# Sequential Character File Manager (SCF)

## SCF General Description

The Sequential Character File Manager (SCF) is a re-entrant subroutine package for I/O service requests to devices which operate on a character-by-character basis, such as terminals, printers, and modems. SCF can handle any number or type of character-oriented devices. It includes some input and output editing functions for line-oriented operations such as backspace, line delete, repeat line, auto line feed, screen pause, and return delay padding.

The following I/O service requests are handled by SCF:

| | | | | |
|---|---|---|---|---|
| I$Close | I$Create | I$GetStt | I$Open | I$Read |
| I$ReadLn | I$SetStt | I$Write | I$Writln | |

The following I/O service requests are not valid for SCF:

| | | | |
|---|---|---|---|
| I$ChgDir | I$Delete | I$MakDir | I$Seek |

When an I$ChgDir, I$Delete, or I$MakDir is made to SCF, an appropriate error code is returned. I$Seek does not return an error.

The following I/O service requests do not call SCF:

| | | |
|---|---|---|
| I$Attach | I$Detach | I$Dup |

SCF device drivers are responsible for the actual transfer of data between their own internal buffers and the device hardware.

SCF transfers data to/from the driver in register d0. The driver typically operates as follows, depending upon whether or not the driver uses interrupts:

### Polled Mode

The WRITE routine writes the data to the hardware and the driver returns immediately. The READ routine checks for available data, waits if there is no data, and returns the data when ready. Polled-mode drivers usually do not buffer the data internally.

**NOTE:** Polled I/O operation can have a harmful effect on real-time system operation. Polled I/O is acceptable if the device is always ready to send or receive data (for example, output to a memory-mapped video display). Polled I/O is not acceptable if the driver has to wait for the device to send or receive data.

### Interrupt Mode

Interrupt-driven drivers typically use input FIFO and output FIFO buffers for the data being read and written. The WRITE routine deposits the data in the output FIFO buffer, arms the output interrupts (if necessary), and allows the device's output interrupt service routine to empty the output FIFO. When the output FIFO is empty, output interrupts are usually disabled. The READ routine checks the input FIFO buffer. If data is available, READ takes the next character from the buffer and returns. If no data is available, READ suspends itself until data is available. The device's input interrupt service routine is responsible for filling the input FIFO and waking any waiting process. Input interrupts are usually enabled for the time that the device is attached to the system.

## *SCF Line Editing*

The I$Read and I$Write service requests to SCF devices pass data to/from the device without modification; SCF does not add line feeds or NULLs after writing a carriage return.

The I$ReadLn and I$Writln service requests to SCF devices perform all line editing functions enabled for the particular device.

Line editing functions are initialized when a path is first opened by copying the option table from the device descriptor associated with that device into the path descriptor. They may be altered later by programs using the I$GetStt and I$SetStt (SS_Opt) service requests. You can use the xmode utility to modify the option table of SCF device descriptors in writable memory, so that changes can be applied prior to opening a path to the device. You can also use the tmode utility to modify the options from the keyboard. Line editing functions are disabled when the option table field is set to zero.

**CAVEAT:** If software handshaking (X-ON/X-OFF) is enabled, these characters are intercepted by the device driver and not processed by SCF.

### *SCF I/O Service Requests*

When a process makes one of the following system calls to a SCF device, SCF executes the file manager functions described for that call.

I$Close SCF performs the following functions:

- **Checks for additional paths open to the device by the calling process**
  If no additional paths are open, a SS_Relea SetStat is performed to release the device signal conditions and disassociate the device signals from the process.
- **Checks for any other users of the path**
  If there are none, SBF:
  - **Performs a SS_Close SetStat to the driver**
  - **Performs an I$Detach if the device has an output (echo) device**
  - **Returns buffers allocated by the original I$Open call**

I$Create SCF considers this system call synonymous with I$Open.

I$GetStt The SS_Opt GetStat function is supported by SCF. It is passed to the driver to enable the driver to update hardware specific parameters such as the baud rate. If the driver returns an E$UnkSvc error, it is ignored. All other GetStat calls are passed directly to the driver.

Refer to the I$GetStt system call description in the ***OS-9 Technical Manual*** for specific information on the various SCF-oriented I$GetStt functions.

I$Open SCF performs the following functions:

- **Validates the pathname**
- **Allocates memory for the "path buffer"**
- **Initializes the path descriptor with the default options section**
- **Performs an I$Attach if the device has an output (echo) device**
- **Calls the driver with an SS_Open SetStat**
  If the driver returns an E$UnkSvc error, SCF ignores it.

I$Read

I$Read requests read input from the device without modifying the data. The read terminates under any of these circumstances:

- The requested number of bytes has been read.
- An end-of-record character is detected (PD_EOR).
- An end-of-file (PD_EOF) is detected as the first character of the read.
- An error occurs.

You have control over the method of transfer in the following ways:

- De-select (set to zero) the end-of-record (PD_EOR) character using I$GetStt and I$SetStt. This prevents the read from terminating early, due to PD_EOR detection. The read continues until the requested number of characters has been read.
- De-select (set to zero) the end-of-file (PD_EOF) character using I$GetStt and I$SetStt. This prevents the read from terminating when receiving an end-of-file character as the first character of the read.

If the requested data is not immediately available, the driver waits (F$Sleep) for the data. This will "busy" the driver (other processes I/O block) until the data READ request has completed. If you do not wish a process to wait for data, use the SS_Ready GetStat or SS_SSig SetStat calls to detect when an I$Read can be issued.

I$ReadLn

I$ReadLn requests read input from the device and may edit the data. The read terminates under any of these circumstances:

- An end-of-record character is detected (PD_EOR).
- An end-of-file (PD_EOF) is detected as the first character of the read.
- An error occurs.

If the end-of record character is not encountered before the requested number of bytes has been read, SCF echos the line overflow character (PD_OVF) for each subsequent character read. This indicates that the characters are being ignored. This condition is maintained until the end-of-record character is read. You have control over how the data stream is edited by setting the path descriptor options using I$GetStt and I$SetStt.

**NOTE:** *Never* use I$ReadLn on a path that has its end-of-record (PD_EOR) function disabled, as I$ReadLn can then only terminate on an error or end-of-file condition.

I$SetStt The SS_Opt SetStat function is supported by SCF. After SCF updates the path descriptor option section, it is passed to the driver to enable the driver to update hardware specific parameters such as the baud rate. If the driver returns an E$UnkSvc error, SCF ignores it. All other SetStat calls are passed directly to the driver.

Refer to the I$SetStt system call description in the ***OS-9 Technical Manual*** for specific information on the various SCF-oriented I$SetStt functions.

I$Write I$Write requests output data to the device without modifying the data being passed. The write terminates only when all characters have been sent or an error occurs.

I$Writln I$Writln is similar to I$Write except that I$Writln writes data until an end-of-record character (PD_EOR) is written or until the specified number of bytes has been sent. The line editing that I$Writln performs for SCF devices consists of auto line feed, null byte padding at end-of-record, tabulation, and auto page pause.

