The Neki32 Software Development Manual

Nekisoft Pty Ltd

November 15, 2024

Contents

Preface

Numbers are in decimal (base-10) unless otherwise stated.
Numbers beginning with a `0x` prefix are hexadecimal (base-16).

Australian Company Number 680 583 251

1 Introduction

Thank you for your interest in developing software for Neki32!

1.1 What is Neki32?

Neki32 is a 32-bit game console that delivers a focused experience to both gamers and game developers. The system runs on an ARM9 CPU clocked at 300MHz. Up to 24MBytes of memory is available for the game process. Software-rendering is used to produce bitmapped graphics and digital sound.

An operating system hides details of the hardware. Its custom kernel is called "PVMK", the Puny Video Machine Kernel. The system interface is kept to a minimum, to ensure plug-and-play compatibility.

The Neki32 console itself is powered by USB-C. It outputs audio and video to an HDMI TV. Games are distributed on read-only SD cards. There are four controller ports, compatible with Genesis and Mega-Drive controllers. There is 4MBytes of internal memory for savegames of 128KBytes per game.

(A real, live Neki32)

1.2 Technical Specifications

- Power input:
  - Connector: USB-C compatible[1]
  - Voltage: $5V \pm 10\%$
  - Current: $\leq 100mA$
- A/V output:
  - Connector: HDMI compatible[2]
  - Resolution: 640x480, or 320x240 with pixel-doubling
  - Refresh rate: 60Hz
  - Aspect ratio: 4:3
  - Color depth: 16 bits per pixel, RGB565
  - Audio format: 48KHz 16-bit stereo LPCM
- User inputs:
  - Connector: 9-pin D-Sub (male) x4
  - Layout: 8-way directional pad, 6 face buttons, 2 menu buttons
  - Protocol: Mega-Drive compatible[3]
- Game media:
  - Connector: SD/MMC card (full-size)
  - Capacity: 16MBytes to 2TBytes
  - Format: El-Torito bootable image, platform `0x92`

[1]The Neki32 console is not USB™-certified but should work with any USB-C power supply.
[2]It is is also not HDMI™-certified but should work with any HDMI television.
[3]Mega-Drive™ is owned by Sega and used without permission. They are unaffiliated with us.

1.3 Overview of a Game

A Neki32 game is quite simple. It is an SD card containing a program to run and possibly other data. Very little is necessary to get code running.

1.3.1 Game cards

The card is read-only and formatted much like an optical disc. It contains an El Torito boot record for platform `0x92`. The El Torito boot record points at the executable file for the game.

1.3.2 Executable and environment

The game executable is a flat, 0-mapped binary image of the game's initial memory content. For example, if the game executable is 3MBytes in size, the system will load it into virtual memory addresses `0` to `3145727`. The game process is initially 24 megabytes in size, and the rest will be filled with `0x00` bytes. The zero page, afterward, is inaccessible. An access of address `0` to `4095` always faults. On disk, a small magic number is placed at the start of the executable instead (the 8-byte string constant `NNEARM32` and the 8-byte little-endian address `0x1000`).

The game starts execution at address `4096`, the first accessible address. All registers are zeroed on entry. The process starts in ARM mode but may switch to Thumb mode at its choice.

System-calls can be made using a `udf 0x92` instruction. The call number is placed in `r0`, while parameters are placed in `r1` through `r5`. Return values are left in `r0` after the call is performed. System-calls are nonblocking and can be retried until completion or interleaved with other operations.

Exceptions are delivered to the game process as a signal. Signals are initially blocked. An exception causing a blocked signal will terminate the process. A signal can be unblocked and handled instead. In this case, it causes the process to restart to address `4096` with the signal number in `r0`. A system-call is used to return from the signal.

1.3.3 Hardware features

Games can make system-calls to access hardware features.

Images can be displayed from a linear framebuffer, at least 4-byte-aligned, anywhere in the process address space. The front-buffer can be changed only at the start of vertical blanking. The kernel will set aside the last requested address and make the flip when vblank starts. The video scanout loads 16-bit RGB565 pixel data

directly from process memory. Framebuffers should avoid crossing 1MByte address boundaries, for best performance.

Sound can be played using PCM data, at least 4-byte-aligned, anywhere in the process address space. The kernel sets aside a copy of the data and plays it back using DMA. Only 16-bit 48KHz left-then-right stereo is supported.

Input can be read using a system-call to a buffer, at least 4-byte-aligned, anywhere in the process address space. Input is retrieved from up to 4 Megadrive-compatible gamepads plugged into the system. The ports are labelled "A", "B", "C", and "D". Input is delivered as a stream of events tagged with these characters as the first byte.

A feature like `exec()` is available for replacing the running program with another. The kernel keeps a separate address space ready for a process to load, and then move into place. Arguments and environment variables can be preserved in a kernel-side buffer during this process.

Nonvolatile memory is available inside the console for savegames. 4MBytes are dedicated to the NVM savegame system, split into records of 128KBytes. Games are allocated a NVM record, on boot, if their card has a valid Volume ID that doesn't start with `'-'`. Saving and loading an NVM record are both atomic operations. The contents of the NVM record is replaced entirely on a successful save. Interrupted saves do not corrupt the existing record.

2 The boot process

The boot process treats the game card like an optical disc - it is read-only and organized in naturally-aligned 2048-byte sectors. Sector `0` contains bytes `0` to `2047`, sector `1` contains bytes `2048` to `4095`, and so on.

