

# **Sequential Block File Manager (SBF)**

## **SBF General Description**

The Sequential Block File Manager (SBF) is a re-entrant subroutine package for I/O service requests to sequential block-oriented mass storage devices, such as tape systems. SBF can handle any number or type of such systems simultaneously.

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

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 SBF:

I\$ChgDir	I\$Delete	I\$MakDir	I\$Seek
-----------	-----------	-----------	---------

When one of these service requests is made to SBF, an appropriate error code is returned.

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

I\$Attach	I\$Detach	I\$Dup
-----------	-----------	--------

SBF is designed to support both buffered and unbuffered I/O. It is capable of handling variable logical block sizes. SBF has no knowledge of the media's physical block size, and the driver is responsible for translating the logical block requests by SBF into the media's physical block requests. The logical block size for an SBF device is defined in the PD\_Blksiz field of the path descriptor.

## **Unbuffered I/O**

Unbuffered I/O is used when the `PD_NumBlk` field of the path descriptor is set to 0.

When operating in unbuffered mode, SBF uses a single buffer for `I$ReadLn` and `I$WriteLn` calls. `I$Read` and `I$Write` calls do not use an intermediate buffer, and the data is transferred directly between the caller's data buffer and the driver.

Unbuffered I/O operates synchronously with the requesting process. The process makes a read or write request and SBF returns to the caller when the I/O operation has completed.

## **Buffered I/O**

Buffered I/O is used when the `PD_NumBlk` field of the path descriptor is set to a positive number. All buffered I/O is initiated asynchronously by an auxiliary process created by SBF. SBF uses a "pool" of buffers to accomplish this. The maximum number of buffers to use is specified by the `PD_NumBlk` field of the path descriptor. The size of each buffer is specified by the `PD_BlkSiz` field of the path descriptor.

`I$Read` requests cause SBF to copy data from the buffer pool. If a full buffer is not yet available, SBF allocates a new buffer and passes it to the auxiliary process. SBF then waits for the auxiliary process to return the buffer containing the next block. Multiple buffers (up to the number specified by `PD_NumBlk`) may be allocated, thus allowing SBF to copy data from one buffer while the auxiliary process reads data into others.

`I$Write` requests cause SBF to copy data into a buffer and return to the user immediately. When a buffer fills, SBF passes it to the auxiliary process for writing. If another buffer is required before the auxiliary process has had time to write the previous buffer, SBF allocates a new buffer and copies data to it. This allows SBF to copy data into one buffer while the auxiliary process writes from others.

## **Considerations When Writing to Tapes**

When an SBF path is opened, any I/O operations may be done on the path. However, after an `I$Write` call is made, SBF flags the path as "in write mode" and will not allow any `I$Read` calls until an `I$SetStt` call is made. Typically, when writing a tape, an `I$Close` call follows an `I$Write` call and SBF performs its normal close processing. When an `I$SetStt` call follows an `I$Write` call, SBF waits for any pending writes to complete, clears the write mode flag, and performs the `I$SetStt`. It is recommended that `I$SetStt` writes one or more filemarks, to ensure that a filemark follows the data written.

## End-Of-Tape Processing

There is no “end-of-tape” error on Read requests. Consequently, SBF requires an end-of-file mark to be present or the user process to handle the situation (to know the size of the file or use an end-of-data record).

`I$Write` requests return a media full error (`E$Full`) when end-of-tape is reached. All prior writes will have completed; no other data may be written other than filemarks after the end-of-tape has been reached.

## SBF I/O Service Requests

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

`I$Close` SBF performs the following functions:

- If the use count for the path is non-zero (other processes are still using this path), SBF does not return an error.
- If the use count is zero, SBF determines if the path is in write mode. If so, SBF calls the device driver to write two filemarks to the tape.
- If the path is in write mode and the `f_eras_b` flag is set in the `PD_Flags` field of the path descriptor, SBF calls the device driver to erase to the end of the tape.
- If the `f_rest_b` flag is set in `PD_Flags`, SBF calls the device driver to rewind the tape. If the path is in write mode and `f_rest_b` is not set, SBF calls the device driver to skip back one filemark. This positions the tape between the two filemarks just written.
- If the `f_offl_b` flag is set in `PD_Flags`, SBF calls the device driver to take the tape drive off-line.
- Any buffers associated with the path are returned to the system.

`I$Create` SBF considers `I$Create` to be synonymous with `I$Open`.

`I$GetStt` Refer to the `I$GetStt` description in the **OS-9 Technical Manual** for a detailed explanation of the SBF-supported `I$GetStt` functions:

<code>SS_Ready</code>	Test for data ready.
<code>SS_EOF</code>	Check for end of file condition.

All other GetStat calls are passed to the driver.