## SCF Device Descriptor Modules

This section describes the definitions of the initialization table contained in device descriptor modules for SCF devices. The initialization table immediately follows the standard device descriptor module header fields and defines initial values for the I/O editing features. The size of the table is defined in the M$Opt field.

| Device Descriptor Offset | Path Descriptor Label | Description |
|---|---|---|
| $48 | PD_DTP | Device Type |
| $49 | PD_UPC | Upper Case Lock |
| $4A | PD_BSO | Backspace Option |
| $4B | PD_DLO | Delete Line Character |
| $4C | PD_EKO | Echo |
| $4D | PD_ALF | Automatic Line Feed |
| $4E | PD_NUL | End Of Line Null Count |
| $4F | PD_PAU | End Of Page Pause |
| $50 | PD_PAG | Page Length |
| $51 | PD_BSP | Backspace Input Character |
| $52 | PD_DEL | Delete Line Character |
| $53 | PD_EOR | End Of Record Character |
| $54 | PD_EOF | End Of File Character |
| $55 | PD_RPR | Reprint Line Character |
| $56 | PD_DUP | Duplicate Line Character |
| $57 | PD_PSC | Pause Character |
| $58 | PD_INT | Keyboard Interrupt Character |
| $59 | PD_QUT | Keyboard Abort Character |
| $5A | PD_BSE | Backspace Output |
| $5B | PD_OVF | Line Overflow Character (bell) |
| $5C | PD_PAR | Parity Code, # of Stop Bits, and # of Bits/Character |
| $5D | PD_BAU | Adjustable Baud Rate |
| $5E | PD_D2P | Offset To Output Device Name |
| $60 | PD_XON | X-ON Character |
| $61 | PD_XOFF | X-OFF Character |
| $62 | PD_TAB | Tab Character |
| $63 | PD_TABS | Tab Column Width |

**NOTE:** In this table the offset values are the device descriptor offsets, while the labels are the path descriptor offsets. To correctly access these offsets in a device descriptor using the path descriptor labels, you must make the following adjustment: (M$DTyp - PD_OPT).

For example, to access the letter case in a device descriptor, use PD_UPC + (M$DTyp - PD_OPT). To access the letter case in the path descriptor, use PD_UPC. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l or usr.l.

**NOTE:** You can change or disable most of these special editing functions by changing the corresponding control character in the path descriptor. Do this with the I$SetStt service request, the tmode utility, or the xmode utility.

| Name | Description |
|---|---|
| PD_DTP | **Device Type**<br>Set to zero for SCF devices. |
| PD_UPC | **Letter case**<br>If PD_UPC is not equal to zero, input or output characters in the range "a..z" are made "A..Z". |
| PD_BSO | **Destructive Backspace**<br>If PD_BSO is zero when a backspace character is input, SCF echoes PD_BSE (backspace echo character). If PD_BSO is non-zero, SCF echoes PD_BSE, space, PD_BSE. |
| PD_DLO | **Delete**<br>If PD_DLO is zero, SCF deletes by backspace-erasing over the line. If PD_DLO is not zero, SCF deletes by echoing a carriage return/line-feed. |
| PD_EKO | **Echo**<br>If PD_EKO is not zero, then all input bytes are echoed, except undefined control characters which are printed as periods. If PD_EKO is zero, input characters are not echoed. |
| PD_ALF | **Automatic line feed**<br>If PD_ALF is not zero, carriage returns are automatically followed by line-feeds. |
| PD_NUL | **End of line null count**<br>Indicates the number of NULL padding bytes to be sent after a carriage return/line-feed character. |
| PD_PAU | **End of page pause**<br>If PD_PAU is not zero, an auto page pause occurs upon reaching a full screen of output. See PD_PAG for setting page length. |

| Name | Description |
|---|---|
| PD_PAG | **Page length**<br>Contains the number of lines per screen (or page). |

PD_BSP **Backspace "input" character**
Indicates the input character recognized as backspace. See PD_BSE and PD_BSO.

PD_DEL **Delete line character**
This field indicates the input character recognized as the delete line function. See PD_DLO.

PD_EOR **End of record character**
This field defines the last character on each line entered (I$Read, I$ReadLn). An output line is terminated (I$Writln) when this character is sent. Normally PD_EOR should be set to $0D. **WARNING:** If PD_EOR is set to zero, SCF's I$ReadLn will *never* terminate, unless an EOF or error occurs.

PD_EOF **End of file character**
This field defines the end-of-file character. SCF returns an end-of-file error on I$Read or I$ReadLn if this is the first (and only) character input.

PD_RPR **Reprint line character**
If this character is input, SCF (I$ReadLn) reprints the current input line. A carriage return is also inserted in the input buffer for PD_DUP (see below) to make correcting typing errors more convenient.

PD_DUP **Duplicate last line character**
If this character is input, SCF (I$ReadLn) duplicates whatever is in the input buffer through the first PD_EOR character. Normally, this is the previous line typed.

PD_PSC **Pause character**
If this character is typed during output, output is suspended before the next end-of-line. This also deletes any "type ahead" input for I$ReadLn.

PD_INT **Keyboard interrupt character**
If this character is input, SCF sends a keyboard interrupt signal to the last user of this path. It terminates the current I/O request (if any) with an error identical to the keyboard interrupt signal code. PD_INT is normally set to a control-C character.

**Name** **Description**

PD_QUT **Keyboard abort character**
If this character is input, SCF sends a keyboard abort signal to the last user of this path. It terminates the current I/O request (if any) with an error code identical to the keyboard abort signal code. PD_QUT is normally set to a control-E character.

PD_BSE **Backspace "output" character (echo character)**
This field indicates the backspace character to echo when PD_BSP is input. See PD_BSP and PD_BSO.

PD_OVF **Line overflow character**

If I$ReadLn has satisfied its input byte count, SCF ignores any further input characters until an end-of-record character (PD_EOR) is received. It echoes the PD_OVF character for each byte ignored. PD_OVF is usually set to the terminal's bell character.

PD_PAR **Parity code, number of stop bits & bits/character**

Bits zero and one indicate the parity as follows:

0 = no parity
1 = odd parity
3 = even parity

Bits two and three indicate the number of bits per character as follows:

0 = 8 bits/character
1 = 7 bits/character
2 = 6 bits/character
3 = 5 bits/character

Bits four and five indicate the number of stop bits as follows:

0 = 1 stop bit
1 = 1 1/2 stop bits
2 = 2 stop bits

Bits six and seven are reserved.