2.1 The beginning of the media

The system first reads sectors `0x10` and `0x11`. Sector `0x10` always contains the ISO9660 Primary Volume Descriptor, describing the game media. The ISO9660 Primary Volume Descriptor gives the name of the media, in a `32`-byte field at offset `40`. If this is present, not entirely whitespace, and not `CDROM` or `cdrom`, it is taken to be the game's name.

Savegame memory will be allocated and named automatically according to the name given in the Volume ID of the PVD. (If the name is not present, or starts with the character `-` (`0x2D`), no savegame memory will be allocated.)

Sector `0x11` always contains the El Torito Boot Volume Descriptor, describing the boot information. The El Torito Boot Volume Descriptor, from sector `0x11`, is

used to locate an El Torito Boot Catalog. The El Torito Boot Volume Descriptor must contain the magic value `EL TORITO SPECIFICATION` at offset `7`. It also contains the sector number of the El Torito Boot Catalog, as a 32-bit little-endian number at offset `0x47`. This is taken to be in units of `2048` bytes, from the beginning of the game media.

2.2 The boot catalog

The system reads the El Torito Boot Catalog from the sector number given in the El Torito Boot Volume Descriptor. Only one sector is read, even though El Torito allows for larger boot catalogs. The Boot Catalog consists of `32`-byte entries, naturally-aligned. The first entry must start with the following one-byte magic values:

Offset	Value
`0x00`	`0x01`
`0x1E`	`0x55`
`0x1F`	`0xAA`

Following this, entries begin with a one-byte value specifying their type, a single byte at offset `0`.

Types `0x01`, `0x90`, or `0x91` begin a section of the Boot Catalog, and contain a Platform ID for entries that follow. The platform ID is located at offset `1` in such an entry. The platform ID for a Neki32 application is `0x92`. A boot entry following this platform will contain game code. The platform ID for a Neki32 system-update bundle is `0x22`. A boot entry following this platform will contain packaged system-update data from Nekisoft.

Type `0x88` indicates a bootable entry. The platform ID of the bootable entry is given in a preceding section entry. The length of the bootable payload is given as a 2-byte little-endian value at offset `6`. The length is given in units of `512` bytes. The location of the bootable payload is given as a 4-byte little-endian value at offset `8`. The location is given in units of `2048` bytes, relative to the beginning of the game media.

The first bootable entry of platform ID `0x92` is taken to be the game executable to boot. The first bootable entry of platform ID `0x22` is taken to be the system update package to examine, if any. The format of the system update package is not described here. It should be obtained from Nekisoft and included verbatim.

2.3 The game executable

Once the location of the game executable is found in the El Torito Boot Catalog, the system begins to load it. The game executable is loaded into a new virtual memory space, by itself. The virtual memory space starts at address `0` and extends to `24M - 1`. Data is filled to the size specified in the El Torito Boot Catalog. The remaining memory is all `0x00`. As the size is given as a 16-bit number in units of `512` bytes, up to 32MBytes could be specified. However, a game on Neki32 is limited to 24MBytes of memory.

After loading, the zero page is checked for an appropriate magic number. The first eight bytes should be `NNEARM32`. The second eight bytes should be the entry point as a 64-bit little-endian value. This value must always be `0x1000`.

Once the game executable is loaded, the game process starts executing it from `0x1000`, with all registers zeroed.

2.4 Failures

If any of these steps encounters a missing magic-number or a failure to read the game media, the booting process stops. Instead of launching the game, the console will drop to its system menu. The system menu allows users to see the data saved on their console or turn it off.

If the game boots successfully, it should not exit. The user ends the game by turning the console off. If the game process terminates after being started, the system will try again to launch it, up to 3 times in total.

3 The instruction set

The game executable runs in user-mode on an ARMv5TE processor, including both ARM and Thumb modes. The system will always start executing the game in ARM mode, with the program counter at `0x1000`. Thumb interworking instructions may be used to transfer into and out of Thumb mode.

No floating-point instructions are available. Floating-point emulation works well enough to run Quake, at least, if you really want that.

The `udf 0x92` instruction, encoded as `0xe7f009f2`, is used to trigger system-calls. Other `udf` instructions should not be used. (If you are curious, we use a `udf` instruction because GDB ARM refuses to single-step over an `svc` instruction.) This implies that system-calls must be made from ARM mode, not Thumb mode.

4 Signals

The kernel of Neki32, "PVMK", provides virtualized handling of processor exceptions. This means that an illegal memory access or invalid opcode is caught by the operating system. When this happens, a signal is made pending on the process which caused it.

By default, all signals are masked, and will not be handled. Of course, this means that no action can be taken by the process to correct the situation. So the process is killed and the game crashes, when an exception causes a masked signal to become pending.

At its option, a game can unmask signals, and handle them instead. When a signal is pending and unmasked, it is handled by the process. The process resets due to the signal at `0x1000`, the same as its initial entry point. However, the signal number is stored in `r0`, rather than 0. All signals are masked when this happens, and the signal taken is no longer pending.

The interrupted context, from before the signal, is saved by the kernel. It can be resumed by using the `_sc_sig_return` system call.

5 Nonvolatile memory system (NVMs)

The Neki32 console includes memory for savegames. This allows games to be distributed on common SD cards made read-only, without requiring partially-read-only cards. Each game can save up to 128KBytes of data. The console has space for 30 such savegames, in a region of 4MBytes of NOR Flash.

To use NVM saving, a game must have a valid title set in the Volume ID of its ISO9660 Primary Volume Descriptor. The title must be nonempty, not all spaces, and not `CDROM` or `cdrom`. Additionally, the title must not start with a – character. Games that do not use NVM saving should start their Volume ID with a – character.

