards compatibility with the C65 KERNAL, SETBNK has two separate modes, one for banks 0 – 5 in the first megabyte of memory, and another for 28-bit addresses.

If the first register's bit 7 is clear, it represents an address in the first megabyte, and the value is the bank number. For example, if A = $04, then the memory address is in bank 4 of the first megabyte ($004.xxxx).

If the first register's bit 7 is set, it represents a 28-bit address, and the lower nibble of the first register is the top-most nibble of the address. The second register represents the next two nibbles of the address. For example, if A = $88 and Y = $03, then the memory address is a 28-bit address of the form $803.xxxx.

The memory address is set with input registers A and Y. The filename address is set with input registers X and Z. The memory bank and filename bank can be different, and can use bank mode or 28-bit mode independently.

See OPEN for a complete example.

Example:

```
setbnk = $ff6b

    ; Memory address:     $4.xxxx
    ; Filename address:   $0.xxxx
    lda #$04
    ldx #$00
    jsr setbnk

    ; Memory address:   $803.xxxx
    ; Filename address:   $0.xxxx
    lda #$88
    ldy #$03
    ldx #$00
    jsr setbnk

    ; Memory address:   $803.xxxx
    ; Filename address: $805.xxxx
    lda #$88
    ldy #$03
    ldx #$88
    ldz #$05
    jsr setbnk
```

SETLFS

Address: JSR $FFBA

Description:

Set file, device, secondary address

Inputs: **A** The logical file number (LA)
X The device number (FA)
Y The secondary address, or $FF (SA)

Remarks: SETLFS must be used for high-level file operations (OPEN, LOAD, SAVE), along with SETBNK and SETNAM.

The logical file number must be unique among opened files. If used with OPEN, it must not yet be open.

If the secondary address does not apply, set Y to $FF. For disk drives, secondary addresses 0 and 1 are reserved for LOAD and SAVE, and 15 is the command channel.

See OPEN for a complete example.

Example:

```
setlfs = $ffba

    lda #$01
    ldx #$08
    ldy #$02
    jsr setlfs
```

SETMSG

Address: JSR $FF90

Description:
Enable/disable KERNAL messages

Inputs: **A** bit 7 = control messages, bit 6 = error messages

Remarks: The KERNAL can print messages to the screen when performing disk operations, such as “LOADING,” “SAVING,” “VERIFYING,” and “I/O ERROR.” These are disabled by default.

To enable KERNAL control messages, set bit 7. To enable KERNAL I/O error messages, set bit 6.

KERNAL messages are not the same as those output by BASIC.

Example:

```
setmsg = $ff90

    ; Enable all KERNAL messages
    lda #$c0
    jsr setmsg
```

SETNAM

Address: JSR $FFBD

Description:

Set filename pointers

Inputs: **A** The string length, or $00
X The string address, low
Y The string address, high

Remarks: SETNAM must be used for high-level file operations (OPEN, LOAD, SAVE), along with SETBNK and SETLFS.

If a filename is not appropriate for the I/O device, SETNAM must be called with A = $00 (and any values for X and Y).

The bank for the address is set by SETBNK.

See OPEN for a complete example.

Example:

```
setnam = $ffbd

    lda #filename_end-filename
    ldx #<filename
    ldy #>fileanme
    jsr setnam

    ; ...

filename:
    !pet "myfile"
filename_end:
```

SETTIM

Address: JSR $FFDB

Description:

Set CIA1 24-hour clock

Inputs:

Y Hours, 0-23, in BCD
X Minutes, 0-59, in BCD
A Seconds, 0-59, in BCD
Z Tenths of a second, 0-9

Remarks: This sets the time-of-day (TOD) clock feature of the CIA1 device. This clock counts tenths of seconds, and maintains hours, minutes, seconds, and tenths of seconds.

Use RDTIM to read the current value of the TOD clock.

For the MEGA65, the CIA1 TOD clock has no relationship to the battery-backed Real-Time Clock (RTC). Setting the TOD clock does not affect the RTC. BASIC 65's TI$ and DT$ special variables use the RTC, and not the TOD.

Values are in Binary Coded Decimal format, where each nibble represents a decimal digit. For example, to set the seconds value to 59 seconds, use the hexadecimal value $59.