**Name** **Description**

PD_BAU **Software adjustable baud rate**

This one-byte field indicates the baud rate as follows:

| | | |
|---|---|---|
| 0 = 50 baud | 6 = 600 baud | C = 4800 baud |
| 1 = 75 baud | 7 = 1200 baud | D = 7200 baud |
| 2 = 110 baud | 8 = 1800 baud | E = 9600 baud |
| 3 = 134.5 baud | 9 = 2000 baud | F = 19200 baud |
| 4 = 150 baud | A = 2400 baud | 10 = 38400 baud |
| 5 = 300 baud | B = 3600 baud | FF = External |

PD_D2P **Offset to output device descriptor name string**

SCF sends output to the device named in this string. Input comes from the device named by the M$PDev field. This permits two separate devices (a keyboard and video display) to be one logical device. Usually PD_D2P refers to the name of the same device descriptor in which it appears.

PD_XON **X-ON character**
See PD_XOFF below.

PD_XOFF **X_OFF character**
The X-ON and X-OFF characters are used to support software handshaking. Output from a SCF device is halted immediately when PD_XOFF is received and will not be resumed until PD_XON is received. This allows the distant end to control its incoming data stream. Input to a SCF device is controlled by the driver. If the input FIFO is nearly full, the driver sends PD_XOFF to the distant end to halt input. When the FIFO has been emptied sufficiently, the driver resumes input by sending the PD_XON character. This allows the driver to control its incoming data stream.

**NOTE:** When software handshaking is enabled, the driver consumes the PD_XON and PD_XOFF characters itself.

PD_Tab **Tab character**
In I$WritLn calls, SCF expands this character into spaces to make tab stops at the column intervals specified by PD_Tabs. **NOTE:** SCF does not know the effect of tab characters on particular terminals. Tab characters may expand incorrectly if they are sent directly to the terminal.

PD_Tabs **Tab field size**
See PD_Tab.

## *SCF Path Descriptor Definitions*

The first 27 fields of the path options section (PD_OPT) of the SCF path descriptor are copied directly from the SCF device descriptor initialization table. The table is shown on the following page.

The fields can be examined or changed using the I$GetStt and I$SetStt service requests or the tmode and xmode utilities.

You may disable the SCF editing functions by setting the corresponding control character value to zero. For example, if you set PD_INT to zero, there is no "keyboard interrupt" character.

**NOTE:** Full definitions for the fields copied from the device descriptor are available in the previous section. The additional path descriptor fields are defined below:

| Name | Description |
|---|---|
| PD_TBL | **Device Table Entry**<br>Contains a user-visible copy of the device table entry for the device. |
| PD_COL | **Current Column**<br>Contains the current column position of the cursor. |
| PD_ERR | **Most Recent Error Status**<br>Contains the most recent I/O error status. |

| Offset | Name | Description |
|---|---|---|
| $80 | PD_DTP | Device Type |
| $81 | PD_UPC | Upper Case Lock |
| $82 | PD_BSO | Backspace Option |
| $83 | PD_DLO | Delete Line Character |
| $84 | PD_EKO | Echo |
| $85 | PD_ALF | Automatic Line Feed |
| $86 | PD_NUL | End Of Line Null Count |
| $87 | PD_PAU | End Of Page Pause |
| $88 | PD_PAG | Page Length |
| $89 | PD_BSP | Backspace Input Character |
| $8A | PD_DEL | Delete Line Character |
| $8B | PD_EOR | End Of Record Character |
| $8C | PD_EOF | End Of File Character |
| $8D | PD_RPR | Reprint Line Character |
| $8E | PD_DUP | Duplicate Line Character |
| $8F | PD_PSC | Pause Character |
| $90 | PD_INT | Keyboard Interrupt Character |
| $91 | PD_QUT | Keyboard Abort Character |
| $92 | PD_BSE | Backspace Output |
| $93 | PD_OVF | Line Overflow Character (bell) |
| $94 | PD_PAR | Parity Code, # of Stop Bits, and # of Bits/Character |
| $95 | PD_BAU | Adjustable Baud Rate |
| $96 | PD_D2P | Offset To Output Device Name |
| $98 | PD_XON | X-ON Character |
| $99 | PD_XOFF | X-OFF Character |
| $9A | PD_TAB | Tab Character |
| $9B | PD_TABS | Tab Column Width |
| $9C | PD_TBL | Device Table Entry |
| $A0 | PD_Col | Current Column |
| $A2 | PD_Err | Most Recent Error Status |
| $A3 | | Reserved |

**NOTE:** ***Offset*** refers to the location of a path descriptor field, relative to the starting address of the path descriptor. Path descriptor offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: sys.l or usr.l.

## SCF Device Drivers

SCF device drivers support I/O devices that read and write data one character at a time, such as serial devices.

Generally, the input data (usually from a keyboard) is buffered by the driver's interrupt service routine. Each read request returns one character at a time from the driver's circular input FIFO buffer. If the buffer is empty when the request occurs, the driver must suspend the calling process until an input character is received. Input interrupts are usually enabled throughout the time the device is attached to the system. If the device is incapable of interrupt-driven operation, the driver must poll the device until the data becomes available. This situation has a harmful effect on real-time system performance.

The output data may or may not be buffered, depending on the physical characteristics of the output device. If the device is a memory-mapped video display driven by the main CPU, buffering and interrupts are not usually needed. If the device is a serial interface, use buffering and interrupts. Each write request passes a single output character to the driver which is placed in a circular FIFO output buffer. The output interrupt routine takes output characters from this buffer. If the buffer is full when a write request is made, the driver should suspend the calling process until the buffer empties sufficiently.

The I$GetStt system call (SS_Ready) and I$SetStt system call (SS_SSig) permit an application program to determine if the input buffer contains any data. By checking first, the program is not suspended if data is not available.

The driver may optionally handle full input buffer conditions using X-ON/X-OFF or similar protocols. The input routine must also handle the special pause, abort, and quit control characters. All other control characters (such as backspace, line delete, etc.) are handled at the file manager level.

## *Special Characters and NULLs*

Line-editing functions (if any) are generally dealt with at the file manager level by SCF. Device drivers are, however, required to deal with the following special characters in their input character routine:

- **NULL character**
  The driver's input routine should first determine if the received character is a NULL. If so, it should skip all special character tests, because the disabled state of these special characters is indicated by a NULL in the appropriate path option field. Failure to check for a received NULL results in erratic terminal and/or line-editing operation.

- **Abort and Interrupt Characters**
  The abort and interrupt characters should cause the appropriate signal to be sent to the last process that used the device. The received character should then be buffered.

- **Page Pause**
  The page pause character should cause a page pause request to be set in the echo device's static storage. The received character should then be buffered.

- **Software Flow Control**
  The start and stop transmission characters should cause the resumption/suspension of output data transmission. When this protocol is used, these characters are consumed by the driver's input character routine.

## *Parity Stripping*

SCF device drivers do not usually modify the raw data stream when receiving and transmitting data. The drivers are expected to pass eight-bit data characters "as is." When parity is enabled, however, the driver may have to be sensitive to the issue of "parity stripping."