When the system reads a valid name from the Primary Volume Descriptor, it will set up NVM saving before booting the game. If insufficient space is available, the user will be warned at that time. They can continue anyway or clean up space. This all happens before launching the game.

Once the game is running, it may assume that NVM saving is available and working. Generally, the `_sc_nvm_save` and `_sc_nvm_load` calls will not fail. The game can freely load and store a region of up to 128KBytes. There is no need to handle no-space-available errors or to prompt the user about making space. If the user chooses not to make space, a temporary buffer in RAM is accessed by these system-calls instead.

NVM records are protected by SHA256 hashes and double-buffered. A single free record is always kept for saving the new version of an existing record. This means that interrupted writes will not corrupt the existing data. The old version will be used until the new version is written entirely. There is no need for a game to warn the user about interrupting a save.

The first time a game starts, before any data is saved, it is still valid to call `_sc_nvm_load`. The savegame will have been created during the boot process. A newly allocated savegame will contain 0 bytes when read back.

6 System calls

System-calls are used to access hardware features in a backward- and forward-compatible way. The system-call interface for Neki32 comprises 18 different calls.

To run a system call, first place its inputs in CPU registers. The call number is passed in register `r0`. Parameters, if any, are passed in registers `r1` to `r5`. Then, use the `udf 0x92` instruction. The kernel will perform the requested operation. The return value is stored in `r0` after the call finishes.

Typically, system-calls will return a negative error number if they fail, or a non-negative value on success. They may read or write in the memory of the caller.

System-calls are nonblocking. If a long-term operation is started, its system-call returns with `-_SC_EAGAIN`. The call can be repeated until it completes, returning a successful result. In the mean time, the `_sc_pause` system-call can be used to block the caller. For example, the following code would wait for any input events:

```
_sc_input_t mybuf[8] = {0};
while(_sc_input(mybuf, sizeof(mybuf[0]), sizeof(mybuf)) == -_SC_EAGAIN)
{
        _sc_pause();
}
```

6.1 Basics

These system-calls relate to the general usage of the system-call interface.

6.1.1 `_sc_none`

Called with: `r0 = 0x00`.
Does nothing. This enters and exits the kernel as usual but does nothing else.

- No parameters.

- No return value.

6.1.2 _sc_pause

Called with: `r0 = 0x01`.
Waits until any event happens to the calling process, or has happened since this call was last made. In this context, "any event" refers to the completion or failure of a prior system-call which returned `-_SC_EAGAIN`.

If an event has already occurred, the call returns immediately. If no event has occurred, the calling process will not be scheduled again until it does. This is the only way to actually block a process at the kernel level. Note that it is likely this system-call returns spuriously. It should be used in a loop if busy-waiting on completion of a system-call which is returning `-_SC_EAGAIN`.

- No parameters.
- No return value.

6.1.3 _sc_print

Called with: `r0 = 0xB0`.
Prints output to the text-mode screen. Prints the sequence of bytes at buf_ptr until a terminating NUL. Currently supports very few control sequences. Basically only used for debugging. Returns the number of bytes printed.

Note that this does not cause the text-mode screen to be displayed. Call `_sc_gfx_flip` with parameters of 0 to display the text-mode screen.

- `r1` : `const char* buf_ptr`
  Location of data to print.
- Returns : `int`
  The number of bytes printed, or a negative error number.

6.2 Game input/output

These system-calls relate to the user's gamepads and television. No setup or configuration is necessary to use these system-calls. Each program starts with the audio-visual and input peripherals ready to use.

6.2.1 _sc_getticks

Called with: `r0 = 0x02`.
Returns the number of milliseconds since the system was booted. Does not fail.

- No parameters.
- Returns : `int`
  The number of milliseconds elapsed since boot.

6.2.2 _sc_gfx_flip

Called with: `r0 = 0x30`.
Enqueues a change of the video front-buffer. The given buffer will become the front-buffer at the next vertical-blanking interval.

If the given mode is 0, text-mode will be displayed, and buffer must be given as NULL. If the given mode is nonzero, a valid buffer must be specified, and large enough for one full-screen image.

Returns the address of the buffer currently displayed. The return value may be 0 if the current front-buffer belongs to another process. This call may occasionally take effect immediately, and return its buffer parameter, if the call is made just before vertical blanking.
The following mode values are allowed:

Mode	Number	Description
_SC_GFX_MODE_TEXT	0	No framebuffer; kernel text mode only
_SC_GFX_MODE_VGA_16BPP	1	640x480@60Hz RGB565, 1280 bytes/line
_SC_GFX_MODE_320X240_16BPP	2	320x240@60Hz RGB565, 640 bytes/line

- `r1 : int mode`
  The video mode in which to display the new buffer.
- `r2 : const void * buffer`
  The location of the buffer in memory to display.
- Returns : `int`
  The currently-displayed image buffer, or a negative error number.

6.2.3 _sc_snd_play

Called with: `r0 = 0x60`.
Enqueues audio samples for playback. Samples are read from the given buffer and

copied into the kernel for playback. Either the whole buffer is consumed or none is. The caller may specify the maximum amount of audio to buffer in the kernel. This allows trading latency for stability. Returns the number of bytes the kernel has still buffered on success, or a negative error number.

The following mode values are allowed:

Mode	Number	Description
_SC_SND_MODE_SILENT	0	Stops all sounds
_SC_SND_MODE_48K_16B_2C	1	LPCM, 48KHz, 16b left-then-right, native endian

- `r1` : `int mode`
  The audio format of the data in the buffer.
- `r2` : `const void * chunk`
  The location of the buffer in memory to enqueue.
- `r3` : `int chunkbytes`
  The number of bytes in the buffer to enqueue.
- `r4` : `int maxbuf`
  The maximum number of bytes to enqueue in the kernel.
- Returns : `int`
  Current buffer usage (bytes) on success, or a negative error number.