---

I\$Open	<p>SBF performs the following functions:</p> <ul style="list-style-type: none"><li>• Validates the pathname.</li><li>• Verifies that the drive number (PD_TDrv) is legal for the device driver (SBF_NDRV).</li><li>• Initializes path descriptor variables.</li><li>• Creates the auxiliary process for the driver (SBF_DPrc), if required.</li></ul>
I\$Read	<p>SBF calls the driver as needed to read the data. Complete blocks of data are transferred directly to the user's buffer while incomplete blocks are transferred into SBF's buffer. The portion of the data requested by the calling process is copied into the calling process' buffer. If buffers are required for the read (for example, buffered I/O mode), these are allocated as required.</p>
I\$ReadLn	<p>I\$ReadLn is similar to I\$Read, except that SBF stops the read if an end-of-record character (carriage return) is found. I\$ReadLn requests always transfer the data through an intermediate SBF buffer.</p>
I\$SetStt	<p>Refer to the I\$SetStt description in the <b>OS-9 Technical Manual</b> for a detailed explanation of the SBF supported I\$SetStt functions:</p> <p style="padding-left: 40px;">SS_Opt            Write the path descriptor options.</p> <p>All other SetStat calls are passed to the driver. If the block size (PD_Blksiz) has changed, SBF ensures that all current buffers are flushed prior to calling the device driver. <b>NOTE:</b> Only SS_Opt is passed to the driver after processing by SBF. If an unknown service request error (E\$UnkSvc) is returned by the driver, it is ignored.</p>
I\$Write	<p>SBF calls the driver as needed to transfer the data as follows:</p> <p><b>Buffered I/O</b></p> <p>SBF copies the user's data into the next free buffer in the SBF buffer pool. The user process is reactivated immediately. As each buffer fills (PD_Blksiz), SBF calls the driver to write the data when the driver is available.</p> <p><b>Unbuffered I/O</b></p> <p>SBF calls the driver with the data pointer pointing to the user's data buffer. The driver writes the data to tape; the user process is reactivated when the driver completes the write operation.</p>
I\$WritLn	<p>I\$WritLn is similar to I\$Write, except that SBF only writes data up to and including the first end-of-record character (carriage return), if there is one in the calling process's buffer. If no end-of-record character is found, SBF writes the amount of data specified by the calling process. I\$WritLn requests always transfer the data through an intermediate SBF buffer.</p>

## SBF Device Descriptor Modules

This section describes the definitions of the initialization table contained in device descriptor modules for SBF devices. The initialization table immediately follows the standard device descriptor module header fields. 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_TDrv	Tape Drive Number
\$4A	PD_SBF	Reserved
\$4B	PD_NumBlk	Maximum Number of Blocks to Allocate
\$4C	PD_BlkJz	Logical Block Size
\$50	PD_Prior	Driver Process Priority
\$52	PD_SBFflags	SBF Path Flags
\$53	PD_DrivFlag	Driver Flags
\$54	PD_DMAMode	Direct Memory Access Mode
\$56	PD_ScsiID	SCSI Controller ID
\$57	PD_ScsiLUN	LUN on SCSI Controller
\$58	PD_ScsiOpts	SCSI Options Flags

**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, the following adjustment must be made: (M\$DType - PD\_OPT).

For example, to access the tape drive number in a device descriptor, use the following value: PD\_TDrv + (M\$DType - PD\_OPT). To access the tape drive number in the path descriptor, use PD\_TDrv. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l or usr.l.

Name	Description
PD_DTP	<p><b>Device class</b> This field is set to three for SBF devices.</p>
PD_TDrv	<p><b>Tape Drive number</b> Used to associate a one-byte integer with each drive that a controller will handle. If using dedicated (for example, non-SCSI bus) controllers, this field usually defines both the <i>logical</i> and <i>physical</i> drive number of the tape drive. If using tape drives connected to SCSI controllers, this number defines the <i>logical</i> number of the tape drive to the device driver. The <i>physical</i> controller ID and LUN are specified by the PD_ScsiID and PD_ScsiLUN fields. Each controller's drives should be numbered 0 to n-1 (n is the maximum number of drives the controller can handle). This number also defines how many drive tables are required by the driver and SBF. SBF verifies this number against SBF_NDRV prior to calling the driver.</p>
PD_NumBlk	<p><b>Number of Buffers/Blocks Used For Buffering</b> Specifies the maximum number of buffers to be allocated by SBF for use by the auxiliary process in buffered I/O. If this field is set to 0, unbuffered I/O is specified.</p>
PD_BlkJiz	<p><b>Logical Block Size Used For I/O</b> Specifies the size of the buffer to be allocated by SBF. This buffer size is used when allocating multiple buffers used in buffered I/O. Unless the driver manages partial physical blocks, this size should be an integer multiple of the physical tape block size.</p>
PD_Prior	<p><b>Driver Process Priority</b> The priority at which SBF's auxiliary process will run. This value is used during initialization. Changing this value after initialization has no effect.</p>
PD_SBFflags	<p><b>SBF Path Flags</b> Specifies the actions that SBF takes when the path is closed. A user can update this field using GetStat/SetStat (SS_Opt). SBF supports the following flag definitions:</p> <ul style="list-style-type: none"> <li>bit 0: (f_rest_b) 0 = No rewind on close. 1 = Rewind on close.</li> <li>bit 1: (f_offl_b) 0 = Do not put drive off-line on close. 1 = Put drive off-line on close.</li> <li>bit 2: (f_eras_b) 0 = Do not erase to end-of-tape on close. 1 = Erase to end-of-tape on close.</li> </ul>