For eight-bit data characters, parity is not normally an issue (except for error checking), because the character parity status is signalled "out-of-band" from the character itself (there is a parity-error status flag). For smaller sized data characters (for example, seven-bit characters), the hardware sometimes passes the value of the parity bit in the high-bit of the received character. If a driver supports parity checking and non-eight-bit character formats, then the driver's input character routine must be sensitive to the current communications mode and strip the parity flag from the data prior to processing and buffering the character. Failure to strip this parity value from the received character may cause erratic terminal operation (for example, the software flow control characters may not be recognized correctly).

## *Data Flow Control*

Data flow control is the process used to control the transfer of data over the physical interface. It ensures that each end of the connection only transmits data when the other end is capable of receiving data. The data flow may be controlled by either hardware and/or software:

### Hardware Flow Control

Hardware flow control uses physical signal lines to indicate the state of the interface. The Ready To Send (RTS) and Clear To Send (CTS) signals on the RS-232 Standard Interface are examples of these physical lines.

The level of implementation of hardware handshaking in a SCF driver is determined by the capabilities of the serial interface itself, which include the capabilities of the interface-chip and the board-level implementation of the interface.

A driver that implements fully functional hardware flow control performs the following functions:

- Configures the transmitter to only send data when the distant end's "ready-to-receive" is active.
- Controls the distant end's "ready-to-transmit" line so that input buffer over-runs do not occur.
- Supports the SS_EnRTS, SS_DsRTS, SS_DCDOn, and SS_DCDOff SetStat calls, to allow a user application to directly control/monitor the serial connection.

A driver that provides minimal (or no) support for hardware flow control usually configures the hardware control lines so that the interface is "ready" whenever the device is attached. Drivers that provide this level of operation usually implement software flow control.

### Software Flow Control

Software flow control uses a software protocol to indicate the "ready" state of the two ends of the interface.

Support for software flow control is provided via the PD_XON (start transmission) and PD_XOFF (stop transmission) fields of the device descriptor. When these fields are enabled (both non-zero), then the driver implements the protocol as follows:

- If the driver receives the stop transmission character, it should immediately suspend data transmission. The driver can resume transmission when a start transmission character is received. Thus, the distant end is allowed to control its incoming data stream.

- If the driver's input routine detects that its input buffer is about to fill, then it causes a stop transmission character to be sent to the distant end. When the buffer has been sufficiently emptied, the driver can cause transmission of a start transmission character. Thus, the driver is capable of controlling its incoming data stream.

When implementing software flow control, note the following points:

- The start transmission and stop transmission characters are *consumed* by the driver's input routine. If pure binary transfers are desired (the character values for flow control are actually part of the data stream), then software flow control must be disabled and hardware flow control enabled.
- Software flow control only works reliably with interrupt-driven drivers, because the detection of the incoming stop transmission character must take place immediately.
- The characters involved with the protocol must be "agreed upon" by both ends of the connection. Most systems default to the ASCII control characters X-ON and X-OFF. However, any other pair of characters may be used if both ends concur.
- When controlling the input data, the driver's input routine and Read routine will cooperate in the protocol as follows:
    - The input routine detects a "high-water" mark; a point at which the input buffer is almost full. When this mark is reached (ten characters remaining in buffer), the input routine causes the stop transmission character to be sent. The "head room" provided by the high-water mark should be set so that the distant end has time to suspend transmission before the buffer actually fills.
    - The Read routine simply takes characters from the input buffer until the buffer count reaches the "low-water" mark. Then, the Read routine causes the start transmission character to be sent to resume input. The low-water mark is usually set to a low value to keep the total overhead in the software flow control to a minimum.

## *SCF Device Driver Storage Definitions*

SCF device driver modules contain a package of subroutines that perform raw I/O transfers to or from a specific hardware controller. Because these modules are re-entrant, one copy of the module can simultaneously run several identical I/O controllers.

The kernel allocates a static storage area for each device (which may control several drives). The size of the storage area is given in the device driver module header (M$Mem). Some of this storage area is required by the kernel and SCF; the device driver may use the remainder in any manner. Information on device driver static storage required by the operating system can be found in the scfstat.a DEFS file. Static storage is used as follows:

| Offset | Name | Maintained By | Description |
|---|---|---|---|
| $00 | V_PORT | Kernel | Device base address |
| $04 | V_LPRC | File Manager | Last active process ID |
| $06 | V_BUSY | File Manager | Active process ID |
| $08 | V_WAKE | Driver | Process ID to awaken |
| $0A | V_Paths | Kernel | Linked list of open paths |
| $0E | | | Reserved |
| $2E | V_DEV2 | Kernel | Addr. of attached device static storage |
| $32 | V_TYPE | File Manager | Device type or parity |
| $33 | V_LINE | File Manager | Lines left until end of page |
| $34 | V_PAUS | Driver/File Man. | Pause request |
| $35 | V_INTR | File Manager | Keyboard interrupt character |
| $36 | V_QUIT | File Manager | Keyboard abort character |
| $37 | V_PCHR | File Manager | Pause character |
| $38 | V_ERR | Driver | Error accumulator |
| $39 | V_XON | File Manager | X-ON character |
| $3A | V_XOFF | File Manager | X-OFF character |
| $3B | | | Reserved |
| $3C | V_Presvd | | Reserved |
| $46 | V_Hangup | Driver/File Man. | Path lost flag |
| $54 | | | Device Driver Variables begin here |

**NOTE:** ***Offset*** refers to the location of a static storage field, relative to the starting address of the static storage area. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: sys.l.

| Name | Description |
|---|---|
| V_PORT | **Device base address**<br>The device's physical port address. It is copied from M$Port in the device descriptor when the device is attached by the kernel. |
| V_LPRC | **Last active process ID**<br>The process ID of the last process to use the device. The IRQ service routine sends this process the proper signal when an interrupt or quit character is received. |
| V_BUSY | **Current active process**<br>The process ID of the process currently using the device. It is used to implement I/O Blocking by SCF. This field is also used by the interrupt drivers when they wish to suspend themselves, by copying V_BUSY to V_WAKE (prior to suspending themselves). A value of zero indicates the device is not busy. |
| V_WAKE | **Process ID to awaken**<br>The process ID of any process that is waiting for the device to complete I/O. A value of zero indicates that no process is waiting. V_WAKE is set by the driver from V_BUSY and provides the interlock between the driver and the driver's interrupt service routine. |
| V_PATHS | **Linked list of open paths**<br>A singly-linked list of all paths currently open on this device. |
| V_DEV2 | **Attached device static storage**<br>The address of the echo (output) device's static storage area. A device is typically its own echo device, but may not be, as in the case of a keyboard and a memory mapped video display. The interrupt service routine uses this pointer to set an output pause request (see V_PAUS and V_PCHR). If the value in V_DEV2 is zero, there is no echo device. |
| V_TYPE | **Device type or parity**<br>This value is copied from PD_PAR in the path descriptor by SCF, so that it may be used by interrupt service routines, if required. |
| V_LINE | **Lines left until end of page**<br>The number of lines left until the end of the page. Paging is handled by SCF. |