Example:

```
settim = $ffdb

    ; Set the TOD clock to 14:30:57.5
    ; (aka 2:30:57.5 PM)
    ldy #$14
    ldx #$30
    lda #$57
    ldz #$05
    jsr settim
```

SPIN_SPOUT

Address: JSR $FF4D

Description:

Set up fast serial ports

Inputs: **C flag** Clear for SPINP, set for SPOUT

Remarks: This routine relates to low-level driving of the C1571/1581 “fast serial” protocol.

From the C65 specification:

The fast serial protocol utilizes CIA_1 (6526 at $DC00) and a special driver circuit controlled in part by the FSDIR register. SPINP and SPOUT are routines used by the system to set up the CIA and fast serial driver circuit for input or output. SPINP sets up CRA (CIA_1 register 14) and clears the FSDIR bit for input. SPOUT sets up CRA, ICR (CIA_1 register 13), timer A (CIA_l registers 4 & 5), and sets the FSDIR bit for output. Note the state of the TOD_IN bit of CRA is always preserved. These routines are required only by applications driving the fast serial bus themselves from the lowest level.

Example:

```
spin_spout = $ff4d

    clc
    jsr $ff4d
```

STA_FAR

Address: JSR $FF77

Description:

Store a byte to an address in any bank

Inputs:

A Data to store
X Base page pointer
Y Index from pointed address
Z Bank

Remarks: Conceptually, this is the equivalent of `sta (X),y` in bank Z.

This can access any address in the first 1MB of memory. Unlike JMPFAR, this does not change the memory map. It uses a DMA job to access the long address.

For the MEGA65, the 45GS02 supports an instruction that can do this with a 32-bit address stored on the base page, without the need for a KERNAL routine: `sta [zp4],z`

Example:

```
sta_far = $ff77

    ; Store value to address 4.4502
    lda #$00
    sta $fe
    lda #$45
    sta $ff
    ldx #$fe   ; base page pointer -> $4500
    ldy #$02   ; Y index
    ldz #$04   ; bank
    lda #$99   ; the value to store
    jsr sta_far
```

STOP

Address: JSR $FFE1

Description:

Report Stop key, and reset I/O if pressed

Outputs: **Z flag** Set if Stop is pressed

Remarks: If the Stop key (Shift + Run/Stop) was pressed in the most recent call to ScanStopKey, STOP sets the Zero (Z) flag, closes all active channels, and flushes the keyboard queue. ScanStopKey is called routinely by the KERNAL, or can be called explicitly by the program if the KERNAL is disabled.

This behavior can be overridden or extended with the ISTOP vector.

To test for the Stop key without side effects, see ScanStopKey.

Example:

```
stop = $ffe1

    ; Wait for the Stop key
-   jsr stop
    bne -
```

SWAPPER

Address: JSR $FF65

Description:

Toggle between 40x25 and 80x25 text modes

Remarks: This does not yet support 80x50 text mode.

Example:

```
swapper = \$ff65

    jsr swapper
```

TALK

Address: JSR $FFB4

Description:
Send “talk” command to a device

Inputs: **A** The device number (FA, 0-31)

Remarks: This is a low-level serial routine. Most programs will prefer the higher level I/O routines (OPEN et al.).

With the low-level serial protocol, the computer is expected to coordinate all device communication on the serial bus. Each device has a hard-coded device number; some devices, like disk drives, may have switches for changing its recognized device number. All devices see all traffic on the bus, and are expected to change their mode when they see a command with their device number.

When the computer wants to read data from a device, the computer sends the device the “talk” command. Only one device at a time has permission to “talk” on the serial bus. To change talkers, the computer broadcasts an “untalk” command to all devices, then sends another “talk” command with a different device number.

Use UNTLK to tell all devices to stop talking.

This behavior can be overridden or extended with the ITALK vector.

Example:

```
talk = $ffb4

    lda #8
    jsr talk
```

TKSA

Address: JSR $FF96

Description:
Send secondary address to talker

Inputs: **A** Secondary address (SA)

Remarks: This is a low-level serial routine. Most programs will prefer the higher level I/O routines (OPEN et al.).