---

Name	Description
PD_DrivFlag	<b>Driver Flags</b> This field is available for use by the device driver.  <b>NOTE:</b> References to these flags are often made using the PD_Flags offset (defined in sys.l and usr.l). This reference is equivalent to PD_SBFflags. References to PD_DrivFlag should use a value of PD_Flags + 1.
PD_DMAMode	<b>Direct Memory Access Mode</b> This field is hardware specific. If available, you can use this word to specify the DMA Mode of the driver.
PD_ScsiID	<b>SCSI Controller ID</b> This is the ID number of the SCSI controller attached to the device. The driver uses this number when communicating with the controller.
PD_ScsiLUN	<b>Logical Unit Number of SCSI Device</b> This number is the value to use in the SCSI command block to identify the logical unit on the SCSI controller. This number may be different from PD_TDrv, to eliminate allocation of unused drive table storage. PD_TDrv indicates the logical number of the drive to the driver and SBF (drive table to use). PD_ScsiLUN is the physical drive number on the controller.
PD_ScsiOpts	<b>SCSI Driver Options Flags</b> This field allows SCSI device options and operation modes to be specified. It is the driver's responsibility to use or reject these if applicable:  bit 0: 0 = ATN not asserted (no disconnects allowed). 1 = ATN asserted (disconnects allowed).  bit 1: 0 = Device cannot operate as a target. 1 = Device can operate as a target.  bit 2: 0 = asynchronous data transfers. 1 = synchronous data transfers.  bit 3: 0 = parity off. 1 = parity on.  All other bits are reserved.

## SBF Path Descriptor Definitions

The reserved section (PD\_OPT) of the path descriptor used by SBF is copied directly from the initialization table of the device descriptor. The following table is provided to show the offsets used in the path descriptor. For a full explanation of the path descriptor fields, refer to the previous pages.

Offset	Name	Description
\$80	PD_DTP	Device Type
\$81	PD_TDrv	Tape Drive Number
\$82	PD_SBF	Reserved
\$83	PD_NumBlk	Maximum Number of Blocks to Allocate
\$84	PD_BlkJsz	Logical Block Size
\$88	PD_Prior	Driver Process Priority
\$8A	PD_SBFFlags*	SBF Path Flags
\$8B	PD_DrivFlag*	Driver Flags
\$8C	PD_DMAMode	Direct Memory Access Mode
\$8E	PD_ScsiID	SCSI Controller ID
\$8F	PD_ScsiLUN	LUN on SCSI controller
\$90	PD_ScsiOpts	SCSI Options Flags

\* References to these flags are often made using the PD\_Flags offset (defined in sys.l and usr.l). This reference is equivalent to PD\_SBFFlags. References to PD\_DrivFlag should use a value of PD\_Flags + 1.

**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.

## **SBF Device Drivers**

SBF device drivers are designed to support any sequential storage device which reads and writes data in fixed or variable size blocks (tapes).

Because SBF is intended for sequentially accessed files, it does not support a directory structure or provide a byte-oriented file positioning mechanism. Consequently, `I$Makdir`, `I$ChgDir`, `I$Delete`, and `I$Seek` return the error `E$UnkSvc`.

Read and write calls to the driver are made by SBF in terms of logical blocks. The logical block size is specified in the `PD_BlkJsz` field of the path descriptor. The driver is responsible for translating the block request into the appropriate number of physical media blocks. If a "partial" physical block results from this translation, drivers must either buffer the partial block or return an error.

`GetStat` calls are passed straight to the driver, with the exception of `SS_EOF` and `SS_Ready`, which are handled by SBF. Typical drivers ignore all `GetStat` calls and return an unknown service request error (`E$UnkSvc`).

`SetStat` calls are passed straight to the driver, with the exception of `SS_Opt`. SBF determines if the buffer size has changed, and if so, flushes any pending buffers to tape prior to calling the driver. `SetStat` calls to the driver are used for control and positioning operations (for example, write filemark, rewind tape) on the media. These calls can originate from the user or from SBF internal operations (for example, write filemark when file closed).

### ***Sensing the End-of-Tape***

All tape drives can sense the physical end-of-tape (EOT). Many drives also provide an "early" EOT warning. The type of warning(s) provided by the drive determines whether or not buffered I/O (`PD_NumBlk`) is usable, as follows:

#### **Early EOT Warning**

Drives which provide an early EOT capability notify the driver of the EOT condition prior to reaching the end of the physical tape. The amount of tape between the early EOT mark and physical tape end varies among drive models; however, typical drives allow about 1000 physical blocks to be written after the early EOT warning.