| Name | Description |
|---|---|
| V_PAUS | **Pause request**<br>A flag used to signal SCF that a pause character has been received. Setting its value to anything other than 0 causes SCF to stop transmitting characters at the end of the next line. Device driver input routines must set V_PAUS in the echo device's static storage area. SCF checks this value in the echo device's static storage when output is sent. Once paused, SCF clears any type-ahead (I$ReadLn), waits for and consumes the next input character, clears V_PAUS, and resumes output (see V_DEV2 and V_PCHR). |
| V_INTR | **Keyboard interrupt characters**<br>This value is copied from PD_INT in the path descriptor by SCF, so that it may be used by the driver's input routine. Receipt of this character should cause a signal (S$Intrp) to be sent to the last user of the device (V_LPRC). |
| V_QUIT | **Quit character**<br>This value is copied from PD_QUT in the path descriptor by SCF so that it may be used by the driver's input routine. Receipt of this character should cause a signal (S$Quit) to be sent to the last user of the device (V_LPRC). |
| V_PCHR | **Pause character**<br>This value is copied from PD_PSC in the path descriptor by SCF, so that it may be used by the driver's input routine. When the input routine receives this character, it should set the output pause request flag (V_PAUS) in the echo device's static storage (V_DEV2). (See V_DEV2 and V_PAUS.) |
| V_ERR | **Error accumulator**<br>This location is used to accumulate I/O errors. Typically, the IRQ service routine uses it to record input errors so that they may be reported later when SCF calls the device driver read routine. |
| V_XON | **X-ON character**<br>This character is copied from PD_XON of the path descriptor by SCF, so that it may be used for software handshaking by interrupt service routines, if required. |
| V_XOFF | **X-OFF character**<br>This character is copied from PD_XOFF of the path descriptor by SCF, so that it may be used for software handshaking by interrupt service routines, if required. |
| V_Hangup | **Path Lost Flag**<br>This flag should be set to a non-zero value when the driver detects that the path has been lost (for example, carrier lost on a modem). |

## *Linking SCF Drivers*

After a SCF driver has been assembled into its relocatable object file (ROF), the driver needs to be linked to produce the final driver module. Linking resolves all code references in drivers that are comprised of several ROF files. It also resolves the external data and static storage references by the driver.

The most important part of linking is to correctly resolve the static storage references. Generally, the static storage area is composed of two sections, in this order (see Figure 3-1):

- I/O globals
- Driver-declared variables

The driver-declared variables are declared in vsect areas of the driver, but they *must* be allocated after the I/O globals. To allocate all of the storage, *in the correct order*, the scfstat.l *must* be the first module specified. The scfstat.l file is usually found in the system's LIB directory. The following is a typical linker command line for an SCF driver:

```
l68 /dd/LIB/scfstat.l REL/sc335.r -O=OBJS/sc335
```

**NOTE:** Failure to link the I/O global storage first, or not at all, results in erratic driver operation.

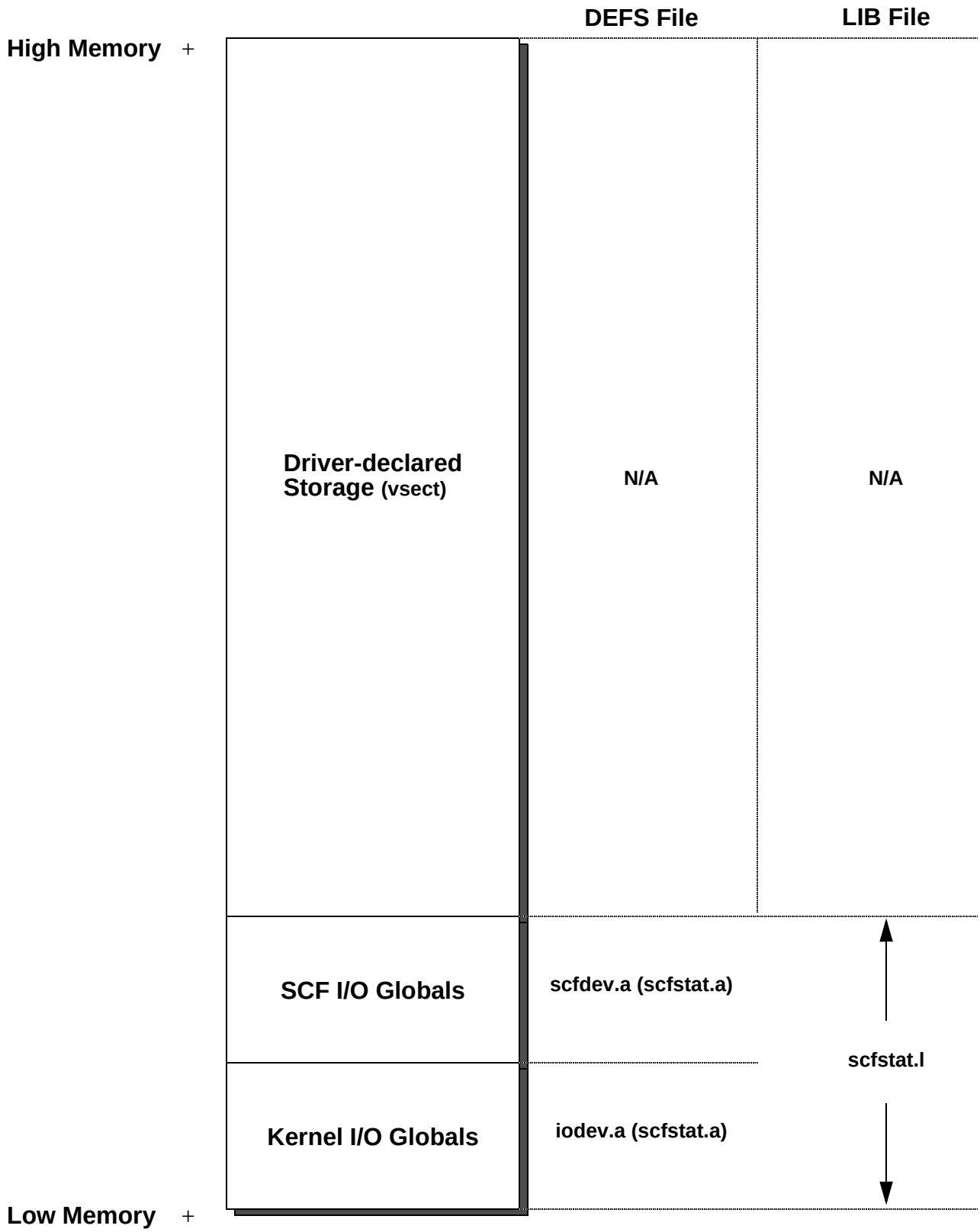


*Figure 3-1: SCF Static Storage Layout*

## SCF Device Driver Subroutines

As with all device drivers, SCF device drivers use a standard executable memory module format with a module type of Drivr (code $E0). SCF drivers are called in system state.

