EPOC16 System Calls

Introduction

The Psion System 3 operating system offers a large number of system calls.

A system call is identified by either a "function number" or by a "function number and a "subfunction number". These are abbreviated to "Fn" and "Sub" in the descriptions of system calls.

Each call can take up to 5 word parameters, called `BX`, `CX`, `DX`, `SI`, and `DI`. In addition, there may be up to two byte parameters (`AH` and `AL`) or an additional word parameter (`AX`). The values returned from the call have the same structure; in addition, there is an error flag and three result flags (the latter are only used by a few system calls). Note that `AX` is equivalent to `AH * 256 + AL`; the description will use whichever is most convenient.

There are two OPL keywords for making system calls: `CALL` and `OS`. They access the same system calls, and differ only in the way in which the parameters are passed and the results returned.

The `CALL` keyword takes from 1 to 6 arguments. If an argument is omitted, the corresponding parameter is set to an undefined value. The first argument should be one of:

Fn
Fn + 256 * Sub
Fn + 256 * AH

according to the specific system call. The remaining arguments provide the word parameters:

- argument 2: `BX`
- argument 3: `CX`
- argument 4: `DX`
- argument 5: `SI`
- argument 6: `DI`

The keyword returns a value, which is one of the results:

`AX`
`AH * 256 + AL`

(which are equivalent). The flags and the other 5 results are not available.

OS takes two or three arguments; if there is no third, it is taken to be identical to the second. The first argument is Fn, while the second and third are each the address of a 12 byte buffer, holding the parameters and results
respectively:

Offset 0 (word): one of:
Sub * 256 + AL (parameters only)
AX (parameters and results)
AH * 256 + AL (parameters and results)
Offset 2 (word): `BX`
Offset 4 (word): `CX`
Offset 6 (word): `DX`
Offset 8 (word): `SI`
Offset 10 (word): `DI`

The keyword returns a value which represents the error or result flags. Each flag is held in a specific bit of the value; the remaining bits are unspecified:

Bit 0: `UL` flag or error flag (depending on context)
Bit 6: `EQ` flag
Bit 7: `SL` flag

Thus, if the call can fail, it has failed if the returned value is odd.

When a parameter or result is described as a `cstr`, a `qstr`, a buffer, or any other construct requiring more than 2 bytes, the actual parameter or result is the address of the construct.

When a register holds an address (for the above reason or otherwise), it is relative to some segment register. If programming entirely in OPL, all the relevant segment registers (DS, SS, and ES) are the same. However, if assembler is being used, it is sometimes useful to have ES different from the others (see the Psionics file PROCESS for more details). A prefix of "e" to a register name (such as "eBX") means that the address is relative to ES. Unprefixed names are relative to DS or SS (which should be the same). A prefix of "u" means that the segment register is unknown.

The system calls are described in a standard form. The first line gives the following information:

- the Fn value
- the Sub value if any

The second line gives the following:

- The name given to the call by the SDK.

- A note such as "v3.1" giving the first version of the operating system to support the call; if omitted, all versions support it.
- The word "fails" if the call can fail; if omitted, the call never fails.
  - If a call can fail, then the error flag shows whether or not it did so. If it failed, AL holds an error code between 128 and 255; this is normally treated as a negative number between -1 and -128, found by evaluating the expression (AX OR $FF00). AH, and any of the 5 word results that the call would have changed, are unspecified.
  - If a call cannot fail, the error flag may still be set; this should be ignored.
  - Even if a call cannot fail, bad parameters can cause a system panic.
- The word "async" if the call is always asynchronous, or the word "vsync" if it is sometimes asynchronous; if omitted, the call is synchronous.
  - A synchronous call is "completed" (has carried out all its actions) when the system call returns.
  - An asynchronous call returns immediately, even though the call might not have completed. A parameter of the call specifies a "status word", which the call initially sets to -46 (the error code meaning "not completed"). When the relevant action has been taken or event occurs, the call is completed, the status word is changed to indicate the result of the call, and an `IOSIGNAL` is sent to the process. This is exactly the same behaviour as the `IOA` keyword.
  - If a call is "vsync", then it may be made as either a synchronous or asynchronous call:
    - to make a synchronous call, set `AL` to zero (the parameter giving the address of the status word will be ignored);
    - to make an asynchronous call, set `AL` to any other value (the address of the status word must be valid);
    - descriptions of such calls will omit `AL` from the list of parameters.
  - An asynchronous call may complete before returning (for example if the relevant event has already happened). In this case the process will never see the status word with the value -46, but an `IOSIGNAL` is still received.
  - Some asynchronous calls have a corresponding "cancel" call. This causes the original call to be completed with the status word set to some negative value other than -46.

This is then followed by a list of the parameters and results used by the call; any parameter not listed is ignored. For the 5 main word results, the result (if the call does not fail) is the same as the corresponding parameter (including when the former is ignored) unless a different meaning is shown following an arrow (->). For the results AH, AL, and AX, and also the result flags, the result is unspecified if no such description is given.

For example, consider the following (non-existent) system call:

Fn $12 Sub $34
GetMagicCookie fails
AX: -> magic cookie table entry zero
BX: table entry number

CX: -> magic cookie entry
SI: addend -> CX out + SI in
Returns entries from the magic cookie table.

This system call could be used either by:

```
entry0% = CALL ($3412, entry%, 0, 0, addend%)
```

which only returns magic cookie table entry zero, or by:

```
LOCAL regs%(6)
regs%(1)=$3400
regs%(2)=entryno%
regs%(5)=addend%
failed%=1 AND OS ($12, ADDR(regs%()))
REM regs%(2,4,6) have not changed
IF NOT failed
 entry0%=regs%(1)
 entry%=regs%(3)
 REM regs%(5) will equal entry%+addend%
ELSE
 REM regs%(1,3,5) are undefined
 error%=regs%(1) and $FF0ENDIF
```

which returns much more information.

System calls

Note that some system calls are deliberately not described.

Function `0x80`

Sub-function `0x00`: `SegFreeMemory`

AX: -> free memory

Returns the amount of free system memory, in units of 16 bytes.

Sub-function 0x01: SegCreate fails

AL: segment type: ordinary processes should use 1.
AX: -> segment handle
eBX: (cstr) name of the segment
CX: segment size in units of 16 bytes

Creates an additional memory segment. Each segment must be given a name, of the form "8.3" (i.e. up to 8 characters, optionally followed by a dot and up to 3 more characters). If the name is already in use, the call will fail. Once created, the segment may be used by the process as additional memory, either via `SegCopyFrom` and `SegCopyTo`, or in assembler code (see the Psionics file KERNEL).

Sub-function `0x02`: `SegDelete` fails

eBX: segment handle

Deletes an additional memory segment. If any other process has opened the segment, the call will fail.

Sub-function `0x03`: `SegOpen` fails

AX: -> segment handle
eBX: (cstr) name of the segment

Opens the additional memory segment with the given name (if no such segment exists, the call will fail). The call will fail if the process has the segment open already. Except where stated, all calls using additional memory segments must be given handles from `SegCreate` or `SegOpen` calls.

Sub-function `0x04`: `SegClose` fails

BX: segment handle
Closes an additional memory segment which the process has open. If this was the only process with the segment open, it is deleted. Open segments are closed when a process terminates.

Sub-function `0x05`: `SegSize`

AX: -> segment size
BX: segment handle

Returns the size of an additional memory segment in units of 16 bytes.

Sub-function `0x06`: `SegAdjustSize` fails

BX: segment handle
CX: new size in units of 6 bytes

Changes the size of an additional memory segment. The memory will be added to or removed from the end of the segment.

Sub-function `0x07`: `SegFind` fails

AX: -> segment handle
BX: last segment handle
SI: address of 14 byte buffer used by the call
eDI: (cstr) pattern to search for

Finds the additional memory segments with names matching the search pattern (use ? and * as wild cards). The first call should be made with the last segment handle set to 0, and subsequent calls should set it to the handle returned by the previous call. The call fails when there are no more segments to return. It is not necessary to keep calling `SegFind` until this happens.

Sub-function `0x08`: `SegCopyTo`

BX: segment handle
CX: number of bytes to copy
DX: high half of start address in segment
SI: address in process of first byte to copy
DI: low half of start address in segment

Copies data from the current process to an additional memory segment.

Sub-function `0x09`: `SegCopyFrom`

BX: segment handle
CX: number of bytes to copy
DX: high half of start address in segment
SI: address in process to copy first byte to
DI: low half of start address in segment

Copies data from an additional memory segment to the current process.

Sub-function `0x0A`: `SegLock`

BX: segment handle

Locks an additional memory segment; a locked segment will not be deleted even if no processes have it open.

Sub-function `0x0B`: `SegUnLock`

BX: segment handle

Unlocks an additional memory segment. The number of unlock calls should equal the number of lock calls; additional unlocks may cause the segment to be deleted while still open.

Sub-function `0x0C`: `SegRamDiskUsed`

AX: -> size of the ram disc in units of 16 bytes

Returns the size of the ram disc; this should be treated carefully, as it can change without warning.

Sub-function `0x0D`: `SegCloseLockedOrDevice`

BX: segment handle

Unlocks an additional memory segment where the handle may be that for a different process. See SegUnLock for more details.

Function `0x81`

Sub-function `0x00`: `HeapAllocateCell` fails

AX: -> allocated block
CX: number of bytes to allocate

Allocates a block of memory from the heap. The returned block may be larger than requested. Heap memory is part of the process's normal memory segment.

Sub-function `0x01`: `HeapReAllocateCell` fails

AX: -> new address of block
BX: block to be reallocated, or zero

CX: new size of block in bytes

Changes the size of a heap block; the block may have to be moved to do this. The memory will be added or deleted to the end of the block. If the call fails, the block is unaffected. If `BX` is zero, then a new cell is allocated. The new block may be larger than requested.

Sub-function `0x02`: `HeapAdjustCellSize` fails

AX: -> new address of block
BX: block to be adjusted (must not be zero)
CX: number of bytes to add or remove
DX: location within block of the change

Changes the size of a heap block; the block may have to be moved to do this. If the call fails, the block is unaffected. The memory will be added or removed at the offset given by `DX`. For example, if the block was 10 bytes long, `CX` is 3, and `DX` is 6, the new block will be 13 bytes, and consist of the first 6 bytes of the old block, then 3 random bytes, then the last 4 bytes of the old block. The new block may be larger than requested.

Sub-function `0x03`: `HeapFreeCell`

BX: block to free

Frees a previously allocated heap block.

Fn $81 Sub $04: HeapCellSize

AX: -> cell size in bytes
BX: address of block

Returns the actual size of a heap block (which may be larger than requested when the block was created or last resized).

Sub-function `0x05`: `HeapSetGranularity`

BX: new granularity in units of 16 bytes

When the application data space is not large enough to satisfy a heap request, it is grown by the granularity. The default is 2kb (128 units), and the maximum is 16k (1024 units).