6.2.4 _sc_input

Called with: `r0 = 0x50`.
Reads input events from the system into the given buffer. Returns how many events were filled in the buffer, or a negative error number.

The first byte of any event is a character indicating its type. Presently, only the following are defined:

First byte	Total bytes	Event type
`'A'` (`0x41`)	4	Player 1 digital gamepad input
`'B'` (`0x42`)	4	Player 2 digital gamepad input
`'C'` (`0x43`)	4	Player 3 digital gamepad input
`'D'` (`0x44`)	4	Player 4 digital gamepad input

The input events defined currently follow the format below:

Offset	Size	Field
0	1	Type
1	1	Unused
2	2	Buttons Pressed

The "buttons pressed" field is a bitmask where a 1-bit is a pressed button and a 0-bit is a released button. The buttons use the following indexes:

Button	Index	Bitmask
Up	0	0x0001
Left	1	0x0002
Down	2	0x0004
Right	3	0x0008
A	4	0x0010
B	5	0x0020
C	6	0x0040
X	7	0x0080
Y	8	0x0100
Z	9	0x0200
Start	10	0x0400
Mode	11	0x0800

This system-call will typically return an input event per frame per controller port. The buffer should be at least 4-byte-aligned.

- `r1` : `_sc_input_t * buffer_ptr`
  The location of the buffer to store the input events.
- `r2` : `int bytes_per_event`
  The number of bytes in each element of the buffer.
- `r3` : `int bytes_max`
  The total size of the buffer in bytes.
- Returns : `int`
  The number of events filled or a negative error number.

6.3 Data access

6.3.1 `_sc_disk_read2k`

Called with: `r0 = 0x91`.
Reads 2048-byte sectors from the disk into the given buffer. The sector number and count are given in units of 2KByte, i.e. not a byte-offset. Returns 0 on success or a negative error number. Any partial completion is considered a failure. May return `-_SC_EAGAIN` if the operation has started and will finish later.

- `r1` : `int sector_num`
  Which sector to load from the disk, in units of 2048 bytes.

- `r2` : `void * buf2k`
  The location in memory to store the data being read.

- `r3` : `int nsectors`
  The number of 2048-byte sectors to load.

- Returns : `int`
  0 on success, or a negative error number.

6.3.2 _sc_disk_write2k

Called with: `r0 = 0x92`.
Writes 2048-byte sectors to the disk from the given buffer. The sector number and count are given in units of 2KByte, i.e. not a byte-offset. Returns 0 on success or a negative error number. Any partial completion is considered a failure. May return `-_SC_EAGAIN` if the operation has started and will finish later.

- `r1` : `int sector_num`
  Which sector to write on the disk, in units of 2048 bytes.

- `r2` : `const void * buf2k`
  The data in memory to store onto the disk.

- `r3` : `int nsectors`
  The number of 2048-byte sectors to store.

- Returns : `int`
  0 on success, or a negative error number.

6.3.3 _sc_nvm_save

Called with: `r0 = 0x81`.
Writes data to the configured nonvolatile memory record, overwriting all previous data. The contents should be in the buffer at "data", of length "len". Writes are atomic and update the whole record each time. No partial writes are possible. If no record is configured, this call writes to a temporary per-process buffer instead. Returns the number of bytes written on success or a negative error number.

- `r1` : `const void * data`
  Location of data in memory to store to nonvolatile memory.
- `r2` : `int len`
  How many bytes to store.
- Returns : `int`
  The number of bytes written or a negative error number.

6.3.4 _sc_nvm_load

Called with: `r0 = 0x82`.
Reads from the configured nonvolatile memory record into the given buffer. The buffer to place the results in is at "buf", of length "len". Reads are protected by SHA256; corruption will result in the file being lost (`-_SC_ENOENT`). If no record is configured, this call reads from a temporary per-process buffer instead. Returns the number of bytes read on success or a negative error number.

- `r1` : `void * buf`
  The location in memory to put the data from nonvolatile memory.
- `r2` : `int len`
  How many bytes to load.
- Returns : `int`
  The number of bytes read or a negative error number.

6.4 Process setup

6.4.1 _sc_sig_mask

Called with: `r0 = 0x20`.
Alters the signal mask of the calling thread. Returns the *old* mask. The following signal numbers are used by the kernel:

Signal Name	Number	Description
`_SC_SIGILL`	4	Illegal operation, i.e. bad opcode
`_SC_SIGTRAP`	5	Debug trap requested
`_SC_SIGKILL`	9	Killed (cannot be blocked)
`_SC_SIGBUS`	10	Bus fault, i.e. misaligned access
`_SC_SIGSEGV`	11	Segmentation violation, i.e. out-of-bounds memory access
`_SC_SIGCHLD`	20	Child process changed state (not applicable to game processes)

The signal mask can be altered in the following ways:

Operation	Value	Description
_SC_SIGMASK_BLOCK	0	Blocks signals where the input bit is "1".
_SC_SIGMASK_UNBLOCK	1	Unblocks signals where the input bit is "1".
_SC_SIGMASK_SETMASK	2	Blocks signals if the input is "1" and unblocks all others.

- `r1` : `int how`
  Which operation to perform on the signal mask.

- `r2` : `int bits`
  The input bitmask for the given operation.

- Returns : `int`
  The old signal mask, before the successful operation. Or a negative error number on failure.

6.4.2 _sc_sig_return

Called with: `r0 = 0x22`.
Returns from a signal handler.