**NOTE:** I/O system modules must have the following module attributes:

- They must be owned by a super-user (0.n).
- They must have the system-state bit set in the attribute byte of the module header. (OS-9 does not currently make use of this, but future revisions will require that I/O system modules be system-state modules.)

The execution offset address in the module header points to a branch table that has seven entries. Each entry is the offset of the corresponding subroutine. The branch table appears as follows:

```
ENTRY dc.w INIT          initialize device
      dc.w READ          read character
      dc.w WRITE         write character
      dc.w GETSTAT       get device status
      dc.w SETSTAT       set device status
      dc.w TERM          terminate device
      dc.w TRAP          handle illegal exception (0 = none)
```

Each subroutine should exit with the carry bit of the condition code register cleared, if no error occurred. Otherwise, set the carry bit and return an appropriate error code in the least significant word of register d1.w.

The TRAP entry point is currently not used by the kernel, but in the future will be defined as the offset to error exception handling code. Because no handler mechanism is currently defined, this entry point should be set to zero to ensure future compatibility.

The following pages describe each subroutine.

# INIT — Initialize Device and its Static Storage

***INPUT:*** **(a1) = address of device descriptor module**
**(a2) = address of device static storage**
**(a4) = process descriptor pointer**
**(a5) = caller's register stack pointer**
**(a6) = system global data pointer**

***OUTPUT:*** **None**

***ERROR OUTPUT:*** **cc = carry bit set**
**d1.w = error code**

***FUNCTION:*** The INIT routine must:

- ¨ Initialize the device static storage.
- ¦ Initialize the device control registers.
- Æ Place the driver IRQ service routine on the IRQ polling list by using the F$IRQ service request, if required.
- Ø Enable interrupts if necessary.

Prior to being called, the device static storage is cleared (set to zero) except for V_PORT which contains the device port address. Do not initialize the portion of static storage used by SCF.

If INIT returns an error, it does not have to clean up its operation, for example, remove device from polling table or disable hardware. The kernel calls TERM to allow the driver to clean up INIT's operation before returning to the calling process.

**NOTE:** If the INIT routine causes an interrupt to occur, the interrupt can be handled in one of the following ways:

- Process the interrupt directly by masking interrupts to the level of the device, polling/servicing the device hardware, and then restoring the previous interrupt level. This is the preferred technique unless the interrupt is time-consuming.
- Allow the interrupt service routine to service the hardware. In this case, the process descriptor contains the process ID (P$ID) to which V_WAKE should be set. V_BUSY cannot be used because it is zero when INIT is called.

# READ Get Next Character

***INPUT:*** **(a1) = address of path descriptor**
**(a2) = address of device static storage**
**(a4) = process descriptor pointer**
**(a5) = caller's register stack pointer**
**(a6) = system global data pointer**

***OUTPUT:*** **d0.b = input character**

***ERROR OUTPUT:*** **cc = carry bit set**
**d1.w = error code**

***FUNCTION:*** This routine returns the next character available. Depending upon whether or not the routine is interrupt-driven, READ typically operates as follows:

### Polled I/O Mode

A polled I/O read routine checks the hardware for available data. If there is none, the routine must wait until data is available. When data is available, READ should strip parity (if required) and then determine whether or not the character requires special handling:

- If the character is the output pause character (V_PCHR), READ sets a pause request (V_PAUS) in the echo device's static storage (V_DEV2).
- If the character is a keyboard interrupt (V_INTR) or quit (V_QUIT) character, READ sends the appropriate signal to the last process to use the device (V_LPRC).

**NOTE:** If the received character is a NULL character, then special character tests should be ignored.

**NOTE:** Software handshaking, as specified by V_XON/V_XOFF is not usually implemented for polled-mode I/O, as the lack of interrupt-driven operation makes this handshake feature unreliable. Polled I/O drivers can usually only perform hardware handshaking.

The character read is returned to SCF in register d0.

### Interrupt I/O Mode

For interrupt-driven drivers, READ gets data from the driver's input FIFO buffer. This buffer is filled by the input interrupt service routine. The following describes how READ operates.

- READ determines if another process has set up a "send signal on data ready"

condition. If so, READ returns a "not ready" (E$NotRdy) error (the device is busy for reading, but not for writing).

¦ READ then determines if data is available in the input FIFO buffer. If not, the driver should suspend itself by copying its process ID from V_BUSY to V_WAKE and then performing an F$Sleep service request to put itself to sleep indefinitely.

When the driver awakens, either data is available in the FIFO or a signal occurred. If a signal occurred, either the signal value is in P$Signal (process descriptor) or the process is condemned (condemn bit set in P$State). If the process is condemned or the signal value is deadly to I/O (less than S$Deadly), then the driver should return immediately to SCF with the carry bit set and the signal code (if any) as the error code.

Æ READ should get the next character from the input FIFO.

Ø If software handshaking is implemented, READ should determine if input has been halted (V_XOFF sent to distant end). If so, and reading this character causes the FIFO count to go below the "low-water mark" of the FIFO, then resume input by sending a V_XON character to the distant end and flagging input resumed.

× READ should determine if any errors have been logged by the input interrupt service routine (V_ERR). If so, READ returns an error (E$Read) to SCF and clears V_ERR. Otherwise, READ returns the character read to SCF in register d0.

**NOTE:** Data buffers for queueing data between the main driver and the IRQ service routine are *not* automatically allocated by SCF. They should be defined in the device driver's static storage area (vsect) or allocated dynamically by the driver (for example, at INIT call).

**NOTE:** Normally, READ should not have to enable the device's "data-buffer-full" interrupt. The device should normally be configured so that any input while the device is attached causes an interrupt. This is usually done during INIT. Input interrupts are typically disabled only when the device is detached (TERM routine).

# WRITE — Output a Character

***INPUT:*** **d0.b = character to write**
**(a1) = address of the path descriptor**
**(a2) = address of device static storage**
**(a4) = process descriptor pointer**
**(a5) = caller's register stack pointer**
**(a6) = system global data pointer**

***OUTPUT:*** **None**

***ERROR OUTPUT:*** **cc = carry bit set**
**d1.w = error code**

***FUNCTION:*** The WRITE routine writes a character. Depending upon whether or not the routine is interrupt-driven, WRITE typically operates as follows:

### Polled I/O Mode

A polled I/O driver checks the hardware for "ready-to-transmit". When ready, the character is written to the hardware and the driver returns to SCF without an error.

### Interrupt I/O Mode

For interrupt-driven drivers, WRITE attempts to put the character into the driver's output FIFO buffer and then ensures that output interrupts are enabled. The driver's output interrupt service routine empties the output FIFO. WRITE operates as follows:

¨ WRITE determines if space is available in the output FIFO buffer. If not, the device driver should suspend itself by copying its process ID from V_BUSY to V_WAKE and then performing a F$Sleep service request to put itself to sleep indefinitely.