After sending the TALK command to a serial device to tell it to provide output, TKSA sends a secondary address to the device. This is typically used to configure the device prior to reading data. The behavior depends on the device.

To send a secondary address to a LISTNing device, use SECND.

This behavior can be overridden or extended with the ITALKSA vector.

Example:

```
talk = $ffb4
tksa = $ff96

    lda #8
    jsr talk
    lda #15
    jsr tksa
```

UNLSN

Address: JSR $FFAE

Description:
Send “unlisten” command

Remarks: This is a low-level serial routine. Most programs will prefer the higher level I/O routines (OPEN et al.).

UNLSN sends the “unlisten” command to all serial devices. See LISTN.

This behavior can be overridden or extended with the IUNLISTEN vector.

Example:

```
unlsn = $ffae

        jsr unlsn
```

UNTLK

Address: JSR $FFAB

Description:

Send “untalk” command

Remarks: This is a low-level serial routine. Most programs will prefer the higher level I/O routines (OPEN et al.).

UNTLK sends the “untalk” command to all serial devices. See TALK.

This behavior can be overridden or extended with the IUNTALK vector.

Example:

```
untlk = $ffab

        jsr untlk
```

VECTOR

Address: JSR $FF8D

Description:

Read/set KERNAL vector table

Inputs: To read:
C flag = 1
X Destination address, low
Y Destination address, high

To set:
C flag = 0
X Source address, low
Y Source address, high

Remarks: A program can override or extend some behaviors of the KERNAL, screen editor, and BASIC by installing custom routines. The KERNAL has a fixed set of extension points, known as *vectors*, where KERNAL behaviors jump to vector addresses under certain circumstances. The default values of these vectors point to routines in KERNAL ROM. A program installs a routine by replacing the original vector address with a custom routine's address, and typically (but optionally) rewriting the custom routine to call the original KERNAL vector address.

To install new vectors:

1. Use VECTOR to copy the vector table to a location in the program's memory.
2. As needed, copy the original vector address into the custom routine's code, such as to return control to the original KERNAL routine after performing a custom action.
3. Write the custom routine's address into the vector table in memory.
4. Use VECTOR to install the updated vector addresses.

Some Commodore programmers are accustomed to reading and writing custom vector addresses directly from internal KERNAL memory, without using the VECTOR routine. For the MEGA65, it is important for programs to use the VECTOR accessor routine, and to not depend on the internal KERNAL memory location, to assure compatibility with future versions of the ROM.

NOTE: The KERNAL uses a memory map for BASIC that hides most program memory from the CPU. Custom routines must live in the range $1600 – $1EFF to be visible when BASIC and the screen editor are running. This only applies to cases when the program returns control to BASIC or the screen editor, a common use of vectors. For longer custom routines, you can use a small dispatch routine that disables interrupts, changes the CPU memory map, then calls a routine somewhere else in memory. When that routine returns, the dispatch routine restores the KERNAL memory map and enables interrupts, then jumps to the KERNAL's original vector.

When updating vectors related to interrupt handling, disable interrupts before calling VECTOR.

When installing vectors that return control to the original KERNAL routines, it's a good practice to call RESTOR before reading the table with VECTOR. This ensures that the installation routine sees the original KERNAL addresses, and not the addresses from a previous installation.

Use RESTOR to reset the vector table to the KERNAL defaults.

Vector table:

The vector table is 56 bytes, two bytes for each 16-bit vector address.

Nearly all vectors are for KERNAL jump table routines. Replacing one of these vectors will cause all calls to the KERNAL jump table entry to instead call the custom routine. The custom routine is expected to support the documented preconditions (such as CPU register arguments) of the KERNAL routine. If it subsequently calls the KERNAL routine, it must also meet those preconditions on exit. If it doesn't call the KERNAL routine, it must meet the routine's postconditions (such as CPU register return values) on exit.

The IIRQ, IBRK, and INMI vectors refer to the KERNAL's interrupt handlers. The KERNAL uses a once-per-frame raster IRQ. The KERNAL's interrupt handlers preserve all CPU registers on the stack, pushed in this order: A, X, Y, Z, B. For an IRQ, the handler then tests whether the IRQ is caused by the raster frame or by a BRK instruction, and JMPs to either the IIRQ or IBRK handler accordingly. For an NMI, the handler disables IRQs, then calls the INMI handler. If a custom routine calls the original KERNAL vector when it is finished, the KERNAL will restore the CPU registers, then exit the interrupt handler cleanly. If the custom routine opts to not invoke the original KERNAL vector, it must handle the five bytes on the stack.