Sub-function `0x06`: `HeapFreeMemory`

AX: -> maximum possible free space on heap
BX: -> address of start of heap

Returns the amount of heap space which can be allocated (by using freed blocks or by growing the data space, or both), plus the address of the base of the heap.

Function `0x82`

Sub-function `0x00`: `SemCreate` fails

AX: -> semaphore handle
BX: initial value of semaphore

Creates a new semaphore for process interlocking. Each semaphore has a list of processes associated with it (this list is initially empty) and a value, which must initially be zero or positive.

Sub-function `0x01`: `SemDelete`

BX: semaphore handle

Deletes a semaphore. Any processes on the semaphore's list will be restarted. When a process terminates, any semaphore it created will be deleted.

Sub-function `0x02`: `SemWait`

BX: semaphore handle

If the value of the semaphore is positive or zero, one is subtracted from it. If the value is now or initially -1, the current process is then blocked and added to the end of the list for that semaphore.

Sub-function `0x03`: `SemSignalOnce`

BX: semaphore handle

If the list for the semaphore is not empty, the first process on the list is removed and restarted. If the list is empty (either initially or after the only process is removed), one is added to the value of the semaphore. If the restarted process has a higher priority than the current one, a reschedule will take place and the other process will start running (see the description of priorities in the Psionics file PROCESS).

Sub-function `0x04`: `SemSignalMany`

BX: semaphore handle
CX: number of calls to make (must be positive)

This is equivalent to making several calls to `SemSignalOnce`.

Sub-function `0x05`: `SemSignalOnceNoReSched`

BX: semaphore handle

This is identical to SemSignalOnce except that a reschedule never takes place because of the call (though one may take place due to the current process using up its time slot).

Function 0x83

Sub-function `0x00`: `MessInit` fails

BX: number of message slots + 256 * maximum data length

Initialize the message system so that the current process can receive inter-process messages. The call reserves the indicated number of message slots, with each slot having room for the indicated amount of data. The slots are allocated on the heap.

Sub-function `0x01`: `MessReceiveAsynchronous` async

BX: address of message slot pointer
DI: address of the status word

When a message arrives, the message slot pointer is set to the address of the message, the status word is set to zero, and the call completes. The message has the format:

Offset 0 to 3: used by the message system
Offset 4 (word): message type
Offset 6 (word): process sending the message
Offset 8 onward: data in message

Sub-function `0x02`: `MessReceiveWithWait`

BX: address of message slot pointer

When a message arrives (this may have happened before the call; otherwise the call waits until a message arrives), the message slot pointer is set to the address of the message. The message format is as for `MessReceiveAsynchronous`.

Sub-function `0x03`: `MessReceiveCancel`

Cancel any pending `MessReceiveAsynchronous`.

Sub-function `0x04`: `MessSend` fails

BX: process to receive the message
CX: message type
SI: address of first byte of message data

Send a message to a process. The call will block until there is a free message slot in the receiving process. The receiving process determines the amount of data sent. The reply is ignored.

If the sender has priority 128 or above, the message is placed at the front of the queue of messages waiting for the recipient (if it is not already waiting for a message). Otherwise it is placed at the back of the queue.

Sub-function `0x05`: `MessSendReceiveAsynchronous` fails async

BX: process to receive the message
CX: message type
SI: address of first byte of message data
DI: address of the status word

Send a message to a process; when the recipient replies, the status word is set to the reply and the process is sent an IOSIGNAL. The call will block until there is a free message slot in the receiving process.

Sub-function `0x06`: `MessSendReceiveWithWait` fails

AX: -> reply
BX: process to receive the message
CX: message type
SI: address of first byte of message data

Send a message to a process, and blocks until the recipient replies.

Sub-function `0x07`: `MessFree`

BX: address of received message
CX: reply

The reply is sent to the recipient of the message, and the message slot is freed and can be used for another incoming message.

Sub-function `0x08`: `MessSignal` fails

BX: process to watch
CX: message type

Requests that, when the specified process terminates, the kernel sends a message of the indicated type to the current process. The data in the message has the format:

Offset 0 (word): process id of terminating process
Offset 2 (byte): reason code (see ProcKill) or panic code
Offset 3 (byte):

0 = process terminated or was killed
1 = process caused a panic
2 = a subtask of the process caused a panic in the process

If the message slots do not have room for 4 bytes of data, not all this information is available.

Sub-function `0x09`: `MessSignalCancel` fails

Sub-function `0x0A`: `MessSignalCancelX` fails

BX: process to watch
CX: message type (`MessSignalCancelX` only)

Cancels a call to MessSignal for the indicated process. `MessSignalCancel` ignores the message type and should not be used if more than one MessSignal call has been made for the process.

Function `0x84`

Used to control dynamic libraries and is not described here.

Function `0x85`

Fn $85 Sub $00
IoOpen fails
AX: -> handle of opened device
eBX: (cstr) name of device driver
CX: mode
DX: handle of device being attached to
Opens a channel to a device driver. If the driver is a base driver, then DX is ignored (and this call is equivalent to the IOOPEN keyword). If it is a stacked driver (see the Psionics file DEVICES), then the handle to be stacked on must be specified. The meaning of the mode is determined by the device. A driver can support several channels at once, and each has its own handle. The driver must be a logical device driver.

Fn $85 Sub $01 and $02 involve physical device drivers and should only be called from within logical device drivers.

Fn $85 Sub $03 to $05 should only be used by the operating system.

Fn $85 Sub $06
DevLoadLDD fails
Fn $85 Sub $07
DevLoadPDD fails
eBX: (cstr) name of file holding the driver
Loads a device driver into the system. A driver cannot be opened until it is loaded. The correct call for the driver type must be used.

Fn $85 Sub $08
DevDelete fails
eBX: (cstr) name of the device driver
DX: $DD01 for logical device drivers, or $DD21 for physical device drivers
Unloads a device driver from the system. Open drivers and those in the ROM cannot be unloaded.

Fn $85 Sub $09
DevQueryUnits fails
AX: -> number of channels supported
eBX: (cstr) name of the device driver, without a trailing colon
Returns the number of simultaneous open channels supported by a driver (which must be a logical one); $FFFF indicates no limit.

Fn $85 Sub $0A
DevFind fails
AX: -> find code
BX: last find code
DX: $DD01 for logical device drivers, or $DD21 for physical device drivers
SI: address of 14 byte buffer
eDI: (cstr) pattern to search for
Finds all devices with names matching the search pattern (use ? and * as wild cards). The first call should be made with the last find code set to 0, and subsequent calls should set it to the code returned by the previous call. The call fails when there are no more drivers to return. It is not necessary to keep calling DevFind until this happens. The names do not have a trailing colon.

Fn $85 Sub $0B should only be used by the operating system.

Fn $85 Sub $0C is used for special operations on device drivers.

Function 0x86

Fn $86 Sub $00
IoAsynchronous fails async
Fn $86 Sub $01
IoAsynchronousNoError async
Fn $86 Sub $02

IoWithWait fails
AL: service number
AX: -> result from driver
BX: handle of driver
CX: address of first argument
DX: address of second argument
DI: address of the status word (ignored by IoWithWait)
These calls are equivalent to the IOA, IOC, and IOW keywords respectively
[For those without documentation of IOC, this is the same as IOA, except that errors are handled by setting the status word and calling IOSIGNAL, so that IOC always succeeds, unlike IOA.]

Fn $86 Sub $03 and $04 should only be used by device drivers.

Fn $86 Sub $05
IoWaitForSignal
This call is equivalent to the IOWAIT keyword.

Fn $86 Sub $06
IoWaitForStatus
DI: address of the status word
This call is equivalent to the IOWAIT keyword.

Fn $86 Sub $07
IoYield
DI: address of the status word
This call is equivalent to the IOYIELD keyword.

Fn $86 Sub $08
IoSignal
Sends an IOSIGNAL to the current process. This call is equivalent to the IOSIGNAL keyword.

Fn $86 Sub $09
IoSignalByPid fails
Fn $86 Sub $0A
IoSignalByPidNoReSched fails
BX: process ID
Sends an IOSIGNAL to a process. With the latter call a reschedule never takes

place because of the call (though one may take place due to the current process using up its time slot).

Fn $86 Sub $0B to $0F should only be used by device drivers.

Fn $86 Sub $10
IoClose fails
AX: -> result from driver
BX: handle
This call is equivalent to the IOCLOSE keyword.

Fn $86 Sub $11
IoRead fails
Fn $86 Sub $12
IoWrite fails
AX: -> amount actually read or result from driver
BX: handle
CX: address of buffer
DX: number of bytes read or written
These calls are equivalent to the IOREAD and IOWRITE keywords.

Fn $86 Sub $13
IoSeek fails
AX: -> result from driver
BX: handle
CX: mode
DX: address of long holding seek position
This call is equivalent to the IOSEEK keyword.

Fn $86 Sub $14 should only be used by the window manager.

Fn $86 Sub $15 to $17 should only by used from assembler.

Fn $86 Sub $18
IoShiftStates
AL: modifiers
This call makes available the state of the various modifier keys:
Bit 1: shift

Bit 2: control
Bit 3: psion

Fn $86 Sub $19
IoWaitForSignalNoHandler
This call should be used instead of IOSEEK by tasks (subsidiary processes of a process).

Fn $86 Sub $1A
IoSignalKillAsynchronous fails async
BX: process to watch
DI: address of the status word
This call completes, and the status word is set to 0, when the specified process terminates.

Fn $86 Sub $1B
IoSignalKillCancel fails
BX: process to watch
Cancel any pending IoSignalKillAsynchronous.

Fn $86 Sub $1C and $1D should only be used by the window manager.

Fn $86 Sub $1E
IoPlaySoundW v3
AL: -> failure code
BX: (cstr) sound file name
CX: duration to play (in 1/32 second)
DX: volume: 0 (loudest) to 5 (softest)
Plays a sound file. A duration of 0 means the file header specifies the duration; otherwise, the sound is clipped or padded with space as needed.
The name may be either an ordinary filename, or a string of the form "*ABCD", meaning the first found of the files:
ROM::ABCD.WVE
LOC::M:\WVE\ABCD.WVE
LOC::A:\WVE\ABCD.WVE
LOC::B:\WVE\ABCD.WVE
Note that this call does not fail in the usual way, but just returns a failure code.

Fn $86 Sub $1F
IoPlaySoundA v3 async
BX: (cstr) sound file name
CX: duration to play (in 1/32 second)
DX: volume: 0 (loudest) to 5 (softest)
DI: address of the status word
Plays a sound file asynchronously; the call completes when the sound has finished. The other arguments are as for IoPlaySoundW.

Fn $86 Sub $20
IoPlaySoundCancel v3
Cancel any pending IoPlaySoundA.

Fn $86 Sub $21
IoRecordSoundW v3
AL: -> failure code
BX: (cstr) sound file name
CX: number of samples to record, in units of 2048 samples
Records a sound file. Note that 2048 samples are slightly more than a quarter of a second. The file will be created before recording, and must not be on a flash device. Note that this call does not fail in the usual way, but just returns a failure code.