The kernel saves the context that is interrupted when a signal is taken. It then masks all signals and restarts the process to address `4096`. This system-call returns to the interrupted context, including the old signal mask.

As the kernel only saves a single context, signals are not reentrant. You probably do not want to unmask signals from inside a running signal handler.

- No parameters.

- This call does not return.

6.4.3 _sc_env_save

Called with: `r0 = 0x08`.
Appends the given data to the kernel's argument/environment buffer for the calling process. Subsequent calls append to the buffer; call with buf=len=0 to reset the buffer. This buffer is preserved across calls to exec() and mexec_apply(). Conventionally it should contain a series of NUL-terminated argument strings, then an extra NUL, then a series of NUL-terminated environment strings. Returns the number of bytes written or a negative error number.

- `r1` : `const void * buf`
  The data in memory to store to the environment buffer.

- `r2` : `int len`
  How many bytes to store.

- Returns : `int`
  The number of bytes written or a negative error number.

6.4.4 _sc_env_load

Called with: `r0 = 0x09`.
Reads from the kernel's argument/environment buffer for the calling process. Writes the result into the calling process's user memory, usually after an exec() or mexec_apply(). Unlike _sc_env_save, starts from the beginning each time it's called. Returns the number of bytes copied or a negative error number.

- `r1` : `void * buf`
  The location in memory to put the data from the environment buffer.

- `r2` : `int len`
  How many bytes to load.

- Returns : `int`
  The number of bytes read or a negative error number.

6.4.5 _sc_mexec_append

Called with: `r0 = 0xA1`.
Appends the given data to the kernel's pending memory image for the calling process. Subsequent calls append to the buffer; call with buf=len=0 to reset the buffer. The first 4KBytes appended are always inaccessible afterwards and can contain anything. The memory at address `0x1000` (+4KBytes in) is where execution starts after _sc_mexec_apply. Returns the number of bytes appended on success. Returns a negative error number if a failure occurs before any bytes were appended.

- `r1` : `const void * buf`
  The data in memory to append to the new memory space.

- `r2` : `int len`
  How many bytes to append.

- Returns : `int`
  The number of bytes appended or a negative error number.

6.4.6 _sc_mexec_apply

Called with: `r0 = 0xA2`.
Concludes an in-memory exec and replaces the current with the pending image. Does not return and does not fail. If there is no pending image, the caller exits as though killed by _SC_SIGSEGV.

- No parameters.
- This call does not return.

6.4.7 _sc_exit

Called with: `r0 = 0x07`.
Generally obliterates the calling process. Optionally reports a signal that was responsible for its demise.

- `r1 : int exitcode`
  The return value to report to the parent process.
- `r2 : int signal`
  The signal number responsible for the exit, if any. Zero otherwise.
- This call does not return.

6.5 Error codes

The following error codes may be returned by the kernel. These values are defined as positive integers here. When returned by a system-call, they are usually negated. For example, a system-call which fails because there is not enough memory might then return -12.

Error	Number	Description
_SC_EPERM	1	Operation not permitted.
_SC_ENOENT	2	No such file or directory.
_SC_ESRCH	3	No such process.
_SC_EIO	5	I/O error.
_SC_ENXIO	6	No such device or address.
_SC_E2BIG	7	Argument list too long.
_SC_ENOEXEC	8	Executable file format error.
_SC_EBADF	9	Bad file descriptor.
_SC_ECHILD	10	No child processes.
_SC_EAGAIN	11	Resource unavailable, try again.
_SC_ENOMEM	12	Not enough space.
_SC_EFAULT	14	Bad address.
_SC_EBUSY	16	Device or resource busy.
_SC_EEXIST	17	File exists.
_SC_ENOTDIR	20	Not a directory or a symbolic link to a directory.
_SC_EISDIR	21	Is a directory.
_SC_EINVAL	22	Invalid argument.
_SC_ENFILE	23	Too many files open in system.
_SC_EMFILE	24	File descriptor value too large.
_SC_ENOTTY	25	Inappropriate I/O control operation.
_SC_EFBIG	27	File too large.
_SC_ENOSPC	28	No space left on device.
_SC_ESPIPE	29	Invalid seek.
_SC_EPIPE	32	Broken pipe.
_SC_EDEADLOCK	35	Resource deadlock would occur.
_SC_ENAMETOOLONG	36	Filename too long.
_SC_ENOSYS	38	Functionality not supported.
_SC_ENOTEMPTY	39	Directory not empty.
_SC_ELOOP	40	Too many levels of symbolic links.

6.6 System calls by number

The following table lists all system-calls usable by a game, sorted by call number.

Number	Name
0x00	_sc_none
0x01	_sc_pause
0x02	_sc_getticks
0x07	_sc_exit
0x08	_sc_env_save
0x09	_sc_env_load
0x20	_sc_sig_mask
0x22	_sc_sig_return
0x30	_sc_gfx_flip
0x50	_sc_input
0x60	_sc_snd_play
0x81	_sc_nvm_save
0x82	_sc_nvm_load
0x91	_sc_disk_read2k
0x92	_sc_disk_write2k
0xA1	_sc_mexec_append
0xA2	_sc_mexec_apply
0xB0	_sc_print

7 The C SDK

The C SDK is an easy way to start writing programs for Neki32. It provides a runtime environment that behaves like a POSIX-compatible system, such as Linux or FreeBSD. The SDK includes the following parts.

7.1 SDK Contents

7.1.1 Compiler Wrappers

The compiler wrappers help to invoke GCC or LLVM with the right options for building a Neki32 executable. You must have an appropriate GCC or LLVM compiler installed, of course. It needs to be able to target the ARMv5TE architecture.