When a driver that is writing blocks encounters the early EOT warning, it should write the blocks to the tape and return a media full error (`E$Full`). If the device is using buffered I/O, subsequent write calls may still be made by SBF to the driver to flush all currently buffered blocks to the tape. The driver should not refuse these write requests: it should continue to write the data to tape and continue returning `E$Full`.

The driver should maintain this mode of operation until a “control” operation occurs (for example, write filemark or rewind), at which time the driver can clear its EOT status. This technique of writing all currently buffered blocks to tape ensures that the application knows which blocks are on which tape.

When setting up the device descriptors block size (`PD_BlkSiz`) and buffer count (`PD_NumBlk`), you should ensure that there is enough room on the tape after the early EOT mark to accommodate the total amount of data that could be buffered (`PD_NumBlk * PD_BlkSiz`).

Drives which provide early EOT warning can operate in buffered or unbuffered I/O mode.

### **Physical EOT Warning**

Drives which only provide a physical EOT warning notify the driver when the actual end-of-tape is about to be reached. There is sufficient tape remaining to allow the last write to complete and a filemark to be written. No additional blocks can be written to the tape.

You can only operate physical EOT devices in unbuffered I/O mode, because there is no guarantee that you can write SBF-buffered blocks to tape after the physical EOT is detected. When the driver detects EOT, it should ensure that the last write has completed and return a media full error (`E$Full`). The next access to the driver is typically a write filemark operation and rewind.

## ***Tape Positioning Operations***

SetStat functions are available to allow tape positioning operations. These calls allow the driver to skip forward or backward on the tape, using a specified block or filemark count.

Depending upon the capabilities of the tape drive in use, reverse tape movement may require driver assistance. If the tape drive supports reverse movement, the driver simply hands the count to the drive. If the tape drive only supports forward movement, the driver has to maintain counters for the current filemark and block position on the tape. The driver must use movement commands supported by the tape drive to simulate reverse movement. For example, if the tape's current position is filemark #2, block #20, then a request to move back five blocks would (typically) be simulated by:

- .. Rewind tape
- | Skip forward two filemarks
- Æ Skip forward 15 blocks

When this situation is in effect, drivers maintain these tape position counters in an external module (for example, data module), so that the counters are not erased when the device is attached and detached. The INIT routine attempts to create and link to the module, while the TERM routine unlinks the module.

Some tape motion commands (for example, rewind, skip blocks, retension) may take a long time. When using SCSI tape drives, these types of functions can busy the SCSI bus to other users for excessive lengths of time. To improve this situation, drivers should follow these guidelines:

- If possible, set the “immediate return” flag in the SCSI command packet, to enable the tape drive to return status without waiting for motion to complete.
- If possible, implement disconnect/reconnect, to enable the tape drive to release the bus during long motion functions, allowing other SCSI activity (such as disk accesses) to occur.

## **Tape Streaming**

Tape “streaming” is achieved when the process and driver are able to send/receive data to/from the tape device at a rate that is equal to or faster than the tape drive’s data I/O rate. The tape drive can keep the tape in motion continuously, thus achieving the minimum data transfer time. If the data rate falls below this threshold, the tape drive has to perform stop-motion/reverse/start-motion functions whenever it has to wait for the process/driver to issue the next I/O request. This stop/start motion can significantly increase the time it takes for the overall tape operations.

To achieve maximum streaming on tapes, drivers should follow these guidelines:

- Use buffered I/O (PD\_NumBlk) on tape drives that support early EOT detection.
- Set the logical block size (PD\_BlkJsz) to the size of the tape drive’s internal buffer (typical tape drives have an internal buffer to assist streaming).
- If the tape drive supports “immediate returns” on writes, turn this function on. Immediate returns allow the tape drive’s controller to indicate “command complete” to the driver when the data is in the controller’s internal buffer, but prior to writing the data to physical tape. The controller then begins writing to tape while SBF is preparing for the next write.
- On SCSI-based systems, implement disconnect/reconnect if possible, so that tape operations minimize SCSI bus occupancy. This allows situations such as SCSI-disk to SCSI-tape backups to achieve maximum overlaps of disk/tape activity.

## SBF Device Driver Storage Definitions

SBF device driver modules contain a package of subroutines that perform block-oriented I/O 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 SBF; 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 `sbfdev.a` and `sbfdrvtb.a` DEFS files. Static storage is used as follows:

Offset	Name	Maintained By	Description
\$00	V_PORT	Kernel	Device base address
\$04	V_LPRC	Kernel	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
\$30	SBF_NDRV	Driver	Number of Drives
\$32	SBF_Flag	File Manager	Driver Flags
\$34	SBF_Drvr	File Manager	Driver Module Pointer
\$38	SBF_DPrc	File Manager	Driver Process Pointer
\$3C	SBF_IPrc	Driver	Interrupt Process Pointer
\$40			Reserved
\$80			Drive Tables Begin

**NOTE:** *Offset* refers to the location of a static storage field relative to the starting address of the static storage. 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	<b>Device port address</b> Contains 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	<b>Last active process ID</b> Contains the process ID of the last process to use the device. While this field is required for all static storage by the kernel, it is not used by SBF.