Fn $86 Sub $22
IoRecordSoundA v3 async
BX: (cstr) sound file name
CX: number of samples to record, in units of 2048 samples
DI: address of the status word
Records a sound file asynchronously; the call completes when the recording has finished. The other arguments are as for IoRecordSoundW.

Fn $86 Sub $23
IoRecordSoundCancel v3
Cancel any pending IoRecordSoundA.

Fn $86 Sub $24
IoPlaySoundAO v3.9 async
BX: (cstr) sound file name
CX: duration to play (in 1/32 second)

DX: volume: 0 (loudest) to 5 (softest)
DI: address of the status word
SI: duration to skip (in 1/32 second)
Plays part of a sound file asynchronously, skipping some initial portion of the sound; the call completes when the sound has finished. The other arguments are as for IoPlaySoundA.

Function `0x87`

Fn $87 Sub $00 is carried out automatically for OPL programs.

Fn $87 Sub $01
FilExecute fails vsync
eBX: (cstr) program file name
CX: (qstr) program information
DX: address of the status word
DI: address of word to be filled with process ID of new process
Starts a new process and places it in "suspended" state. The program file name will have the extension ".IMG" added if one is not specified. The program information is a qstr whose interpretation depends on the program; it is available from location 36 of the new process's data segment (see Psionics file PROCESS). The process name will be set to the name part of the file (excluding any extension). If another process of this name is already running, both must execute the same code, or the call will fail.

> More in S3/S3a manual chapters 1 and 2.

Program information often consists of the following catenated together (note that all these elements except the first are cstrs):

(1) the character 'C' (create new file) or 'O' (open existing file);
(2) (cstr) the application name, with the first character uppercased and the rest lowercased;
(3) (cstr) one of:
(a) the normal extension for filenames, including the leading dot;
(b) the normal extension with dot, followed by a space and aliasing information;
(c) an empty string
(4) (cstr) the data file path passed as the initial argument;
(5) (cstr) the application path (may be omitted if not required).

A binary application may be used to operate on a standard file type. Examples include:

Program Application Extension
[item 2] [item 3]
"ROM::DATA.APP" "Data" ".DBF"
"ROM::WORD.APP" "Word" ".WRD"
"ROM::WORD.APP" "Program" ".OPL"
"LOC::C:\APP\COMMS.APP" "Comms" ".SCO"

In each of these cases there is no aliasing information or application path. An example program information qstr is "OData~.DBF~LOC::M:\DBF\MYDATA.DBF~" (tilde indicates a zero byte).

> @This section needs rewriting@

The word processor may also be used as a program editor, by making item 2 its name ("Program") and item 3 an extension (usually ".OPL") followed by the necessary aliasing information: this is empty for normal Word, or:

* "O" for OPL editor, "S" for SCR editor, "$" for plain text editor (3a only), "/" for custom template (3a only); any letter uses translator `SYS$PRG<letter>.IMG` to translate, then
* then "R" indicates that the translator can execute resulting code, anything else means it can't, then
* extension and directory ("OPO" or "SCO" for example) of translated files,
* then on 3a only optional "*" to indicate compatibility mode exists.

An example program information qstr is

"Program~.OPL OROPO~LOC::M:\OPL\MYPROG.OPL~"

template mode for Word means there must be a \WDR\<template>.WRT file on the current drive; template name is that if the application name. For example, alias file of:

Letter.LET
\LET\
1083
Word
/

uses template LETTER. Note mode 80 in application type in alias file.

Finally, a translator may be used to execute a translated file: in the case of the translator "ROM::SYS$PRGO", it can execute either OPO or OPA files. The application name is given in the APP ... ENDA clause for OPA files, and is "RunOpl" for OPO files; there is no extension or aliasing information.

For OPA files, the data file is passed to the application, and the application path shows which application is actually run. For OPO files, the data file should name the OPO file itself, and there is no application path. Examples are "RunOpl~~M:\OPO\MYPROG.OPO~" and "MyApp~~M:\DAT\MYDATA~M:\APP\MYAPP.OPA~".

Fn $87 Sub $02
FilParse fails vsync
BX: (cstr) file name to be parsed
CX: (cstr) file specification to be used
DX: address of the status word
SI: address of a 6 byte buffer to be filled in
DI: address of 128 byte buffer to be filled with the parsed filename

This call is equivalent to the `PARSE$` keyword. The buffer is filled as follows:

Offset 0 (byte): length of node name
Offset 1 (byte): length of device name
Offset 2 (byte): length of path name
Offset 3 (byte): length of base name
Offset 4 (byte): length of extension
Offset 5 (byte): flags
Bit 0: the base name or extension contains a wildcard ("*" or "?")
Bit 1: the base name contains a wildcard
Bit 2: the extension contains a wildcard

Fn $87 Sub $03
FilPathGet fails vsync
BX: 128 byte buffer
DX: address of the status word
The buffer is filled with the current filing system default path (a cstr).

Fn $87 Sub $04
FilPathSet fails vsync
BX: (cstr) new path
DX: address of the status word
Sets the default path (equivalent to the SETPATH keyword).

Fn $87 Sub $05
FilPathTest fails vsync
BX: (cstr) pathname to be tested
DX: address of the status word

Equivalent to the EXIST keyword; the call succeeds if the file with that pathname exists.

Fn $87 Sub $06
FilDelete fails vsync
BX: (cstr) pathname to be deleted
DX: address of the status word
Equivalent to the DELETE keyword; non-empty directories cannot be deleted.

Fn $87 Sub $07
FilRename fails vsync
BX: (cstr) old pathname
CX: (cstr) new pathname
DX: address of the status word
Equivalent to the RENAME keyword.

Fn $87 Sub $08
FilStatusGet fails async
BX: (cstr) filename
CX: 16 byte buffer
DX: address of the status word
The buffer is filled in with information about the file:
Offset 0 (word): buffer format version: always 2
Offset 2 (word): attributes
Offset 4 (long): file size in bytes
Offset 8 (long): time last modified

The attribute bits have the following meanings (question marks indicate that the bit name was specified but the meaning is unknown; equals signs mean that the bit can be set by FilStatusSet):
= Bit 0: file is not ReadOnly
= Bit 1: file is Hidden
= Bit 2: file is a System file
Bit 3: file is a Volume name
Bit 4: file is a Directory
= Bit 5: file is Modified
? Bit 8: file is a readable file
? Bit 9: file is an executable file
? Bit 10: file is a byte stream file
Bit 11: file is a text file
The LOC:: node (see Psionics file FILEIO) does not distinguish text and binary

files, and always leaves bit 11 clear. Other nodes may distinguish the types of files and set the bit where appropriate.

Fn $87 Sub $09
FilStatusSet fails vsync
BX: (cstr) filename
CX: mask of attributes to be changed (only specify settable bits)
DX: address of the status word
DI: new values for attributes to be altered
The attributes indicated in the mask of the specified file are altered to the new values; all other attributes are left unchanged.

Fn $87 Sub $0A AL sync
FilStatusDevice fails vsync
BX: (cstr) device name (such as "A:" or "LOC::A:")
CX: 64 byte buffer
DX: address of the status word
The buffer is filled in with information about the device:
Offset 0 (word): buffer format version
Offset 2 (byte): device type
0 = unknown
1 = floppy
2 = hard disc
3 = flash
4 = ram
5 = rom
6 = write-protected
Offset 3 (byte): device properties:
Bit 3: formattable device
Bit 4: dual density device
Bit 5: internal device
Bit 6: dynamically sizeable device
Bit 7: compressible (worth compressing database files)
Offset 4 (word): non-zero if the device contains removable media
Offset 6 (long): total space on the device in bytes
Offset 10 (long): free space on the device in bytes
Offset 14 (cstr): volume name
Offset 46 (word): device battery status
0 = low battery
1 = battery OK
-4 = status not available for this device type

This field is not valid if the format version (offset 0) is less than 3.

Fn $87 Sub $0B
FilStatusSystem fails vsync
BX: (cstr) node name (such as "LOC::" or "REM::")
CX: 32 byte buffer
DX: address of the status word
The buffer is filled in with information about the node:
Offset 0 (word): buffer format version (currently 2)
Offset 2 (word): 0 = flat, 1 = hierarchical
Offset 4 (word): non-zero if devices are formattable, and zero otherwise

Fn $87 Sub $0C
FilMakeDirectory fails vsync
BX: (cstr) pathname
DX: address of the status word
This is equivalent to the MKDIR keyword.

Fn $87 Sub $0D AL sync
FilOpenUnique fails vsync
eBX: (cstr) pathname
CX: open mode (format and access parts only)
DX: address of the status word
Open a file with a unique name. This is equivalent to the IOOPEN keyword with a mode of 4. The pathname is used to determine the directory to create the file in, and will be modified to give the actual name (thus the pathname should have room for 128 characters).

Fn $87 Sub $0E
FilSystemAttach fails vsync
BX: (cstr) name of a file system PDD
DX: address of the status word
The specified physical device driver will be attached to the filing system, possible adding new nodes.

Fn $87 Sub $0F
FilSystemDetach fails vsync
BX: (cstr) name of a filing system
DX: address of the status word

Detaches a filing system. Built-in filing systems cannot be detached.

Fn $87 Sub $10
FilPathGetById fails vsync
BX: process to examine
CX: 128 byte buffer
DX: address of the status word
The buffer is set to the current path of the specified process (a cstr).

Fn $87 Sub $11
FilChangeDirectory fails vsync
BX: (cstr) pathname
CX: 128 byte buffer
DX: address of the status word
SI: (cstr) name of subdirectory
DI: 0 = get root directory, 1 = get parent directory, 2 = get subdirectory
The path name is modified in the requested way. This call should be used instead of manipulating the directory part of a name directly, as it works on all nodes, irrespective of the format that the node uses for path names.

Fn $87 Sub $12 has no effect on OPL processes.

Fn $87 Sub $13
FilSetFileDate fails vsync
BX: (cstr) pathname
CX: low word of new time
DX: address of the status word
DI: high word of new time
Sets the modification time of the specified file.

Fn $87 Sub $14
FilLocChanged fails vsync
AX: -> channel change flags
BX: channel to check
DX: address of the status word
Specifies whether any directory in the LOC:: node has changed (i.e. a file or directory has been created, destroyed, or renamed, but not changes to file contents) since the last use of this system call with this channel. BX should have exactly one bit set, corresponding to the channel to use; bits 8 to 15

are reserved by Psion. The corresponding bit in AX will be set if the node has changed, and will be clear otherwise; the other 15 bits are unspecified. This call can be used to determine whether to update a directory display.