When the driver awakens, either space is available in the output FIFO or a signal occurred. If a signal occurred, either the signal value is in P$Signal (process descriptor) or the process is condemned (condemn bit set in P$State). If the process is condemned or the signal value is deadly to I/O (less than S$Deadly), the driver should return immediately to SCF with the carry bit set and the signal code (if any) as the error code.

¦ WRITE puts the character into the output FIFO buffer.

Æ WRITE determines if output interrupts are currently enabled. If so, this implies that output is currently active (using the output IRQ service routine) and the driver can simply return to SCF without an error.

Ø If output interrupts are disabled, then output is halted due to software handshaking (V_XOFF received from distant end) or a previously empty output

FIFO. If output is halted due to software handshaking, the driver should return to SCF without an error. Otherwise, the driver should enable output interrupts on the device (allowing the output interrupt service routine to empty the output FIFO) and return to SCF without an error.

**NOTE:** Data buffers for queueing data between the main driver and the IRQ service routine are *not* automatically allocated by SCF. They should be defined in the device driver's static storage area (vsect) or allocated dynamically by the driver (for example, at INIT call).

**NOTE:** Typically, this routine should ensure that output interrupts are enabled only when necessary. After an output interrupt is generated, the IRQ service routine continues to transmit data until the output FIFO is empty and then it typically disables the device's "ready-to-transmit" interrupts.

This dynamic enabling/disabling of the device's transmit interrupts is essential to some serial devices, as the "transmit ready" interrupt is generated every "character period" (that is, at the device's baud rate), regardless of whether a character is actually transmitted. This type of situation leads to excessive and unnecessary overhead to the system, and should be avoided.

# GETSTAT/SETSTAT Get/Set Device Status

***INPUT:*** **d0.w = function code**
**(a1) = address of path descriptor**
**(a2) = address of device static storage**
**(a4) = process descriptor pointer**
**(a5) = caller's register stack pointer**
**(a6) = system global data pointer**

***OUTPUT:*** **Depends upon function code**

***ERROR OUTPUT:*** **cc = carry bit set**
**d1.w = error code**

***FUNCTION:*** These routines are wild-card calls used to get/set the device's operating parameters as specified for the I$GetStt and I$SetStt service requests.

Calls which involve parameter passing require the driver to examine or change the register stack variables. These variables contain the contents of the MPU registers at the time the I$GetStt/I$SetStt request was made. Parameters passed to the driver are set up by the caller prior to using the service call. Parameters passed back to the caller are available when the service call completes.

Typical SCF drivers handle the following I$GetStt/I$SetStt calls:

***I$Getstt:*** SS_EOF, SS_Opt, SS_Ready

***I$SetStt:*** SS_Break, SS_DCOff*, SS_DCOn*, SS_DsRTS, SS_EnRTS, SS_Open, SS_Opt, SS_Relea*, SS_SSig*,

* only for interrupt-driven drivers

Any unsupported I$GetStt/I$SetStt calls to the driver should return an unknown service error (E$UnkSvc).

**NOTE:** A minimal SCF driver should support SS_Ready and SS_EOF, and if interrupt-driven, SS_SSig.

The following pages describe the driver's role in the implementation of the above I$GetStt/I$SetStt calls.

### *GetStat Calls:*

SS_EOF This routine should exit without an error.

SS_Opt This routine is called when SCF is asked to return the current path options. SCF calls the driver so that the driver can update the path descriptor's baud rate (PD_BAU) and communications mode (PD_PAR) to the current hardware values. This function is usually done by drivers that support dynamic changes to baud rate, etc. Drivers that do not support these changes typically return an unknown service request error (E$UnkSvc).

SS_Ready This routine returns the current count of data available in the input FIFO buffer. If data is available, the count should be returned in the caller's d1 register (R$d1 offset from passed a5) and the driver should return to SCF without an error. If no data is available, then a "not ready" error (E$NotRdy) should be returned to SCF.

### *SetStat Calls:*

SS_Break This routine is called when an application wishes to assert a "break" condition on the outgoing serial line.

SS_DCOff
SS_DCOn These routines are called when you wish to notify an application that the Data Carrier has been asserted(SS_DCOn) or negated (SS_DCOff). Typically, this routine saves the process ID (PD_CPR), path number (PD_PD), and signal code (user's d2 register) in static storage and then returns without error. The IRQ service routine detects the presence or loss of the Data Carrier, sends the signal, and clears down the signal condition.

Drivers which have hardware detection of a change-of-state only on the Data Carrier line typically have to track the current state (asserted or negated) of the line and signal a change of state accordingly.

**NOTE:** Only interrupt-driven drivers should implement these calls.

SS_DsRTS These routines are called by applications that wish to SS_EnRTS explicitly assert (SS_EnRTS) or negate (SS_DsRTS) the RTS handshake line. Typically, the driver performs the hardware action and returns without an error.

SS_Open This routine is called by SCF whenever a new path to the device is opened. Typically, drivers handle this call in the same way as a SetStat (SS_Opt) call, i.e. check for baud-rate, configuration mode changes.

SS_Opt This routine is called when SCF is asked to change the current path options. SCF passes the call to the driver so that it may implement baud-rate, configuration mode, etc., changes to the hardware. Typically, the driver checks PD_BAU and PD_PAR to determine if they have changed. If not, the driver simply returns without an error. If one or both of these have changed, the driver validates the requested change and if correct, implements the change in hardware (for example, new baud rate). If the request is for an unsupported or illegal I/O mode (for example, invalid stop-bit count), then the driver typically returns a "bad I/O mode" error (E$BMode) and refuses the change.

SS_Relea This routine is called when either SCF or an application wishes to clear down device signalling. This routine should erase any pending signal conditions (due to SS_SSig, SS_DCOn, SS_DCOff) and return without an error.

**NOTE:** When clearing down the signal condition(s), the driver should only clear the signal if the process ID (PD_CPR) and path number (PD_PD) of the caller match the process ID and path number of the original set-up call.

SS_SSig This routine is called when applications wish to have a signal sent to them when input data is available. Typically, the routine operates as follows:

- ¨ It determines if another process has set up a SS_SSig condition. If so, a "not ready" error (E$NotRdy) is returned.
- ¦ It determines if data is available in the input FIFO buffer. If so, the specified signal (user's d2 register value) is sent to the process (PD_CPR) and the routine returns.
- Æ If no data is available, the process ID, path number (PD_PD), and signal are saved in static storage and the routine simply returns. When the data arrives, the input IRQ service routine sends the signal and releases the send-signal condition.

**NOTE:** Setting up a "send signal on data ready" condition will "busy" the driver for read requests (see READ description), but allow writes to proceed as normal.

**NOTE:** Only interrupt-driven drivers should implement this call.

# TERM — Terminate Device

***INPUT:*** **(a1) = device descriptor pointer**
**(a2) = pointer to device static storage**
**(a4) = process descriptor pointer**
**(a6) = system global data pointer**

***OUTPUT:*** **None**

***ERROR OUTPUT:*** **cc = carry bit set**
**d1.w = error code**

***FUNCTION:*** This routine is called when a device is no longer in use in the system (see I$Detach).

The TERM routine must:

- Copy the process ID from the process descriptor (P$ID) into V_BUSY and V_LPRC.