Name	Description
V_BUSY	<p><b>Current active process</b></p> <p>The process ID of the process currently using the device. It is used to implement I/O Blocking by SBF. 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.</p>
V_WAKE	<p><b>Process ID to awaken</b></p> <p>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. The driver sets V_WAKE from V_BUSY. V_WAKE provides the interlock between the driver and the driver's interrupt service routine.</p>
V_PATHS	<p><b>Linked List of Open Paths</b></p> <p>A singly-linked list of all paths currently open on this device.</p>
SBF_NDRV	<p><b>Number of drives</b></p> <p>Contains the number of drives that the controller can use. It is defined by the device driver as the maximum number of logical drives with which the controller can work. SBF assumes that there is a drive table for each drive. SBF validates the tape drive number (PD_TDrv) against this value to ensure that the logical drive number is valid for the driver.</p>
SBF_Flag	<p><b>Driver Flags</b></p> <p>Contains flags used by SBF to indicate the current state of the path.</p>
SBF_Drvr	<p><b>Driver Module Pointer</b></p> <p>Contains the pointer to the device driver.</p>
SBF_DPrc	<p><b>Driver Process Pointer</b></p> <p>Contains the pointer to the process associated with the driver. SBF initializes this when a path is opened to the device. The driver's TERM routine should check this field, and if non-zero, delete the process (F\$DelPrc).</p>
	<p><b>SBF_IPrcInterrupt Process Pointer (obsolete)</b></p> <p>This field is available for the driver to use when the driver wishes to create its own process (for example, interrupt handler process). <b>NOTE:</b> Do not confuse this process with the SBF process created for buffered I/O. (See SBF_DPrc.)</p>
	<p><b>Drive Tables</b></p> <p>Contains one table per drive that the controller will handle. SBF assumes there are as many tables as specified in SBF_NDRV.</p>

## Device Driver Tables

There must be as many drive tables as were specified in SBF\_NDRV. The format of each drive table is given below:

Offset	Name	Maintained By	Description
\$00	SBF_DFlg	File Manager	Drive Flag
\$02	SBF_NBuf	File Manager	Buffer Count
\$04	SBF_IBH	File Manager	Pointer to Head of Input Buffer List
\$08	SBF_IBT	File Manager	Pointer to Tail of Input Buffer List
\$0C	SBF_OBH	File Manager	Pointer to Head of Output Buffer List
\$10	SBF_OBT	File Manager	Pointer to Tail of Output Buffer List
\$14	SBF_Wait	File Manager	Pointer to Waiting Process
\$18	SBF_SErr	Driver	Number of Recoverable Errors
\$1C	SBF_HErr	Driver	Number of Non-Recoverable Errors
\$20			Reserved

Name	Description
SBF_DFlg	<p><b>Drive Flag</b></p> <p>The high byte of this field contains the current status of the logical drive. The flags are maintained by SBF, and are defined as follows:</p> <ul style="list-style-type: none"> <li>bit 1: Set if write mode.</li> <li>bit 2: Set if driver servicing this drive.</li> <li>bit 3: Set if EOF (end of file).</li> </ul> <p>All other bits and the low byte bits are reserved.</p>
SBF_NBuf	<p><b>Buffer Count</b></p> <p>Contains the number of buffers currently allocated to the drive.</p>
SBF_IBH	<b>Pointer to Head of Input Buffer List</b>
SBF_IBT	<b>Pointer to Tail of Input Buffer List</b>
	These fields contain the head and tail pointers, respectively, of the buffers being returned to SBF by the driver.
SBF_OBH	<b>Pointer to Head of Output Buffer List</b>
SBF_OBT	<b>Pointer to Tail of Output Buffer List</b>
	These fields contain the head and tail pointers, respectively, of the buffers being sent to the driver by SBF.

Name	Description
SBF_Wait	<b>User process' process descriptor pointer</b> This pointer is set when the user process is suspended, waiting for driver I/O to complete.
SBF_SErr	<b>Number of Recoverable Errors</b> This field allows the driver to keep a count of “soft” errors during I/O operations. The value would typically be returned by a SS_ELog GetStat call. After reading this value, it is typically reset to zero.
SBF_HErr	<b>Number of Non-Recoverable Errors</b> This field allows the driver to keep a count of “hard” errors during I/O operations. The value would typically be returned by a SS_ELog GetStat call. After reading this value, it is typically reset to zero.