Fn $87 Sub $15
FilLocDevice v3 fails vsync
BX: local device indicator (%A to %H, or %I or %M)
CX: address of word set to the media type
DX: address of the status word
Provides the media type of a device on the LOC:: node (I and M both refer to the internal ramdisc). The device type consists of:
Bits 0 to 3:
0 = unknown
1 = floppy
2 = hard disc
3 = flash
4 = ram
5 = rom
6 = write-protected
Bits 7 to 8:
0 = device battery measurement not supported
1 = device battery measurement not supported
2 = device battery voltage low
3 = device battery voltage good

Fn $87 Sub $16
FilLocReadPDD v3 fails vsync
BX: local device indicator (%A to %H, or %I or %M)
CX: (long) location on the device
DX: address of the status word
SI: number of bytes to read
DI: address in process to copy first byte to
Copies data from a device on the LOC:: node to the current process. This call is very efficient, and accesses the raw device.

PSIONICS FILE - SYSCALLS.2
==========
System calls (part 2)
Last modified 1998-11-07
==========

See part 1 for general notes and explanations.

Function `0x88`

Fn $88 Sub $00
ProcId
AX: -> process ID of the current process
Gets the process ID of the current process.

Fn $88 Sub $01
ProcIdByName fails
AX: -> process ID
eBX: (cstr) pattern
Gets the process ID of a process whose name matches the pattern (usual wildcards apply, and case is ignored).

Fn $88 Sub $02
ProcGetPriority fails
AL: -> priority
BX: process ID
Gets the priority of the specified process.

Fn $88 Sub $03
ProcSetPriority fails
AL: new priority
BX: process ID
Sets the priority of the specified process.

Fn $88 Sub $04 should only be called by the operating system.

Fn $88 Sub $05
ProcCreateTask
Tasks are processes which share the data segment of another process. They cannot be conveniently handled in OPL.

Fn $88 Sub $06
ProcResume fails
BX: process ID

Take a process out of the suspended state and start it executing.

Fn $88 Sub $07
ProcSuspend fails
BX: process ID
Place a process in the suspended state. If the process is waiting for a system service (e.g. a semaphore), then it will be suspended when the service has been carried out.

Fn $88 Sub $08
ProcKill fails
AL: reason code
BX: process ID
Kills the specified process, without allowing it to execute any cleanup code. Only use this on the current process or in emergencies.

Fn $88 Sub $09
ProcPanicById fails
AL: panic code
BX: process ID
Simulate the specified panic on the specified process.

Fn $88 Sub $0A
ProcNameById fails
BX: process ID
eDI: 13 byte buffer
Places the name (a cstr) of the specified process in the buffer.

Fn $88 Sub $0B
ProcFind fails
AX: -> process ID
BX: 0 or process ID
SI: 14 byte buffer
eDI: (cstr) pattern
Obtains process IDs for processes whose name matches the pattern ("?" and "*" wildcards have their Unix meaning, and case is ignored). BX should be zero to return the first matching process, or the process ID returned by a previous call to return subsequent matching processes. Process are returned in *task* ID order. The buffer is filled with a cstr giving the process name.

Fn $88 Sub $0C
ProcRename fails
BX: process ID
eDI: (cstr) new name
Rename the specified process; the new name must be between 1 and 8 characters.

Fn $88 Sub $0D
ProcTerminate fails
BX: process ID
Terminates the indicated process. The process will be sent a termination message if it has so requested, and will be killed otherwise.

Fn $88 Sub $0E
ProcOnTerminate
BX: message type
When the current process is terminated, it will be sent a message of the specified type; type 0 cancels the request. After receiving the message and executing any clean-up code, the process should use ProcKill to kill itself.

Fn $88 Sub $0F is reserved for the Shell process.

Fn $88 Sub $10
ProcGetOwner fails
AX: -> owning process ID
BX: process ID
Gets the ID of the process owning the specified process (normally the

creator of that process).

Function `0x89`

Fn $89 Sub $00
TimSleepForTenths fails
CX: high half of delay
DX: low half of delay
Sleep for the specified delay (in units of 0.1 seconds). Changing the system clock does not affect the call. Only time when the machine is switched on is measured; any time when the machine is switched off will be in addition to the requested delay.

Fn $89 Sub $01
TimSleepForTicks fails
CX: high half of delay
DX: low half of delay
Sleep for the specified delay (in system ticks; there are 32 ticks per second on the Series 3 and 18.2 on the PC emulation). Changing the system clock does not affect the call. Only time when the machine is switched on is measured; any time when the machine is switched off will be in addition to the requested delay.

Fn $89 Sub $02
TimGetSystemTime
AX: -> high half of system clock
BX: -> low half of system clock
Reads the system clock (an abstime).

Fn $89 Sub $03
TimSetSystemTime
CX: high half of new system clock
DX: low half of new system clock
Sets the system clock to the given abstime.

Fn $89 Sub $04
TimSystemTimeToDaySeconds
CX: high half of abstime

DX: low half of abstime
DI: 8 byte buffer
Splits an abstime into a day number and an interval, placed in the buffer as follows:
Offset 0 (long): day number
Offset 4 (long): interval

Fn $89 Sub $05
TimDaySecondsToSystemTime fails
AX: -> high half of abstime
BX: -> low half of abstime
SI: 8 byte buffer
Converts a day number and an interval to an abstime. The former are in the buffer, in the same format as TimSystemTimeToDaySeconds.

Fn $89 Sub $06
TimDaySecondsToDate fails
SI: 8 byte buffer (day number and interval)
DI: 8 byte buffer (broken down time)
Converts a day number and an interval to broken-down time information. The former is in the same format as TimSystemTimeToDaySeconds. The latter is in the format:
Offset 0 (byte): year - 1900
Offset 1 (byte): month (0 = January, 11 = December)
Offset 2 (byte): day - 1
Offset 3 (byte): hours
Offset 4 (byte): minutes
Offset 5 (byte): seconds
Offset 6 (word): day number in year (0 to 364 or to 365)

Fn $89 Sub $07
TimDateToDaySeconds fails
SI: 8 byte buffer (broken down time)
DI: 8 byte buffer (day number and interval)
Converts broken-down time to a day number and an interval. The day number in year (offset 6) is ignored.

Fn $89 Sub $08
TimDaysInMonth fails
AX: -> number of days in month

CX: month * 256 + year - 1900
Gets the number of days in the specified month (0 = January, 11 = December).

Fn $89 Sub $09
TimDayOfWeek
AX: -> day of week (0 = Monday, 6 = Sunday)
CX: high half of day number
DX: low half of day number
Gets the day of the week of the given date.

Fn $89 Sub $0A
TimNameOfDay fails
AL: day of week (0 = Monday, 6 = Sunday)
BX: 33 byte buffer
The buffer is filled with a cstr giving the name of that day of the week.

Fn $89 Sub $0B
TimNameOfMonth fails
AL: month (0 = January, 11 = December)
BX: 33 byte buffer
The buffer is filled with a cstr giving the name of that month.

Fn $89 Sub $0C
TimWaitAbsolute fails
CX: high half of abstime
DX: low half of abstime
Sleep this process until the specified abstime. If the machine is turned off
at that time, it will turn back on. Changing the system clock will affect
when the call returns.

Fn $89 Sub $0D
TimWeekNumber fails
AX: -> week number (1 to 53)
CX: high half of day number
DX: low half of day number
Gets the week number of the specified day.

Fn $89 Sub $0E
TimNameOfDayAbb v3 fails

AL: day of week (0 = Monday, 6 = Sunday)
BX: 4 byte buffer
The buffer is filled with a cstr giving the abbreviated name of that day of the week. The length of the abbreviated name varies between languages, but is the same for all days in a given language.

Fn $89 Sub $0F
TimNameOfMonthAbb v3 fails
AL: month (0 = January, 11 = December)
BX: 4 byte buffer
The buffer is filled with a cstr giving the abbreviated name of that month. The length of the abbreviated name varies between languages, but is the same for all months in a given language.

Fn $8A Sub $00
ConvUnsignedIntToBuffer
AX: -> length of converted value
BX: value to be converted
CX: radix
eDI: buffer to hold converted value
The value is converted to a string in the specified radix and written to the buffer. No trailing zero byte is written; instead, the length of the string is returned. The radix can be any value from 2 to 200. If the radix is 11 or greater, digits greater than 9 are written as "A", "B", etc; characters other than digits and uppercase letters are used when the radix is 37 or more.

Fn $8A Sub $01
ConvUnsignedLongIntToBuffer
AX: -> length of converted value
BX: low half of value to be converted
CX: radix
DX: high half of value to be converted
eDI: buffer to hold converted value
The value is converted to a string, in the same way as ConvUnsignedIntToBuffer.

Fn $8A Sub $02
ConvIntToBuffer
AX: -> length of converted value
BX: value to be converted
eDI: buffer to hold converted value

The value is converted to a string in radix 10 and written to the buffer. If the value is negative, a leading "-" will be included. No trailing zero byte is written; instead, the length of the string is returned.

Fn $8A Sub $03
ConvLongIntToBuffer
AX: -> length of converted value
BX: low half of value to be converted
DX: high half of value to be converted
eDI: buffer to hold converted value
The value is converted to a string in radix 10, as for ConvIntToBuffer.

Fn $8A Sub $04
ConvArgumentsToBuffer
AX: -> length of converted format
BX: buffer holding the format arguments
SI: (cstr) format
eDI: buffer to hold converted format
The format is written to the buffer, with certain sequences of characters - "conversion specifiers" - being replaced by the values of arguments converted to strings. No trailing zero byte is written; instead, the length of the string is returned.

All conversion specifiers begin with a percent sign (to include a literal percent in the output, use "%%"), followed by:
- an optional alignment code
- an optional width code (required if there is an alignment code)
- an optional length code
- a conversion code.

The alignment code consists of two characters. The first is:
- "-": left align (fill on the right)
- "=": centre align (fill at both ends)
- "+": right align (fill on the left).
The second is either the character to fill with, or "*". In the latter case, the next word is taken from the arguments and used as the fill character. The default alignment is to right align filling with spaces ("+ "). Also, the alignment "+0" may be abbreviated to "0".

The width code gives the number of characters generated by the conversion. If the output would be longer, it is truncated. The code is either an unsigned decimal number (not beginning with a zero), or "*". In the latter case, the

next word is taken from the arguments and gives the (unsigned) width.

The length code can be "l" or "L" (equivalent); it is equivalent to making the conversion code uppercase.

Each conversion code (apart from "f") takes an argument of the type stated, and then converts it as described.

Code Type Conversion

"b" word convert to unsigned binary representation
"B" long convert to unsigned binary representation
"c" word output the character with that code
"C" long output the character with that code
"d" word convert to signed decimal representation
"D" long convert to signed decimal representation
"f" ---- output an empty string filled with the fill character.
"F" ---- output an empty string filled with the fill character.
"o" word convert to unsigned octal representation
"O" long convert to unsigned octal representation
"m" word output the value as 2 bytes, least significant first
"M" long output the value as 4 bytes, least significant first
"s" cstr output the cstr
"S" cstr output the cstr
"u" word convert to unsigned decimal representation
"U" long convert to unsigned decimal representation
"w" word output the value as 2 bytes, most significant first
"W" long output the value as 4 bytes, most significant first
"x" word convert to unsigned hexadecimal representation
"X" long convert to unsigned hexadecimal representation
Codes "m", "M", "w", and "W" are available in EPOC v2.17 and later only.