- Determine if the output FIFO buffer contains any data waiting to be written. If so, the driver should suspend itself by copying its process ID from V_BUSY to V_WAKE and performing an F$Sleep service request to put itself to sleep indefinitely.

  If the driver awakens before the output FIFO has emptied (due to a signal), the driver should suspend itself again until the buffer is empty.

- After the pending output data has been written, the driver should disable hardware handshake protocols and then disable all device interrupts, if the driver is interrupt-driven. The device should then be removed from the system's IRQ polling table (F$IRQ), if applicable.

- Return any buffers the driver has requested on behalf of itself. **NOTE:** The driver should not attempt to return buffers within its defined static storage area. The kernel releases this memory when the TERM routine completes.

**NOTE:** If an error occurs during the device's INIT routine, the kernel calls the TERM routine to allow the driver to clean up. If the TERM routine uses static storage variables (for example, interrupt mask values, dynamic buffer pointers), it should validate these variables prior to using them. The INIT routine may not have set up all the variables prior to exiting with the error.

# IRQ Service Routine — Service Device Interrupts

***INPUT:*** **(a2) = static storage**
**(a3) = port address**
**(a6) = system global static storage**

***OUTPUT:*** **None**

***ERROR OUTPUT:*** **cc = carry bit set (interrupt not serviced)**

***FUNCTION:*** This routine is called directly by the kernel's IRQ polling table routines. Its function is to:

- Check the device for a valid interrupt. If the device does not have an interrupt pending, the carry bit must be set and the routine exited with an RTS instruction as quickly as possible. Setting the carry bit signals the kernel that the next device on the vector should have its IRQ service routine called.

- Service device interrupts. There are three categories of interrupts: control interrupts, input interrupts, and output interrupts. Usually, input interrupts are checked first, because most serial hardware devices have minimal (or no) hardware data buffering. After the interrupt is serviced, many drivers check for another pending interrupt prior to exiting to the kernel. This technique (for example, service input interrupt, service pending output interrupt, service next input interrupt) provides efficient interrupt servicing because it allows the driver to service multiple interrupts with one call to the IRQ service routine.

- Clear the carry bit and exit with a RTS instruction after servicing an interrupt.

Avoid exception conditions (for example, a Bus Error) when IRQ service routines are executing. Under the current version of the kernel, an exception in an IRQ service routine will crash the system.

**NOTE:** IRQ service routines may destroy the contents of the following registers only: d0, d1, a0, a2, a3, and a6. You must preserve the contents of all other registers or unpredictable system errors (system crashes) will occur.

The interrupt categories (control, input, and output) are described in the following pages.

### Control Interrupts

These interrupts are usually associated with non-data type information on the serial port, such as the receipt of a break character or a change in the Data Carrier line. Control interrupts may also signal error conditions on the data stream (for example, parity error).

When signaling is set up for Data Carrier transactions (see SetStat, SS_DCOn, SS_DCOff), the routine should send the specified signal to the specified process, clear down the signal condition, mark the path as "lost" (V_HangUp set to non-zero), and then exit (carry bit clear) or service more interrupts.

### Input Interrupts

The input interrupt routine typically performs the following:

¨ Read the character from the hardware, clear down the interrupt, and strip parity (if required).

¦ Check character error status. If in error, update V_ERR to indicate the error.

Æ If the character is not a NULL character, determine whether or not the character requires special handling.

a) If the character is the output pause character (V_PCHR), set a pause request (V_PAUS) in the echo device's static storage (V_DEV2).

b) If the character is a keyboard interrupt (V_INTR) or quit character (V_QUIT), send the appropriate signal to the last process to use the device (V_LPRC).

c) If the character is a software handshake character (V_XON or V_XOFF), service the handshake request. For an "output resume" case (V_XON), this typically involves clearing the "output halted due to X-OFF" flag, checking for data in the output FIFO, and enabling output interrupts, if so. For an "output halt" case (V_XOFF), this typically involves setting the "output halted due to X-OFF" flag and disabling output interrupts on the hardware.

**NOTE:** The software handshake characters are consumed by this routine. After processing these characters, the IRQ service routine exits to the kernel (carry bit clear) or services the next pending device interrupt.

Ø Put the character into the input FIFO buffer. If there is no room in the buffer, the character is lost and the driver should indicate "input buffer overrun" in the accumulated error status (V_ERR). In this case, the driver often returns to the kernel at this point, after waking the driver process (V_WAKE).

× Determine if any process has set up a "send signal on data ready" condition (SS_SSig). If so, signal the process, clear down the signaling condition, and exit (carry bit clear) or service the next pending interrupt.

± Examine the number of characters in the input FIFO, if the driver supports handshaking.

For software handshaking, if the buffer is nearly full (reached the "high-water mark"), the driver should send a suspend transmission character (V_XOFF) to the distant end and flag that input has been halted. This function allows the driver to prevent input FIFO overrun errors when the data is being received at a faster rate than it is being read from the FIFO. Typically, the READ routine re-enables input data flow when it has emptied the input FIFO to a suitable low value ("low-water mark") by causing the V_XON character to be sent.

For hardware handshaking, the input interrupt routine should signal its desire to suspend input by negating its "ready to receive" line.

ð If desired, the input IRQ service routine can now service more interrupts. Once fully completed, it should exit to the kernel with the carry bit clear. Prior to exiting, it should send a wake-up signal (S$Wake) to any waiting driver process. You can find the process ID in V_WAKE, which you should clear.

### Output Interrupts

The output interrupt routine typically performs the following:

¨ Determine if V_XON or V_XOFF is pending, due to input buffer software handshaking. If so, send the required character, flag it sent, and mark the current state of input (halted or resumed). The driver should then determine if output is currently halted (buffer empty or software handshake). If so, it should disable output interrupts and return to the kernel (carry bit clear). If not, further interrupts may be processed or an exit may be made to the kernel (carry bit clear).

¦ Determine if output is halted due to software handshaking. If so, disable output device interrupts and return to the kernel (carry bit clear).

Æ Determine if any data is waiting in the output FIFO for transmission. If so, write the data to the hardware.

Ø Determine the remaining data count in the output FIFO.

a) If zero, flag the buffer empty, disable output device interrupts, wake any waiting process (V_WAKE) and exit to the kernel (carry bit clear).

b) If not zero, check if current count is below the output buffer's "low-water mark". If not, exit to the kernel (carry bit clear) *without* waking the driver process. If so, wake the driver process before exiting.

This technique minimizes contention between the driver's WRITE routine (filling the output buffer) and the output IRQ service routine (emptying the output buffer), as the buffer is allowed to empty significantly before the WRITE process is re-activated.

***End of Chapter 3***