## Linking SBF Drivers

After a SBF 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 three sections in this order (see Figure 4-1):

- I/O globals
- | Drive tables (one per logical drive)
- Æ Driver-declared variables

The driver-declared variables are declared in `vsect` areas of the driver, but they *must* be allocated after the drive table storage areas. The method that must be used to allocate all of the storage, *in the correct order*, is to link the `sbfstat.r` library file, 'n' instances of `sbfdrvtb.r`, and then the driver `vsect`. The `sbfstat.r` and `sbfdrvtb.r` files are located in the system's LIB directory.

The following examples show how a driver should be linked. The first link line creates a driver that supports one logical drive, as only one drive table `vsect` is allocated:

```
l68 /dd/LIB/sbfstat.r /dd/LIB/sbfdrvtb.r RELS/sbviper.r -O=OBJS/sbviper
```

The second link line creates a driver that supports two logical drives, as two drive table `vsects` are allocated:

```
l68 /dd/LIB/sbfstat.r /dd/LIB/sbfdrvtb.r /dd/LIB/sbfdrvtb.r RELS/sbtape.r  
-O=OBJS/sbtape
```

**NOTE:** Failure to link the I/O system globals and the correct number of drive tables, and in the correct order, results in erratic driver operation.

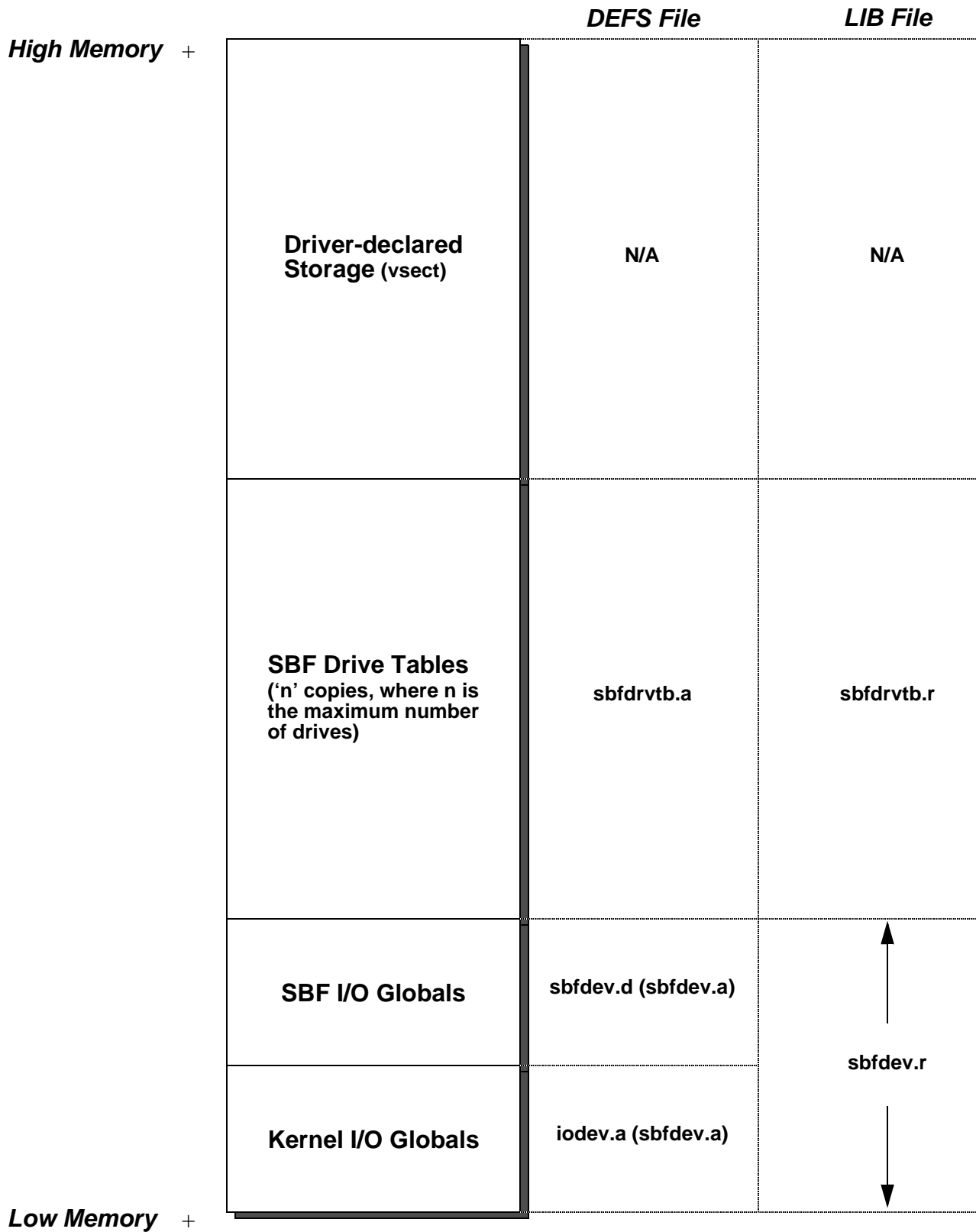


Figure 4-1: SBF Static Storage Layout

## SBF Device Driver Subroutines

As with all device drivers, SBF device drivers use a standard executable memory module format with a module type of `Drivr` (code `$E0`). SBF 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:

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

Each subroutine should exit with the carry bit of the condition code register cleared, if no error occurred. Otherwise, the carry bit should be set and an appropriate error code returned 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 the device descriptor module  
 (a2) = address of device static storage  
 (a4) = process descriptor pointer  
 (a6) = system global data pointer

**OUTPUT:** None

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

**FUNCTION:** The INIT routine must:

- Initialize the device's permanent storage. Minimally, this consists of initializing SBF\_NDRV to the number of drives with which the controller will work.

If the driver maintains flags/variables that must "span" detach/attach sequences (for example, for reverse movement simulation), then the INIT routine should create/link to an external module (for example, a data module). The module pointer should then be saved. If the module was created, its storage area should then be initialized.

- Place the IRQ service routine on the IRQ polling list by using the F\$IRQ service request, if required.

Æ Initialize device control registers (enable interrupts if necessary).

Prior to being called, the device permanent storage is cleared (set to zero) except for V\_PORT which will contain the device address.

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, handle the interrupt in one of two ways:

- Process the interrupt directly by masking interrupts to the level of the device, polling/servicing the device hardware, 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. You cannot use **V\_BUSY** because it is zero when **INIT** is called.

**READ****Read Block(s)**

**INPUT:** d0.l = buffer size  
(a0) = address of buffer  
(a2) = address of device static storage  
(a3) = drive table  
(a4) = process descriptor pointer  
(a6) = system global data storage pointer

**OUTPUT:** d1.l = block size read

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

**FUNCTION:** The READ routine must:

- Initialize the drive, if required.
- | Convert the requested byte-count into the block-count for the media. If the requested count does not specify an integral number of media blocks, the driver should return an error (typical case) or take steps to buffer the partial block.
- Æ Issue the READ command to the device and wait for I/O to complete (using interrupts if possible).
- Ø When the I/O operation is complete, check the status of the READ. If a fatal error occurred, return it to SBF.
- × If no error, or a non-fatal error occurred, check the amount of data actually read and return that count to SBF.

Most tape devices terminate a READ request when a filemark is encountered. The tape device returns the data from the current position up to the filemark. Thus, the byte-count returned may be less than the requested amount. This is a typical non-fatal error on tape devices.

**WRITE****Write Block(s)**

**INPUT:**     **d0.l** = buffer size  
              **(a0)** = address of buffer  
              **(a2)** = address of the device static storage area  
              **(a3)** = drive table  
              **(a4)** = process descriptor pointer  
              **(a6)** = system global data storage pointer

**OUTPUT:**    The buffer is written to tape.

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

**FUNCTION:**  The WRITE routine must:

- Initialize the drive, if required.
- | Convert the requested byte-count into the block-count for the media. If the requested count does not specify an integral number of media blocks, then the driver should return an error (typical case) or take steps to buffer the partial block.
- Æ Issue the WRITE command to the device and wait for I/O to complete (using interrupts if possible).
- Ø When the I/O operation has completed, check the status of the WRITE. If a fatal error occurred, return it to SBF.
- × If no error, or a non-fatal error occurred, check the amount of data actually written.

Many tape devices terminate a write request when an early end-of-tape (EOT) is detected. For these types of devices, the data can still be written to tape because the EOT state is a warning that there is a small amount of tape remaining. The driver should ensure that this write is fully completed, and return a media full error (E\$Full).

Subsequent write calls should not be refused at this point, as SBF may need to flush its current buffers (if in buffered I/O mode) to the tape. The application is notified of the media full condition on its next write, so that it may close the file. When the file closes, SBF issues appropriate **SetStats** (for example, write filemark) to finalize tape operation.

If the tape device is one which only detects a physical EOT condition, then the driver should only be operated in unbuffered I/O mode. In this case, the driver should ensure that the write invoking the physical EOT condition is written to tape and a media full error (E\$Full) returned to SBF. No further writes should be presented to the driver, as the application is notified immediately of the media full condition. The application can then close the path, allowing SBF to write the final filemark and finalize tape operation.

**GETSTAT/SETSTAT****Get/Set Device Status**

**INPUT:**     **d0.w** = status code  
               **d2.l** = argument count  
               **(a1)** = address of the path descriptor  
               **(a2)** = address of the device static storage area  
               **(a3)** = drive table  
               **(a4)** = process descriptor pointer  
               **(a6)** = system global data storage pointer

**OUTPUT:**    Depends on the function code

**ERROR**     **cc** = carry bit set  
**OUTPUT:**   **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. The register stack image pointer is stored in the path descriptor (PD\_RGS).