Fn $8A Sub $05
ConvStringToUnsignedInt fails
AX: -> converted value
CX: radix
SI: (cstr) string to convert -> pointer to first unused character
The string is converted to an unsigned integer in the specified radix. The conversion ends at the first character which is not a valid digit for the radix, which may be the terminating zero byte; if the first character of the string is not valid, the call fails. The letters A to F, in either case, are used for digits 10 to 15 if the radix is 11 or more. The radix may be greater

than 16, but the valid digits remain restricted to the range 0 to 15.

Fn $8A Sub $06
ConvStringToUnsignedLongInt fails
AX: -> high half of converted value
BX: -> low half of converted value
CX: radix
SI: (cstr) string to convert -> pointer to first unused character
The string is converted, in the same manner as ConvStringToUnsignedInt.

Fn $8A Sub $07
ConvStringToInt fails
AX: -> converted value
SI: (cstr) string to convert -> pointer to first unused character
The string is converted to an signed integer in radix 10. The conversion ends at the first character which is not a valid digit or leading sign; this may be the terminating zero byte. If the first character of the string is not a digit or sign, the call fails.

Fn $8A Sub $08
ConvStringToLongInt
AX: -> high half of converted value
BX: -> low half of converted value
SI: (cstr) string to convert -> pointer to first unused character
The string is converted, in the same manner as ConvStringToInt.

Fn $8A Sub $09
ConvFloatToBuffer fails
AX: -> length of converted value
DX: pointer to 6 byte conversion information block
SI: address of real value to be converted
DI: buffer to hold converted value
The real value is converted to a cstr and placed in the buffer. The exact format of the string depends on the contents of the conversion information block:
Offset 0 (byte): 0 = fixed format, 1 = exponent format, 2 = variable format
Offset 1 (byte): maximum length of converted value
Offset 2 (byte): number of decimal digits (fixed and exponent formats only)
Offset 3 (byte): radix character
Offset 4 (byte): triad character (fixed format only)

Offset 5 (byte): triad threshold (fixed format only)
If the converted value would be greater than the maximum length, or the real to be converted has an exponent less than -99 or greater than 99, the call fails and the contents of the buffer are undefined (note that the length of the buffer need only be the maximum specified plus an extra byte for the terminating zero).

For fixed format, the resulting string will take the form:
* minus sign if the value is negative; nothing if it is positive
* the integer part of the number, with no leading zeros (one zero if the integer part is zero); if the triad threshold is non-zero and there are more than that number of digits in the integer part, then the triad character is inserted between groups of 3 digits
* if the number of decimal digits is non-zero:
- the radix character
- the specified number of digits
For example, the number 1234321.4731 is converted as follows (assuming that the radix character is "." and the triad character is ","):
0 decimal digits, triad threshold 0: "1234321"
0 decimal digits, triad threshold 5: "1,234,321"
0 decimal digits, triad threshold 7: "1234321"
2 decimal digits, triad threshold 5: "1,234,321.47"
6 decimal digits, triad threshold 1: "1,234,321.473100"

For floating format, the resulting string will take the form:
* minus sign if the value is negative; nothing if it is positive
* one non-zero digit
* if the number of decimal digits is non-zero:
- the radix character
- the specified number of digits
* the letter "E"
* a plus or minus sign
* two digits (using leading zeros if necessary)
For example, the number 1234321.4731 is converted as follows:
0 decimal digits: "1E+06"
2 decimal digits: "1.23E+06"
7 decimal digits: "1.234321E+06"
12 decimal digits: "1.234321473100E+06"
Zero (with either sign) is always converted as "0E+00" or "0.000E+00" etc.

For general format the resulting string will take a form depending on its value. If the maximum width is W and the real value is R, then:
R <= -1E+(W-1) as for exponent format with W-7 decimal digits

-1E+(W-1) < R <= -1E+(W-2) a string containing a minus sign and W-1 digits
-1E+(W-2) < E <= -1E+(W-1) a string containing a minus sign and W-2 digits
-1E+(W-3) < R <= -1E-4 a string containing a minus sign and then W-2
significant digits with the radix character in
the appropriate position
-1E-4 < R < 0 as for exponent format with W-7 decimal digits
0 = R the string "0"
0 < R < 1E-4 as for exponent format with W-6 decimal digits
1E-4 <= R < 1E+(W-2) a string containing W-1 significant digits with
the radix character in the appropriate position
1E+(W-2) <= E < 1E+(W-1) a string containing W-1 digits
1E+(W-1) <= E < 1E+(W) a string containing W digits
1E+(W) <= R as for exponent format with W-7 decimal digits

Fn $8A Sub $0A
ConvStringToFloat fails
DX: radix character
SI: address of a word pointing to the start of the cstr to convert
DI: address of real variable to be set
Converts a string representing a floating-point value and places it in the
variable. The string must have the form:
* an optional sign
* a mantissa containing at least one digit and an optional radix character
* an optional exponent consisting of:
- the letter "E" or "e"
- an optional sign
- an integer
The resulting value must have an exponent in the range -99 to 99 inclusive.
If it is outside this range, the call fails; for underflow the value 0 is
stored, while for overflow an undefined value is stored.

If the call succeeds, the pointer to the string is altered to point to the
first character beyond the converted number.

Fn $8B Sub $00
GenVersion
AX: -> kernel version
Gets the version of the operating system. Version 1.23 is reported as:
$123A for alpha release
$123B for beta release
$123F for final release

Fn $8B Sub $01
GenLcdType
AL: -> display type
Gets the display type:
0 = 640x400 LCD (MC)
1 = 640x200 LCD small version (MC)
2 = 640x200 LCD large version or CGA (PC emulation)
3 = 720x348 LCD or Hercules graphics
4 = 160x80 LCD (HC)
5 = 240x80 LCD (Series 3t)
6 = MDA (PC emulation)
7 = EGA monochrome (PC emulation)
8 = EGA colour (PC emulation)
9 = VGA monochrome (PC emulation)
10 = VGA colour (PC emulation)
11 = 480x160 LCD (Series 3a and Series 3c)
12 = 240x100 LCD (Workabout)
14 = 240x160 LCD (Siena)
255 = unknown

Fn $8B Sub $02
GenStartReason
AL: -> reason code
Gets the reason for the last cold start:
0 = system RAM invalid
1 = forced power down
2 = user reset (using reset button)
3 = kernel fault
4 = new operating system installed
The environment variables and the internal disc are valid after reasons 1 and 3, and are valid after reason 2 unless ESC was also pressed.

Reason 1 only happens if a faulty device driver delayed a normal battery-low powerdown for too long.

Fn $8B Sub $03
GenParse fails
BX: 18 byte buffer
Parse filenames according to certain basic rules. Unlike FilParse, this does not invoke any device drivers. The buffer has the format:
Offset 0 (word): address of file name 1 (a cstr)
Offset 2 (word): address of file name 2 (a cstr)
Offset 4 (word): address of file name 3 (a cstr)
Offset 6 (word): address of buffer to be filled with final name (a cstr)
Offset 8 (word): address of 6 byte buffer
Offset 10 (byte): device separator
Offset 11 (byte): path separator
Offset 12 (byte): extension separator
Offset 13 (byte): maximum length of device including separators
Offset 14 (byte): maximum length of path including separators
Offset 15 (byte): maximum length of name
Offset 16 (byte): maximum length of extension including separator
The three filenames are split into 5 components, any of which may be missing.
* node (ends with "::")
* device (ends with device separator)
* path (ends with last path separator)
* name
* extension (starts with extension separator)
The final name is constructed by taking, for each component, the value from file name 1 if present, otherwise the value from file name 2 if present, and otherwise the value from file name 3. The 6 byte buffer is then filled in with the same data as for FilParse. File names 2 and 3 should be in different places in memory.

Fn $8B Sub $04
LongUnsignedIntRandom
AX: -> high half of random number
BX: address of seed -> low half of random number
Generates a 32 bit unsigned random number from a seed; the number also replaces the seed, so that the sequential calls with the same seed address will generate a random sequence based on the initial seed.

Fn $8B Sub $05
GenGetCountryData
BX: 40 byte buffer
Fills the buffer with country-specific data:
Offset 0 (word): country code (e.g. UK is 44) of locale
Offset 2 (word): current offset from GMT in minutes (+ is ahead)
Offset 4 (byte): date format (0 = MDY, 1 = DMY, 2 = YMD)
Offset 5 (byte): time format (0 = am-pm, 1 = 24 hour)
Offset 6 (byte): currency symbol position (0 = before, 1 = after)
Offset 7 (byte): currency symbol separated with space (0 = yes, 1 = no)
Offset 8 (byte): currency decimal places
Offset 9 (byte): currency negative format (0 = minus, 1 = brackets)
Offset 10 (byte): currency triad threshold
Offset 11 (byte): triad separator
Offset 12 (byte): decimal separator
Offset 13 (byte): date separator
Offset 14 (byte): time separator
Offset 15 to 23: currency symbol (cstr)
Offset 24 (byte): start of week (0 = Mon, 1 = Tue, ... 6 = Sun)
Offset 25 (byte): currently active summer times:
Bit 0: home
Bit 1: European
Bit 2: Northern
Bit 3: Southern
Bits 4 to 7: unused, always zero
Offset 26 (byte): clock type (0 = analogue, 1 = digital)
Offset 27 (byte): number of letters in day abbreviation (0 to 6)
Offset 28 (byte): number of letters in month abbreviation (0 to 255)
Offset 29 (byte): workdays (the set bits indicate the workdays)
Bit 0: Monday
Bit 1: Tuesday
Bit 2: Wednesday
Bit 3: Thursday
Bit 4: Friday
Bit 5: Saturday
Bit 6: Sunday
Bit 7: always zero
Offset 30 (byte): units (0 = inches, 1 = centimetres)

If the triad threshold is non-zero and there are more than that number of digits in the integer part of an amount of money, then the triad character should be inserted between groups of 3 digits. See ConvFloatToBuffer for a way to do this.

Fn $8B Sub $06
GenGetErrorText
AL: error number
BX: 64 byte buffer
The buffer will be filled with a cstr giving an error message corresponding to the error number (a generic message including the number is used when there is no stored message in the current locale).

Fn $8B Sub $07
GenGetOsData
CX: number of bytes to fetch
SI: offset of first byte in kernel workspace
DI: buffer to be filled in
Copy a number of bytes from the kernel workspace to the current process. Within assembler code, this can also be done via GenDataSegment (see Psionics file KERNEL).

Fn $8B Sub $08 only applies to MC systems.