7.1.2 C Runtime

The C Runtime is what initially starts running when the Neki32 kernel loads your program. It requests memory for all your variables, unmasks signals, and calls your

main function. It also contains the linker configuration, so the linker can make an executable in the right format (a flat 0-based binary).

7.1.3 Picolibc

This is a port of PicoLibC to the Neki32. PicoLibC is a portable C Standard Library. It provides functions like printf and strcpy.

7.1.4 PVMK OS library

The OS Library is a link between the PicoLibC code and the system-calls available on the Neki32. It handles some things that aren't implemented in the "PVMK" kernel itself.

For example, the system-call interface on Neki32 provides block-level access to sectors from the game card. The system doesn't care what kind of filesystem is on the card. So, when your application calls fopen(), someone has to go understand the filesystem on the card and find the file you wanted. The code is in this library, so you can open/read/close like normal.

7.1.5 Updates Package

This is the official Neki32 system-update package from Nekisoft. This can be included on game media to ensure that players have the latest system software. Note that a license is required to distribute this - you must adhere to our software quality and marketing guidelines. See the licensing section for more information.

7.1.6 SDL System-Call Shims

This is an implementation of some system-calls on top of the SDL2 library. Using this, instead of the real system-call library, you can build and test your code on a desktop Linux machine. Then, you could compile the same application for Neki32 with minimal changes.

8 Examples

8.1 Development workflows

8.1.1 Just Assembly

It is possible to make a bootable Neki32 game card using only an ARM assembler. The executable code can be written in assembly, of course. The same assembly can also be used to make structures for ISO9660 and El Torito, referencing the executable code. For simplicity, we can place the executable at the beginning of the disk image. Then, a byte location in the disk, the executable, and the process memory are all the same. The output of the assembler is directly used as the disk image.

This approach can be used for simple games which fit entirely into the 24MByte RAM budget of the Neki32. That's 5 times bigger than the biggest Genesis cartridge!

Source code for this example is found in `examples/allasm` in the SDK. It doesn't require anything but GNU Make and the GNU Assembler for ARM.

8.1.2 Assembly and some Data

A program running on the Neki32 can load more data off the game card. The system-call `_sc_disk_read2k` is used to perform this operation. The previous example can be extended to load more data off the game card, after the boot program is already running.

This approach can be used for games which swap out graphical assets in simple ways. It has some advantages over using a filesystem - namely, that all locations are resolved at link-time. This makes the game a bit faster and more reliable. No overhead is spent accessing file metadata.

Source code for this example is found in `examples/asmdata` in the SDK. It doesn't require anything but GNU Make and the GNU Assembler and linker for ARM.

8.1.3 Assembly Accessing a Filesystem

For games with lots of data, it is helpful to keep the data in a filesystem. This aids in managing the data during development, and makes the environment more similar to a desktop workstation.

The ISO9660 filesystem is simple enough that it can be parsed from assembly code. The assembly code can then access any number of different files included in the ISO9660 filesystem.

In this case, the assembler is used to make an executable by itself (we call this format a "no-nonsense executable", or NNE). Then, mkisofs is used to put this executable in a bootable El Torito / ISO9660 image. Other files containing data are also placed in the image. Once running, the code can look through the ISO9660 filesystem to find its data files.

Source code for this example is found in `examples/asmisofs` in the SDK. It requires GNU Make and the GNU Assembler and linker for ARM, as well as the mkisofs utility.

8.1.4 Adding some Freestanding C

It is unlikely that you want to write an entire game in assembly. The previous examples can be extended by linking them against freestanding C code. This provides a barebones environment, lacking a standard library. It does, however, allow writing C while retaining control of the entire resulting executable.

Source code for this example is found in `examples/freestanding` in the SDK. It requires GNU Make and the GNU Assembler and linker for ARM, as well as the mkisofs utility, and the GNU C Compiler for ARM.

8.1.5 Using the C SDK Instead

For convenience, a C SDK is provided which gets the basics of a C runtime environment handled. It includes a port of Picolibc with an ISO9660 implementation providing file access via `_sc_disk_read2k`. It also includes startup files necessary to call `main()` in a sane way.

Source code for this example is found in `examples/sdkusage` in the SDK. It requires the SDK to be set up properly, including the GNU ARM toolchain and the C Runtime and C Standard Library for Neki32.

8.2 System call usage

8.2.1 Reading inputs

It is straightforward to read user input on the Neki32. A single system-call is used to retrieve user input as a series of events. Each event starts with a byte identifying what type of event it is. Then, event data follows.

A single invocation of the system-call may return multiple events in an array. The caller specifies the size of each array element, and the overall size of the array. The game knows, of course, the largest event it will handle, and can size its array

elements appropriately. Array entries are zero-filled if the event is smaller. Events are truncated if they do not fit.

Right now all events are 4 bytes in length. A single character identifies which controller port is considered, "A", "B", "C", or "D" (ASCII codes 0x61, 0x62, 0x63, 0x64). Then, one dummy byte is usually zero. Then a 16-bit value identifies which buttons are pressed as a bit-map.

The format of the input event and the button bitmasks are defined in `sc.h` in the C SDK. This example reads input from all 4 players and displays their control data on screen.

Source code for this example is found in `examples/showinput` in the SDK. It requires the SDK to be set up properly, including the GNU ARM toolchain and the C Runtime and C Standard Library for Neki32.

8.2.2 Double-buffered animation

Double-buffering can be used to synchronize the game code and video output. This is the simpler option, in contrast to triple-buffering.