Typical SBF drivers have routines to handle the following I\$SetStt codes:

SS_Feed	Erase tape
SS_Opt	Write path options section
SS_Reset	Rewind tape
SS_Reten	Retension tape
SS_RFM	Skip past tape mark(s)
SS_Skip	Skip block(s)
SS_SQD	Place drive off-line
SS_WFM	Write tape mark(s)

Usually all I\$GetStt codes and other I\$SetStt codes return with an unknown service request error (E\$UnkSvc).

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

**SS\_Feed** This call erases all or part of the tape. The number of blocks to be erased is passed in register d2. If the count is -1, the entire tape is to be erased from the current position to end-of-tape (EOT), otherwise, the specified count of blocks should be written, starting at the current tape position.

The erase routine should:

- “ Initialize the drive, if required.
- | Issue the appropriate command to achieve the desired erase function. Many tape devices support a direct “erase” command. If the tape device does not support this feature, the driver should perform “writes” to simulate the desired effect. Once the command is issued, the driver should wait for I/O to complete (with interrupts if possible).
- Æ Check the status of the I/O command and return any error to SBF.
- Ø If the driver maintains flags pertaining to current tape position, these should be updated.
- × Return status to SBF.

**SS\_Opt** This routine is called when the path descriptor options are changed by the user. Typically, the driver ignores this call.

**SS\_Reset** This call rewinds the tape to beginning-of-tape (BOT). The rewind routine should:

- “ Initialize the drive, if required.
- | Issue the appropriate command to the device and wait for I/O to complete (with interrupts, if possible).
- Æ Check the status of the I/O command and return any error to SBF.
- Ø If the driver maintains internal flags pertaining to current tape position, they should be reset. Typical flags would be end-of-file and end-of-tape. For drivers that count current filemark/block positions, these counters should also be cleared.
- × Return status to SBF.

**SS\_Reten** This call performs a retension pass on the tape. Typically, the tape moves to BOT, moves to EOT, then rewinds to BOT. The sequence of actions for **SS\_Reten** is the same as that for **SS\_Reset**.

Retensioning tape media is highly recommended for new media, shipped media, or any media that has been stored for a long period.

**SS\_RFM** This routine is called when the tape position is to be moved forward or backwards by the specified number of filemarks. (This number is passed in register **d2**.) If the tape device is incapable of directly skipping backward, the driver has to simulate the reverse movement using rewind and skip forward commands. The sequence of actions for **SS\_RFM** is the same as that for **SS\_SQD**.

**SS\_Skip** This routine is called when the tape position is to be moved forward or backward the specified number of tape blocks. The number of blocks to skip is passed as a logical block count (**PD\_BlksSz**) in register **d2**. The driver must translate this count into the media's physical block count. If the tape is incapable of directly skipping backward, it has to simulate the reverse movement using rewind and skip forward commands.

The sequence of actions for **SS\_Skip** is the same as that for **SS\_SQD**.

**SS\_SQD** This routine is called to unload the tape (put the tape device off-line). Depending upon the capabilities of the tape device, this action may turn off the drive-select LED, or unload and eject the media.

The unload routine should:

- Initialize the drive, if required.
- Issue the appropriate command to the device and wait for I/O to complete (with interrupts, if possible).
- Check the status of the I/O command and return any error to SBF.
- If the driver maintains flags pertaining to current tape position, these should be updated.
- Return status to SBF.

**SS\_WFM** This routine is called to write the specified number of filemarks to the tape. (This number is passed in register **d2**.) Applications may place filemarks on the tape as they see fit. The sequence of actions for **SS\_WFM** is the same as that for **SS\_SQD**.

**TERM****Terminate Device**

**INPUT:** (a1) = address of the device descriptor module  
 (a2) = address of device static storage area  
 (a4) = process descriptor pointer  
 (a6) = system global static storage

**OUTPUT:** None

**ERROR** cc = carry set  
**OUTPUT:** dl.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:

- Wait until any pending I/O has completed.
- ! Disable the device interrupts.
- Æ Remove the device from the IRQ polling list.
- Ø Kill the driver process created by SBF. If SBF\_DPrc is non-zero, this is a pointer to the driver's process descriptor. This process is returned by making a F\$DelPrc system call with the process ID from P\$ID.
- × If the driver maintains flags/variables that must "span" detach/attach sequence, then the TERM routine should unlink any external modules linked to during INIT.

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

**OUTPUT:** None

**ERROR**

**OUTPUT:** cc = carry 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.

- Æ Wake up the driver mainline, using the synchronization method of the driver:

- Signals:** Send a wake-up signal to the process whose process ID is in V\_WAKE, when the I/O is complete. Also, clear V\_WAKE as a flag to the mainline program that the IRQ has occurred.

- Events:** Signal the event that the IRQ has occurred, using the event system's signal function.

- Ø Clear the carry bit and exit with an 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 following registers only: d0, d1, a0, a2, a3, and a6. The contents of all other registers must be preserved or unpredictable system errors (system crashes) will occur.

**End of Chapter 4**





**NOTES**