Fn $8B Sub $09
GenNotify fails
AL: -> option chosen (0 = first, 1 = second, 2 = third)
BX: first message (cstr, up to 63 characters plus the terminating zero)
CX: second message (cstr, up to 63 characters plus the terminating zero)
DX: first option (cstr, up to 15 characters plus the terminating zero)
SI: second option (cstr, up to 15 characters plus the terminating zero)
DI: third option (cstr, up to 15 characters plus the terminating zero)
Sends a message to the notifier process and waits for a reply. The call fails if there is no notifier process running. The two messages are displayed, and the user is offered the three options. The chosen option is returned.

Any of the arguments except BX may be zero instead; for CX this means that only one message is displayed, while for DX, SI, and DI it means that less options are offered (if all are zero, the option "CONTINUE" is offered).

Fn $8B Sub $0A
GenNotifyError fails
AL: error number -> option chosen (0 = first, 1 = second, 2 = third)
BX: first message
DX: first option
SI: second option
DI: third option
This call is equivalent to GenNotify with the second message derived from the error number via GenGetErrorText.

Fn $8B Sub $0B and $0C are used by the notifier process.

Fn $8B Sub $0D
GenGetRamSizeInParas
AX: -> system RAM size
Gets the amount of system RAM fitted, in units of 16 bytes. Note that this is an unsigned value - $8000 represents 512 kb, not some negative amount.

Fn $8B Sub $0E
GenGetCommandLine
AX: -> zero or command line
Gets a pointer to the command line if the program was started with program information (see FilExecute). If so, the command line consists of a cstr giving the program executed, immediately followed by the program information (a qstr).

Fn $8B Sub $0F
GenGetSoundFlags
AX: -> sound flags
Gets the sound flags:
Bit 0: keyboard clicks enabled
Bit 1: sound via piezo buzzer enabled
Bit 2: sound via SND: device driver enabled
Bit 3: keyboard clicks: set=loud, clear=soft
Bit 4: piezo buzzer: set=loud, clear=soft
Bit 15: set=disable all sound, clear=obey individual flags

Fn $8B Sub $10
GenSetSoundFlags
BX: sound flags

Sets the sound flags to new values (see GenGetSoundFlags).

Fn $8B Sub $11
GenSound
BX: duration in ticks
CX: pitch code (frequency=512kHz/code)
Makes a simple single-frequency note. Access to this call is sequenced; it will wait until the piezo buzzer is not in use, and will return when the sound has started to play. On the Series 3a, the piezo buzzer is emulated by the SND: device, but the emulation is complete (for example, this call works even when SND: is disabled by GenSetSoundFlags).

Fn $8B Sub $12
GenMarkActive
Fn $8B Sub $13
GenMarkNonActive

These two calls alter the state of the current process to "active" (the default when the process starts) or "non-active". Whenever an active process restarts execution after being idle (basically when keywords such as IOWAIT, IOW, GET and so on return), the auto-off timer is reset; the machine switches off when the timer reaches the appropriate value.

Fn $8B Sub $14
GenGetText fails
AL: message number
eBX: 64 byte buffer
Copies a message (a cstr) from the kernel message table to the buffer.

Fn $8B Sub $15
GenGetNotifyState
AL: -> notify state (0=unattended, 1=notify)
Gets the notify state for the process. If this is "unattended", then the file server will not call the notifier process when a correctable error occurs (such as an SSD with open file being removed), but will simply return an error. If it is "notify" (the initial state), then it will use GenNotify to request the user to fix the problem.

Fn $8B Sub $16

GenSetNotifyState
AL: notify state
Sets the notify state for the process. See GenGetNotifyState for details.

Fn $8B Sub $17
GenGetAutoSwitchOffValue
AX: -> auto-off time in seconds, or $FFFF if disabled
Gets the auto-off time.

Fn $8B Sub $18
GenSetAutoSwitchOffValue
BX: auto-off time in seconds, or $FFFF to disable
Sets the auto-off time. Times of less than 15 are adjusted to 15.

Fn $8B Sub $19 and $1A are for device drivers only.

Fn $8B Sub $1B
GenGetLanguageCode
AX: -> current locale code
Gets the current locale code. Locale codes are listed in the Psionics file LOCALES.

Fn $8B Sub $1C
GenGetSuffixes
eBX: 93 byte buffer
The buffer is filled with 31 cstrs giving the correct suffix for each of the 31 days of the month. The suffix for day N is at offset (3*N-3).

Fn $8B Sub $1D
GenGetAmPmText
AL: 0=AM, 1=PM
eBX: 3 byte buffer
The buffer is filled with a cstr giving the correct suffix for "a.m." or "p.m." times.

Fn $8B Sub $1E
GenSetCountryData
eBX: 40 byte buffer

Sets the country-specific data to that in the buffer. See GenGetCountryData for the format of the buffer.

Fn $8B Sub $1F
GenGetBatteryType
AL: -> battery type
Gets the battery type from the following list:
0 = Unknown
1 = Alkaline
2 = NiCaD (600 mAh)
3 = NiCaD (1000 mAh)
4 = NiCaD (500 mAh)
The battery type is used to control the warning thresholds.

Fn $8B Sub $20
GenSetBatteryType
AL: battery type
Sets the battery type (see GenGetBatteryType).

Fn $8B Sub $21
GenEnvBufferGet fails
AX: -> size of value
DX: length of pattern
eSI: buffer filled with value
eDI: pattern to search for
Searches for an environment variable whose name matches the pattern (if there is more than one, which one is chosen is unspecified). The buffer is filled with the variable's value.

Fn $8B Sub $22
GenEnvBufferSet fails
CX: size of value (0 to 255)
DX: length of name (1 to 16)
eSI: buffer holding value
eDI: name of variable
Changes the value of the specified enviromment variable, creating it first if necessary. The name may not contain wildcards.

Fn $8B Sub $23

GenEnvBufferDelete fails
DX: length of name
eDI: pattern to delete
Searches for an environment variable whose name matches the pattern and deletes it (if there is more than one, which one is deleted is unspecified).

Fn $8B Sub $24
GenEnvBufferFind fails
AX: -> new handle
BX: previous handle (0 for first call)
DX: length of pattern
eSI: 273 byte buffer
eDI: pattern to search for
Searches for each environment variable whose name matches the pattern. The call is made multiple times until it fails; the same pattern must be used each time. If the call succeeds, the buffer is filled with two qstrs, the first being the name and the second the value of the variable found.

Fn $8B Sub $25
GenEnvStringGet fails
eSI: 256 byte buffer
eDI: pattern (cstr)
Searches for an environment variable whose name matches the pattern (if there is more than one, which one is chosen is unspecified). The buffer is filled with the variable's value followed by a zero byte (thus a cstr).

Fn $8B Sub $26
GenEnvStringSet fails
eSI: value (cstr)
eDI: name (cstr)
Changes the value of the specified enviromment variable, creating it first if necessary. The name may not contain wildcards and is limited to 16 characters; the value is limited to 255 characters.

Fn $8B Sub $27
GenEnvStringDelete fails
eDI: pattern (cstr)
Searches for an environment variable whose name matches the pattern and deletes it (if there is more than one, which one is deleted is unspecified).

Fn $8B Sub $28
GenEnvStringFind fails
AX: -> new handle
BX: previous handle (0 for first call)
eCX: 256 byte buffer
eSI: 17 byte buffer
eDI: pattern to search for
Searches for each environment variable whose name matches the pattern. The call is made multiple times until it fails; the same pattern must be used each time. If the call succeeds, the 17 byte buffer is filled with the name of the variable, as a cstr, and the 256 byte buffer is filled with the value followed by a zero byte (thus also a cstr).

Fn $8B Sub $29
GenCrc
AX: -> CRC
CX: buffer length
DX: previous CRC
SI: buffer
Calculates the CCITT Cyclic Redundancy Checksum using the X^16+X^12+X^5+1 polynomial (e.g. the CRC of a single byte with value 1 is $1021). If the buffer is to be checked standalone, use zero as the previous checksum. If several blocks are to be checksummed as if they were one long block, use 0 as the previous CRC for the first block, and the previous result for the remainder.

Fn $8B Sub $2A
GenRomVersion
AX: -> rom version number
Gets the version of the system ROM (which includes both the operating system and many files). Version 1.23 is reported as:
$123A for alpha release
$123B for beta release
$123F for final release

Fn $8B Sub $2B and $2C are used by the alarm server process.

Fn $8B Sub $2D
GenAlarmId
AX: -> alarm server pid

Gets the process ID of the alarm server process, or 0 if none is running.

Fn $8B Sub $2E
GenPasswordSet fails
uSI: (qstr) current password
uDI: (qstr) new password
Sets a new system password. The call will fail and take no action if the current password is incorrect.

Fn $8B Sub $2F
GenPasswordTest fails
uSI: (qstr) password to test
Succeeds if the system password is that provided.

Fn $8B Sub $30
GenPasswordControl fails
AL: new state (0 = off, 1 = on)
uSI: (qstr) current password
Turns the system password on and off. The call will fail and take no action if the current password is not provided.

Fn $8B Sub $31
GenPasswordQuery
AX: -> new state (0 = off, 1 = on)
Get the status of the system password.

Fn $8B Sub $32
GenTickle

Resets the auto-off timer. A non-active process (see GenMarkNonActive) can use this to prevent auto-off.

Fn $8B Sub $33
GenSetConfig
@No documentation available at present@

Fn $8B Sub $34
GenMaskInit fails

SI: (qstr) password
DI: 18 byte encryption control block
Initializes an encryption control block according to the password given.

The contents of the control block for the current algorithm are:
Offset 0 to 8: encryption key, generated from the password
Offset 9 to 15: copy of offset 0 to 7
Offset 16 (word): current location within the key (initially zero)
@The algorithm for generating the key from the password is unknown.

It is possible that future versions of this call may implement multiple encryption algorithms, so AL, BX, CX, and DX - unused at present - should be set to zero.

Fn $8B Sub $35
GenMaskEncrypt fails
CX: length of buffer
SI: encryption control block
DI: buffer holding data to be encrypted
Encrypts a block of data according to the encryption control block, which will be updated. The value of the encrypted data depend only on the original data and on the contents of the control block, which should normally be initialized by GenMaskInit. Provided that the control block is not modified in any other way, encrypting a block of data in two or more parts yields the same result as encrypting it all at once.

The current encryption algorithm operates on a byte by byte basis as follows. For each data byte in turn, take the value of offset 16 in the control block (which will be from 0 to 15) and use it to select one of the first 16 bytes in the control block (that is, offset 16 = 0 selects offset 0, offset 16 = 1 selects offset 1, and so on). Add the selected byte to the data byte modulo $100. Finally, add 1, modulo 16, to offset 16 of the control block.

An example of this algorithm in use can be found in the file WORD.FMT.

It is possible that future versions of this call may implement multiple encryption algorithms, so AL, BX, and DX - unused at present - should be set to zero.