Double-buffering means that memory for two video frames is used. At any time, one is used by the CPU to draw the next frame, while the other is being sent to the TV. The buffer that the CPU accesses is called the back buffer, and the buffer being sent to the TV is the front buffer. Note that it takes almost an entire frame of time, to send that one frame of data to the TV.

The TV is given some free time between bursts of video data, called "blanking". When the TV is in blanking, no data is sent to it. At the end of each line in a frame, there is a "horizontal blanking" time. Horizontal blanking is very short in duration. At the end of each frame, there is a "vertical blanking" time.

During vertical blanking, there is about 1ms when no data is sent. A complete frame has already been transmitted. At this time, the buffers can be interchanged, called "swapping" or "flipping" the buffers. Then the CPU can re-draw to the buffer which was previously displayed, as the next one is sent to the TV. In this way, a series of complete frames is sent to the TV.

In double-buffering on Neki32, the `_sc_gfx_flip` system call is used in a loop with `_sc_pause`. Until `_sc_gfx_flip` indicates that the desired buffer is being displayed, the game continues to pause. Effectively, the game blocks on the "flip" syscall and proceeds when the new image is displayed.

Source code for this example is found in `examples/dblbuf` in the SDK.

8.2.3 Triple-buffered animation

Triple-buffering is more complicated than double-buffering, but has some speed advantages. At any time, one image is being sent to the TV, one is being drawn by the CPU, and one is kept spare.

In double-buffering, if the CPU has finished drawing a frame, it must wait for the prior frame to be entirely sent to the TV. In triple-buffering, an additional spare buffer is kept. Therefore, the CPU can move on to render the following frame immediately. If the CPU is running fast, it can continue to render a series of frames, independently of the TV scanout. It ping-pongs between the two buffers which are not being sent to the TV. Whenever one frame is done being sent to the TV, there is always another complete frame ready to go.

This requires three frame buffers to be allocated instead of two, of course. When `_sc_gfx_flip` is called, the game does not wait on its result. Instead, the game evaluates the situation and continues each time it calls `_sc_gfx_flip`. The parameter to `_sc_gfx_flip` will be enqueued for display at the next vertical-blanking interval. The return value from `_sc_gfx_flip` is the buffer currently displayed, at that moment. Therefore, the game finds whichever buffer is neither of those. It continues rendering into that one.

Source code for this example is found in `examples/tplbuf` in the SDK.

8.2.4 Saving saves

It is simple to save the player's progress to Nonvolatile Memory in the console.

An area of memory can be allocated in the game process, big enough to hold the saved data. When the game starts, the system-call `_sc_nvm_load` is used to populate the area with the saved data. Then, whenever appropriate, the system-call `_sc_nvm_save` can be used to store it back to the console.

It is only necessary to check, on start-up, that the region is not all `0x00`. In this case, no data was previously saved, and the savegame region should be initialized in RAM before calling `_sc_nvm_save`.

It is not necessary to handle failures to save, or corruption from interrupted saves. The system will make sure there is space allocated, at boot, before the game executes. The system will keep checksums and double-buffer the saved data, to protect against data loss. Physical writes are ordered to ensure reliability.

Source code for this example is found in `examples/nvmsave` in the SDK.

8.3 SDK features

The C SDK includes a mostly-complete C standard library that enables many POSIX-like operations.

8.3.1 Writing to files

Ordinarily, we recommend that a game card is permanently write-protected, to protect against damage. Savegames can be written to the Nonvolatile Memory in the console in this case, so no special SD card is needed. However, the system supports writing back to game cards if you want to.

The SDK includes support for overwriting file contents on an ISO9660 filesystem. Files cannot be created or deleted, or their length changed. However their contents, at their existing location and length, can be modified. The actual writes are performed when the SDK calls `_sc_disk_write2k`.

The example in `examples/rwcard` uses this to save data which the user has modified. It contains an existing file in the ISO9660 image, called SAVE.BIN, whose contents are modified at runtime.

9 Common errors

This section contains examples of why a program may not work as expected.

9.1 System call failures

Most system-calls in PVMK will always complete predictably if given valid parameters. Unexpected failures are rare due to the restricted nature of the environment. An enumeration of system-calls and potential failures follows.

9.1.1 `0x00 _sc_none`

- This call cannot fail.
- In some sense, it also cannot succeed.

9.1.2 0x01 _sc_pause

- This call cannot fail.
- Note that this call may return spuriously for a number of reasons. A previous system-call may have triggered an unpause which has been ignored. Or, the timing of a following system-call may not be predictable.
- To block waiting on a system-call result, retry the call, and use `_sc_pause` in a loop. Call pause if and only if an `-_SC_EAGAIN` result is returned from the call in question.

9.1.3 0x02 _sc_getticks

- This call cannot fail.
- Note that this returns ticks since the system booted, not since the program began executing.
- When using the debugger, this may also show discontinuities.
- The units are milliseconds. As the value is 32-bit, the tick count may overflow if the user leaves the console running for 25 days. Games do not need to handle this, but may choose to handle the overflow or to crash.

9.1.4 0x07 _sc_exit

- This call cannot fail.
- The system may try to re-launch a game that exits, up to 3 times.

9.1.5 0x08 _sc_env_save

- This call can fail if the buffer is not in valid memory.
- This call can fail if the environment buffer is already full.
- This call can fail if the pointer is NULL but the length is nonzero.
- This call can fail if the length is zero but the pointer is non-NULL.
- This call can fail if the length is too large for the environment at all.
- Note that subsequent calls append to the buffer, not starting again.