CTLVEC, SHFVEC, and ESCVEC allow a custom routine an opportunity to intercept printed and typed characters that involve the Control, Shift, or Escape keys. The accumulator contains the unmodified PETSCII code. A custom routine can cancel further processing by returning with RTS, or it can yield back to the KERNAL by calling the original KERNAL vector. While this is intended primarily to add interactive behaviors to the screen editor when these characters are typed, all such codes can be printed by programs to achieve these effects as well.

KEYSCAN allows a custom routine to intercept typing events as they are read from the keyboard. This vector is called during the KERNAL's implementation of GETIN when the input device is the keyboard. The accumulator contains the unmodified PETSCII code, or $FF to indicate no keypress. Y contains a bitmask of modifier keys; see a description of MODKEY in Appendix F. If the C flag is clear, the typing event is eligible to trigger a function key macro; if set, the event will not trigger a macro. The vector routine can read or change any of these attributes. It must return control to the original KERNAL address in the vector.

Offset	Name	Description
0	IIRQ	Raster IRQ handler
2	IBRK	BRK handler
4	INMI	NMI handler
6	IOPEN	OPEN
8	ICLOSE	CLOSE
10	ICHKIN	CHKIN
12	ICKOUT	CKOUT
14	ICLRCH	CLRCH
16	IBASIN	BASIN
18	IBSOUT	BSOUT
20	ISTOP	STOP
22	IGETIN	GETIN
24	ICLALL	CLALL
26	EXMON	Reserved, do not modify
28	ILOAD	LOAD
30	ISAVE	SAVE
32	ITALK	TALK
34	ILISTEN	LISTN
36	ITALKSA	TKSA
38	ISECOND	SECND
40	IACPTR	ACPTR
42	ICIOUT	CIOUT
44	IUNTALK	UNTLK
46	IUNLISTEN	UNLSN
48	CTLVEC	Control code
50	SHFVEC	Shift code
52	ESCVEC	Escape code
54	KEYSCAN	Handling keyboard input

Examples: The following example installs a short routine at $1600 that increments the screen code in the top left corner of the screen, then sets the IIRQ vector such that the KERNAL calls this routine once per raster frame. After this program returns to BASIC, the screen code updates continuously while BASIC and the screen editor remain operational.

```
vector = $ff8d
```

```
    ; Install custom_irq routine to $1600,
    ; so that it will be visible when BASIC
    ; is active.
    ldx #custom_irq_end-custom_irq+1
-   lda custom_irq-1,x
    sta $1600-1,x
    dex
    bne -

    ; Read vector table into memory
    sei
    jsr restor
    cli
    sec
    ldx #<vectable
    ldy #>vectable
    jsr vector

    ; Copy the iirq vector to custom_irq's
    ; jmp instruction
    lda vectable
    sta $1600+custom_irq_return-custom_irq+1
    lda vectable+1
    sta $1600+custom_irq_return-custom_irq+2

    ; Write custom_irq address to iirq (the first tabl
    lda #$00
    sta vectable
    lda #$16
    sta vectable+1

    ; Install updated vector table
    clc
    ldx #<vectable
    ldy #>vectable
    sei
    jsr vector
    cli

    ; Return to BASIC
    rts

custom_irq:
    ; Rotate the top left character
    inc $0800
custom_irq_return:
    jmp $0000
custom_irq_end:

vectable:
    !fill $38
```

The following example uses the KEYSCAN vector to intercept the Help key. Instead of invoking the F15 function key macro, pressing the Help key changes the border color. The PHP and PLP instructions ensure that the C flag is preserved. Only the vector routine is shown; the installation code is similar to the previous example, using `vectable+54` as the address of the KEYSCAN vector table entry.

```
custom_keyscan:
    php
    cmp #132  ; Help key
    bne +
    lda #$ff  ; cancel "Help"
    inc $d020
+   plp
custom_keyscan_return:
    jmp $0000
custom_keyscan_end:
```