Fn $8B Sub $36
GenMaskDecrypt fails
CX: length of buffer

SI: encryption control block
DI: buffer holding data to be decrypted
Decrypts a block of data according to the encryption control block, which will be updated. The data should have been encrypted with GenMaskEncrypt with the same initial control block. All comments applying to the latter call apply here.

Fn $8B Sub $37
GenSetOnEvents v2.28
AL: 0=disable 1=enable
Enables or disables reporting of power-on events. By default this is enabled on the Series 3 and disabled on other systems. Disabling may affect other applications.

Fn $8B Sub $38
GenGetAutoMains v3
AX: -> setting
Gets the setting of the external power auto-off flag. Zero means that auto-off will take place even if external power is present, while non-zero means that auto-off is suspended while external power is present.

Fn $8B Sub $39
GenSetAutoMains v3
AL: new setting
Sets the external power auto-off flag. Zero means that auto-off will take place even if external power is present (unless it has been disabled by setting the auto-off time to $FFFF), while non-zero means that auto-off is suspended while external power is present.

Fn $8C Sub $00
FloatSin fails [sine]
Fn $8C Sub $01
FloatCos fails [cosine]
Fn $8C Sub $02
FloatTan fails [tangent]
Fn $8C Sub $03
FloatASin fails [arc sine]
Fn $8C Sub $04
FloatACos fails [arc cosine]
Fn $8C Sub $05

FloatATan fails [arc tangent]
Fn $8C Sub $06
FloatExp fails [exponentiation (base e)]
Fn $8C Sub $07
FloatLn fails [natural logarithm (base e)]
Fn $8C Sub $08
FloatLog fails [decimal logarithm (base 10)]
Fn $8C Sub $09
FloatSqrt fails [square root]
SI: argument (real)
DI: result (real)
Calculates the specified function of the argument. The argument and result may be stored in the same place; if not, the argument is not altered. The trignometric functions operate in radians.

Fn $8C Sub $0A
FloatPow fails
DX: power (real)
SI: base (real)
DI: result (real)
Calculates the result of raising the base to the indicated power. The result may be stored in the same place as either argument; if not, neither argument is altered.

Fn $8C Sub $0B
FloatRand
SI: seed (long)
DI: result (real)
Generates a random real number based on the unsigned long integer seed, which is unaltered; the memory used for the two should not overlap.

Fn $8C Sub $0C
FloatMod fails
DX: divisor (real)
SI: dividend (real)
DI: result (real)
Calculates the dividend modulo the divisor, using the formula:
result = dividend - divisor * FloatInt (dividend/divisor)
The result may be stored in the same place as either argument; if not, neither argument is altered.

Fn $8C Sub $0D
FloatInt fails
SI: argument (real)
DI: result (real)
Rounds the argument to the closest integer towards zero (i.e. zeros the fractional part of the argument). The argument and result may be stored in the same place; if not, the argument is not altered.

PSIONICS FILE - SYSCALLS.3
==========
System calls (part 3)
Last modified 1998-07-22
==========

See part 1 for general notes and explanations.

Fn $8D is used for the Window Server, and is described in the Psionics file WSERVER.

Fn $8E Sub $00 to $10 control various parts of the hardware, and should not be used by OPL programs.

Fn $8E Sub $11
HwGetSupplyStatus
BX: 6 byte buffer
Gets information about the power supply. The buffer is filled with the following data:
Offset 0 (word): main battery voltage in mV
Offset 2 (word): backup battery voltage in mV
Offset 4 (word): positive if external power is available, zero if not available, and negative if the detector is disabled because an SSD door is open.
The returned voltages should be taken with a grain of salt: I have observed the following values:
1800: no main batteries
1800: low main batteries
2056: new main batteries
2300: no lithium battery
2556: lithium battery fitted

Fn $8E Sub $12
HwLcdContrastDelta
AL: step direction
Alters the LCD contrast by one step, upwards if AL is between 0 and 127, and downwards if it is between 128 and 255 (all inclusive).

Fn $8E Sub $13
HwReadLcdContrast
AL: -> setting
Gets the current LCD contrast setting. On a Series 3, only the bottom 4 bits are significant.

Fn $8E Sub $14
HwSwitchOff
CX: delay in quarter seconds
Switches the machine off for the specified time, then back on again ($FFFF means never turn back on). The machine will also be turned on by external events, alarms, and timers. The delay must be at least 9 (2.25 seconds).

Fn $8E Sub $15 should only be used by device drivers.

Fn $8E Sub $16
HwExit

Exits the emulation on PC systems; has no effect on actual Psion machines.

Fn $8E Sub $17 to $1A should only be used by device drivers.

Fn $8E Sub $1B
HwGetPsuType
AL: -> PSU type
Gets the PSU type: 0 = old MC, 1 = MC "Maxim", 2 = Series 3t, 3 = Series 3a

Fn $8E Sub $1C
HwSupplyWarnings
BX: 8 byte buffer
Gets information about the power supply. The buffer is filled with the following data:

Offset 0 (word): main battery good voltage threshold in mV
Offset 2 (word): backup battery good voltage threshold in mV
Offset 4 (word): main battery nominal maximum voltage in mV
Offset 6 (word): backup battery nominal maximum voltage in mV
The values will depend on the battery type set by GenSetBatteryType.

Fn $8E Sub $1D
HwForceSupplyReading
@No documentation available at present@

Fn $8E Sub $1E
HwGetBackLight
AX: -> current value
On systems fitted with a backlight, this value indicates control of the backlight function. If the top bit is set, the operating system will ignore the backlight toggle key. The remaining bits give the auto-light-off time in ticks (1/32 second); zero means that auto-light-off is disabled.

Fn $8E Sub $1F
HwSetBackLight
BX: new value
Sets the backlight control value (see HwGetBackLight).

Fn $8E Sub $20
HwBackLight fails
AL: action -> status before call
This call fails if no backlight is fitted. Otherwise, the actions are:
0 = switch backlight off
1 = switch backlight on
2 = toggle backlight
3 = no action
The status returned is that of the backlight before the call: 0 = off, 1 = on.

Fn $8E Sub $21 controls various parts of the hardware, and should not be used by OPL programs.

Fn $8E Sub $22
HwSupplyInfo v3
BX: 22 byte buffer

Fills the buffer with additional information about the power supply:
Offset 0 (byte): main battery status:
0 = not present
1 = very low voltage
2 = low voltage
3 = good voltage
Offset 1 (byte): worst main battery status since batteries last inserted
Offset 2 (byte): non-zero if backup battery voltage is good
Offset 3 (byte): non-zero if external supply is present
Offset 4 (word): warning flags
Bit 0: set if power supply too low for sound
Bit 1: set if power supply too low to use flash
Bit 2: set if offset 6 changed because system clock changed
clear if offset 6 changed because the batteries were changed
Offset 6 (long): abstime when batteries inserted
Offset 10 (long): ticks running on battery
Offset 14 (long): ticks running on external supply
Offset 18 (long): mA-ticks (i.e. mAhours * 60 * 60 * 32)

Fn $8E Sub $28
HwGetScanCodes v3
BX: 20 byte buffer
The buffer is filled with information describing the state of each key on the keyboard. For the Series 3a, the offset and bit for each key are given in the following table ("2:4" means offset 2, bit 4).
System 9:1 Esc 15:0 Delete 4:0 1 14:1 A 12:2 N 0:6
Data 7:1 Tab 0:2 Enter 0:0 2 14:2 B 8:6 O 4:5
Word 11:1 Control 4:7 ShiftR 6:7 3 10:6 C 10:3 P 2:5
Agenda 3:1 ShiftL 2:7 Up 14:5 4 8:2 D 10:4 Q 12:1
Time 1:1 Psion 0:7 Left 0:4 5 8:3 E 10:5 R 8:1
World 5:1 Menu 10:7 Down 0:5 6 14:3 F 10:1 S 12:4
Calc 3:0 Diamond 8:7 Right 0:1 7 6:6 G 8:5 T 8:4
Sheet 1:0 Space 8:0 Help 6:2 8 4:3 H 14:6 U 6:5
+ and = 2:3 - and _ 2:2 9 4:4 I 4:2 V 10:2
* and : 2:6 / and ; 2:1 0 2:4 J 6:4 W 12:5
, and < 6:1 . and > 14:4 K 4:1 X 12:6
L 4:6 Y 0:3
M 6:3 Z 12:3

Fn $8E Sub $29
HwGetSsdData

Nothing is known about this call except its name. @@@@@@

Fn $8E Sub $2A
HwResetBatteryStatus v3.9

The battery status is reset, exactly as if the main batteries had been removed and then replaced.

Fn $8E Sub $2B
HwEnableAutoBatReset v3.9
AX: -> (if querying) 1 = enabled, 0 = disabled
BX: 1 = enable, 0 = disable, -1 = query
Enables, disables, or queries the current setting of the automatic reset of the battery status. This is intended for use on the Workabout, where the batteries can be recharged in situ.

Fn $8E Sub $2C
HwGetBatData v3.9
AX: -> address of a data structure in kernel data space
The data is described in the KERNEL file.

Fn $8E Sub $2E
HwReLogPacks v3.9 fails

This has the same effect as opening and then closing the SSD doors.

Fn $8E Sub $2F
HwSetIRPowerLevel v3.9
AX: -> previous level (0 = low, 1 = high)
BX: new level (0 = low, 1 = high)
Sets the IR power level, returning the previous level.

Fn $8E Sub $30
HwReturnTickCount v3.9
AX: tick count

Returns a tick count that is incremented 32 times per second.

Fn $8E Sub $31
HwReturnExpansionPortState v3.9
AX: -> port type and state
BX: -> zero
Returns the type and state of the expansion port:
Bits 0 to 7: non-zero if SSD doors are open, or something has just been plugged into or removed from the Honda expansion connector
Bits 8 to 14: connector type: 0 = Series 3t/3a, 1 = Workabout LIF, 2 = Siena Honda, 3 = Series 3c Honda, 4 = HC
Bit 15: 1 = contains Condor chip, 0 = no Condor chip

Fn $8F
GenDataSegment

This call is only useful in assembler code; it sets ES to point to the start of the kernel data space (accessible in OPL using GenGetOsData).

Fn $90
ProcPanic
AL: reason
Panic the current process; this call never returns.

Fn $91
ProcCopyFromById fails
BX: process ID
CX: number of bytes to copy
SI: remote address of first byte to copy
eDI: local address of first byte to copy
Copy a number of bytes from the indicated process to the current process.

Fn $92
ProcCopyToById fails
BX: process ID
CX: number of bytes to copy
SI: local address of first byte to copy
DI: remote address of first byte to copy
Copies a number of bytes from the current process to the indicated process.

Fn $93
CharIsDigit
Fn $94
CharIsHexDigit
Fn $95
CharIsPrintable
Fn $96
CharIsAlphabetic
Fn $97
CharIsAlphaNumeric
Fn $98
CharIsUpperCase
Fn $99
CharIsLowerCase
Fn $9A
CharIsSpace
Fn $9B
CharIsPunctuation
Fn $9C
CharIsGraphic
Fn $9D
CharIsControl
AL: character to test
EQ: -> set if test fails, clear if test succeeds
Tests to see whether the character has the indicated property. These functions are language dependent.