9.1.6 0x09 _sc_env_load

- This call can fail if no data is in the environment buffer already.
- This call can fail if the buffer is not in valid memory.
- This call can fail if the given buffer is too small for the environment stored.
- Note that subsequent calls do not continue reading from the buffer, instead starting again.

9.1.7 0x20 _sc_sig_mask

- This call can fail if the "how" parameter is invalid.
- This call can cause a signal to be taken immediately when unmasked, if it was already pending.

9.1.8 0x22 _sc_sig_return

- This call only restores a single saved context.
- This call will certainly misbehave if called before a signal is handled, or called twice for one signal.
- Enabling signals again in a signal handler may cause the old saved context to be lost.

9.1.9 0x30 _sc_gfx_flip

- This call can fail if an invalid buffer pointer is given.
- This call can fail if an invalid mode value is given.
- This call can fail if the mode is "text" mode (0) but the buffer is non-NULL.
- This call can fail if the mode is not "text" (nonzero) but the buffer is NULL.
- This call can occasionally return the buffer it was given, immediately. This happens if the call is made just as the system enters vertical blanking.
- This call can return 0 if the currently-displayed buffer belongs to another process.

9.1.10 0x50 _sc_input

- This call can fail if the buffer is not in valid memory.

- This call can fail if the buffer sizes are zero or negative.
- This call can fail if the buffer size overall is not a multiple of the buffer element size.

9.1.11 0x60 _sc_snd_play

- This call can fail if the chunk size is too large for the kernel to buffer at all.
- This call can fail if the maximum buffered size is set too low.
- This call can fail if the buffer is not in valid memory.
- This call can fail if the mode given is invalid. The only valid mode currently is 1.

9.1.12 0x81 _sc_nvm_save

- This call can fail if the buffer pointer and length do not refer to valid memory.
- This call can fail if the buffer is oversized.
- If the game did not provide a valid name in its ISO9660 Primary Volume Descriptor, this call will not actually write to nonvolatile memory.
- If the user chose not to make space for the game, this call will not actually write to nonvolatile memory.
- In cases where no nonvolatile memory record is prepared, this call writes to a temporary buffer for the process instead.

9.1.13 0x82 _sc_nvm_load

- This call can return 0 bytes if the save record was just created.
- This call can fail for the same reasons as `_sc_nvm_save`.

9.1.14 0x91 _sc_disk_read2k

- This call can fail if the sector number is invalid.
- This call can fail if the buffer is not in valid memory.
- This call can fail of the number-of-sectors parameter is less than 1 or greater than 12288.

9.1.15 0x92 _sc_disk_write2k

- This call can fail if the sector number is invalid.
- This call can fail if the buffer is not in valid memory.
- This call can fail if the card is write-protected.
- This call can fail of the number-of-sectors parameter is less than 1 or greater than 12288.
- It is not recommended to use this call unless the card can be partially write-protected.

9.1.16 0xA1 _sc_mexec_append

- This call can fail if the pending memory image is already too large.
- This call can fail if the given buffer is not in valid memory.
- This call can fail if the given buffer is too large for the pending memory image.
- This call can fail if the buffer pointer is NULL but the length is nonzero.
- This call can fail if the buffer length is zero but the pointer is non-NULL.

9.1.17 0xA2 _sc_mexec_apply

- This call cannot fail.
- If the pending memory image is not valid, the process will die from a segfault.

9.1.18 0xB0 _sc_print

- This call can fail if the buffer is not in valid memory.

9.2 Crashes

A processor exception will result in a crash if the corresponding signal is blocked. The following things can cause this:

- Accessing an invalid memory location will cause `SIGSEGV`.

Valid memory locations start at `0x1000` and extend to one byte before the size of the process. The size of the game process, by default, is always `24*1024*1024` (24 megabytes). If `_sc_mexec_apply` has been used to load a new program, the process

size is the total number of bytes appended by `_sc_mexec_append` beforehand. Note that the caller is required to set the total size of the new process, including the `.bss` section, by appending that many bytes. New memory cannot be added to the process once it is running.

- Accessing a misaligned word or halfword will cause `SIGBUS`.

Memory accesses on ARM must be naturally-aligned. This means that a 32-bit (word) access must be to an address which is a multiple of `4`, and a 16-bit (halfword) access must be to an address which is a multiple of `2`. Otherwise, an alignment-check fault results and causes this signal.

- Executing an undefined instruction will cause `SIGILL`.

The Neki32 supports the ARMv5TE instruction set. It additionally uses a single architecturally-undefined instruction, `udf 0x92`, to initiate a system-call. Any instructions outside this set will trigger an undefined opcode fault and raise this signal.

- System-calls do not work from Thumb mode.

Currently the system only supports the ARM encoding of the `udf 0x92` instruction (`0xe7f009f2`). Thumb code therefore must switch back to ARM mode to execute a system-call.

10 Secret Codes

These features of the Neki32 console may assist in development.

10.1 At power-on

The Neki32 console stores its default system software, called Kernel Zero, in read-only memory. It can also store a single updated version of the system software. If the updated version is valid, it will be booted automatically instead of the original. To boot the original version instead of the updated version, hold the Down direction on controller D while turning the system on.

If system software updates are present on a game card, the system will find the newest one matching the hardware. It will prompt the user to update if it is newer than the current version. To force this prompt to always appear, regardless of the version numbers, hold Y on controller A while turning the system on. To skip this prompt, regardless of the version numbers, hold X on controller A while turning the system on.

10.2 At the boot menu

Some additional information is available from the Neki32 system menu. To see additional hardware test features, enter the code "XYZZY" on controller A at the main menu. To see the version number, and a message about why no game was booted, hold the Z and Mode buttons on controller A at the main menu.