Fn $9E
CharToUpperChar
Fn $9F
CharToLowerChar
Fn $A0

CharToFoldedChar
AL: character 1 -> converted character 1
AH: character 2 -> converted character 2
Converts two characters to uppercase, lowercase, or folded (uppercase with no accents). These functions are language dependent.

Fn $A1
BufferCopy
CX: length to be copied
SI: address of "from" buffer
eDI: address of "to" buffer
Copies a number of bytes from one buffer to another. The case of the buffers overlapping is handled correctly.

Fn $A2
BufferSwap
CX: length to be swapped
SI: address of buffer 1
eDI: address of buffer 2
Swaps the contents of two buffers. The case of the buffers overlapping is handled correctly.

Fn $A3
BufferCompare
Fn $A4
BufferCompareFolded
BX: length of buffer 2
CX: length of buffer 1
SI: address of buffer 1
eDI: address of buffer 2
UL: -> set if buffer 1 less than buffer 2
EQ: -> set if buffers are identical
The contents of the two buffers are compared byte-for-byte, using unsigned comparisons, and the result flags set accordingly. BufferCompareFolded acts as if each character had been passed to CharToFoldedChar before comparison.

Fn $A5
BufferMatch fails
Fn $A6
BufferMatchFolded fails

CX: length of buffer
DX: length of pattern
SI: address of buffer
eDI: address of pattern
The buffer is examined to determine whether it matches the pattern (using the usual wildcards); the call fails if it does not. BufferMatchFolded acts as if each character had been passed to CharToFoldedChar before comparison.

Fn $A7
BufferLocate fails
Fn $A8
BufferLocateFolded fails
AH: character -> [undefined]
AX: -> offset of character
CX: length of buffer
SI: address of buffer
The buffer is searched to determine if the character occurs within it; the call fails if it does not. If it does, the offset from the start of the buffer of the first occurence is returned. BufferLocateFolded acts as if each character had been passed to CharToFoldedChar before comparison.

Fn $A9
BufferSubBuffer fails
Fn $AA
BufferSubBufferFolded fails
AX: -> offset
BX: length of buffer 2
CX: length of buffer 1
SI: address of buffer 1
eDI: address of buffer 2
Buffer 1 is searched to determine if buffer 2 occurs within it; the call fails if it does not. If it does, the offset from the start of buffer 1 of the first occurence of buffer 2 is returned. BufferSubBufferFolded acts as if each character had been passed to CharToFoldedChar before comparison.

Fn $AB
BufferJustify
AX: -> address of first uncopied character
BX: length of buffer 2
CX: length of buffer 1
DX: fill character * 256 + control code

SI: address of buffer 1
eDI: address of buffer 2
Buffer 1 is copied into buffer 2, and the remaining space filled with the fill character according to the control code (0 = fill at end, 1 = fill at start, 2 = centre the copied data). If buffer 1 is longer than buffer 2, then the contents will be truncated to fit. If the length of buffer 2 is negative, then it is assumed to be the same as that of buffer 1 (and the data is just copied).

Fn $AC
StringCopy
Fn $AD
StringCopyFolded
SI: cstr
eDI: large enough buffer
The cstr is copied into the buffer. StringCopyFolded passes each character through CharToFoldedChar during the copy.

Fn $AE
StringConvertToFolded
SI: cstr
Each character of the cstr is passed through CharToFoldedChar.

Fn $AF
StringCompare
Fn $B0
StringCompareFolded
SI: cstr 1
eDI: cstr 2
UL: -> set if cstr 1 less than cstr 2
EQ: -> set if cstrs are identical
The contents of the two cstrs are compared byte-for-byte, using unsigned comparisons, and the result flags set accordingly. StringCompareFolded acts as if each character had been passed to CharToFoldedChar before comparison.

Fn $B1
StringMatch fails
Fn $B2
StringMatchFolded fails
SI: cstr to be searched
eDI: pattern (cstr)

The cstr is searched to determine if the pattern occurs within it (using the usual wildcards); the call fails if it does not. StringMatchFolded acts as if each character had been passed to CharToFoldedChar before comparison.

Fn $B3
StringLocate fails
Fn $B4
StringLocateFolded fails
Fn $B5
StringLocateInReverse fails
Fn $B6
StringLocateInReverseFolded fails
AH: character -> [undefined]
AX: -> offset of character
SI: cstr
The cstr is searched to determine if the character occurs within it; the call fails if it does not. If it does, the offset from the start of the cstr of the first (for StringLocate and StringLocateFolded) or last (for the two Reverse calls) occurence is returned. The two Folded calls act as if each character had been passed to CharToFoldedChar before comparison.

Fn $B7
StringSubString fails
Fn $B8
StringSubStringFolded fails
AX: -> offset
SI: cstr 1
eDI: cstr 2
Cstr 1 is searched to determine if cstr 2 occurs within it; the call fails if it does not. If it does, the offset from the start of cstr 1 of the first occurence of cstr 2 is returned. StringSubStringFolded acts as if each character had been passed to CharToFoldedChar before comparison.

Fn $B9
StringLength
AX: -> length of the cstr
eDI: cstr
Returns the length of a cstr, excluding the terminating zero byte.

Fn $BA

StringValidateName fails
AL: maximum number of characters permitted
AH: non-zero if an extension is permitted
eDI: cstr
The cstr is checked to see if it a valid name, and the call fails if it is not. A name is valid if it:
* begins with a letter;
* contains only letters, digits, dollar signs ($), underscores (_), and at most one dot;
* if AH is zero, does not contain a dot;
* contains no more than the specified number of characters before the dot (if any); and
* contains no more than 4 characters after the dot, if any.
@The manual says 3 characters, but my tests accept 4.@

Fn $BB
LongIntCompare
AX: high half of P
BX: low half of P
CX: high half of Q
DX: low half of Q
SL: -> set if P less than Q
EQ: -> set if P equals Q
Compares P and Q (both signed longs) and sets the result flags accordingly.

Fn $BC
LongIntMultiply fails
AX: high half of P -> high half of product
BX: low half of P -> low half of product
CX: high half of Q
DX: low half of Q
Multiples P and Q (both signed longs); the call fails if the result cannot be represented as a signed long.

Fn $BD
LongIntDivide fails
AX: high half of P -> high half of quotient
BX: low half of P -> low half of quotient
CX: high half of Q -> high half of remainder
DX: low half of Q -> low half of remainder
Divides P by Q (both signed longs); the call fails if Q is zero. The remainder

will have the same sign as P.

Fn $BE
LongUnsignedIntCompare
AX: high half of P
BX: low half of P
CX: high half of Q
DX: low half of Q
UL: -> set if P less than Q
EQ: -> set if P equals Q
Compares P and Q (both unsigned longs) and sets the result flags accordingly.

Fn $BF
LongUnsignedIntMultiply fails
AX: high half of P -> high half of product
BX: low half of P -> low half of product
CX: high half of Q
DX: low half of Q
Multiples P and Q (both unsigned longs); the call fails if the result cannot
be represented as a unsigned long.

Fn $C0
LongUnsignedIntDivide fails
AX: high half of P -> high half of quotient
BX: low half of P -> low half of quotient
CX: high half of Q -> high half of remainder
DX: low half of Q -> low half of remainder
Divides P by Q (both unsigned longs); the call fails if Q is zero.

Fn $C1
FloatAdd fails
Fn $C2
FloatSubtract fails
Fn $C3
FloatMultiply fails
Fn $C4
FloatDivide fails
SI: address of Q
DI: address of P
Calculates the specified one of P+Q, P-Q, P*Q, or P/Q, and places the result

in P. Both P and Q are reals.

Fn $C5
FloatCompare
SI: address of Q
DI: address of P
SL: -> set if P is less than Q
EQ: -> set if P equals Q
Compares two reals and sets the result flags appropriately.

Fn $C6
FloatNegate
DI: address of P
Negates a real in-situ.

Fn $C7
FloatToInt fails
Fn $C8
FloatToUnsignedInt fails
AX: -> result
SI: address of real
Converts a real to a signed or unsigned int; the call fails if the result is out of range. FloatToUnsignedInt ignores the sign of the real.

Fn $C9
FloatToLong fails
Fn $CA
FloatToUnsignedLong fails
AX: -> high half of result
BX: -> low half of result
SI: address of real
Converts a real to a signed or unsigned long; the call fails if the result is out of range. FloatToUnsignedLong ignores the sign of the real.

Fn $CB
IntToFloat
Fn $CC
UnsignedIntToFloat
AX: value

DI: address of real
Converts a signed or unsigned int to a real.

Fn $CD
LongToFloat
Fn $CE
UnsignedLongToFloat
AX: high half of value
BX: low half of value
DI: address of real
Converts a signed or unsigned long to a real.

Fn $CF
LibSend
Fn $D0
LibSendSuper
AX: -> method result
BX: handle of object to receive the message
CX: message number
DX: argument 1
SI: argument 2
DI: argument 3
These functions send a message to an object and invoke a method on that object.
LibSend starts searching for the method in the class of the object, and
LibSendSuper in the superclass.

Fn $D1 to $D3 can only be used from assembler.

Fn $D4
Dummy

This function has no effect.

Fn $D5
GenIntByNumber
AL: Fn -> error and result flags
SI: argument block
DI: result block
This call is equivalent to the OS keyword; it calls another system call with

the arguments and results stored in 12 byte blocks.

Fn $D6 is used for the Window Server, and is described in the Psionics file WSERVER.

Fn $D7 is normally only used by the C library.

Fn $D8 is used for accessing DBF files. See the Psionics file DBF.FMT.

Fn $D9
LibEnterSend
AX: -> method result
BX: handle of object to receive the message
CX: message number
DX: argument 1
SI: argument 2
DI: argument 3
This is identical to LibSend, except that it also starts a new "entry-exit region". The method description will state when this is needed.

Fn $DA
IoKeyAndMouseStatus
@No documentation available at present@

Fn $DB
StringCapitalise
SI: cstr
The first character of the cstr is passed through CharToUpperChar, and the remaining characters through CharToLowerChar.

Fn $DC
ProcIndStringCopyFromById fails
BX: process ID
CX: maximum length to copy
SI: address of location holding address of cstr
eDI: buffer
This copies a cstr from the indicated process to the current process. The cstr is found via a pointer also in the memory of the indicated process, and

the address of this pointer is specified. This call is equivalent to making two calls to ProcCopyFromById, the first to find the location of the cstr, and the second to fetch it, except that the contents of the buffer beyond the terminating zero are unspecified.

Fn $DD is reserved for the window server.

Fn $DE
IoSerManager
@No documentation available at present@

Revision #6
Created Wed, Jan 23, 2019 9:05 PM by Alex
Updated Sun, Jun 2, 2019 3:43 PM by Alex