PS Vita™ Open SDK Specification Version 1.21

Yifan Lu

October 19, 2016

Revision History

Revision	Date	Author(s)	Description
1.0	23/05/2015	Yifan Lu	Initial version
1.1	22/06/2016	Yifan Lu	Fixed some typos and swapped the defination of "module" and "library"
1.2	25/09/2016	Yifan Lu	Revised NID JSON database format to match current implementations. Added information on new import structure. Added information on shared libraries. Added specification for exporting libraries.
1.21	19/10/2016	Yifan Lu	Added a table limitations for ELF values. Modified algorithm for NID generation.

Contents

1 Introduction

The documents outlines the requirements and implementation advice for an open source software development library and toolchain for creating object code for the PS Vita™ device.

1.1 Goals

The main goal for this project is to create an ecosystem of amateur produced software and games (*homebrew*) on the PS Vita™ device for non-commercial purposes. The inspiration for this document comes from observed failures of open toolchains on other gaming platforms. The goal is to define precisely the requirements and implementation of an open source toolchain and SDK for the PS Vita™. Collaboration from the community is expected and desired in the creation of this ecosystem. Comments and suggestions for this document should be sent directly to the author.

1.2 Legal

PS Vita™ is a trademark of Sony Computer Entertainment America LLC. This document is written independently of and is not approved by SCEA. Please don't sue us.

1.3 Overview

The PS Vita™ carries an ARM Cortex A9 MPCore as the main CPU processor in a custom SoC. The processor implements the ARMv7-R architecture (with full Thumb2 support). Additionally, it supports the MPE, NEONv1, and VFPv3 extensions.

The software infrastructure is handed by a proprietary operating system; the details of which is outside the scope of this document. What this document will define is the executable format, which is an extension of the ELF version 1 standards. Thorough knowledge of the ELF specifications[1] is assumed and the SCE extensions will be described in detail. The simplified specifications[2] and ARM extensions to ELF[3] will be referenced throughout this document.

The first part of this document will describe the format of SCE ELF executables including details on SCE extension segment and sections. The second part will detail a proposed SDK format for writing the include files and symbol-NID mapping database. The third part will specify a tool which can convert a standard Linux EABI ELF into a SCE ELF.

2 SCE ELF Format

2.1 ELF Header

The header is a standard ARM ELF[3] header. For the `e_type` field, there are some additional options.

Name	Value	Meaning
`ET_SCE_EXEC`	`0xFE00`	SCE Executable file
`ET_SCE_RELEXEC`	`0xFE04`	SCE Relocatable file
`ET_SCE_STUBLIB`	`0xFE0C`	SCE SDK Stubs
`ET_SCE_DYNAMIC`	`0xFE18`	Unused
`ET_SCE_PSPRELEXEC`	`0xFFA0`	Unused (PSP ELF only)
`ET_SCE_PPURELEXEC`	`0xFFA4`	Unused (SPU ELF only)
`ET_SCE_UNK`	`0xFFA5`	Unknown

Figure 1: SCE specific ELF type values

The difference between executable files and relocatable files is that executables have a set base address. Relocatable ELFs were used before FW 2.50 only for PRX, however in the latest versions, any application with ASLR support is relocatable. The open toolchain is required to support `ET_SCE_RELEXEC`. All others are optional. The PS Vita system imposes some limits on the ELF files.

Field	Limit
`e_phnum`	At most 5 for `ET_SCE_EXEC` and 8 for `ET_SCE_RELEXEC`
`e_phnum`	At most 3 `PT_LOAD` segments
`e_phnum`	At most 3 `SCE_RELOC` segments

Figure 2: ELF limits

There are three filename extensions for SELFs. `self` is usually used for application executables and likely stands for "secure ELF" or "Sony ELF". `suprx` are userland dynamic libraries and is similar to how `so` libraries work on Linux. `skprx` are kernel modules and is similar to how `ko` libraries work on Linux. Even though the extensions are different, all these file types use the same SCE ELF format. There is no difference between an `ET_SCE_RELEXEC` application and a `suprx` except that `suprx` usually exports additional libraries for linking. That means, in theory, a single SCE ELF can act as both an application and a userland library. The only difference between `suprx` and `skprx` is that `skprx` is meant to run in kernel and can use ARM system instructions. A `skprx` can also export libraries to userland in the form of syscalls (more information below).

2.2 ELF Sections

SCE ELFs define additional section types for the `sh_type` field.

Name	Value	Meaning
`SHT_SCE_RELA`	`0x60000000`	SCE Relocations
`SHT_SCENID`	`0x61000001`	Unused (PSP ELF only)
`SHT_SCE_PSPRELA`	`0x700000A0`	Unused (PSP ELF only)
`SHT_SCE_ARMRELA`	`0x700000A4`	Unused (SPU ELF only)

Figure 3: SCE specific ELF section types

The toolchain is required to support `SHT_SCE_RELA`, which is how relocations are implemented in SCE ELFs. The details are described in the following subsection.

2.2.1 SCE Relocations

SCE ELFs use a different relocation format from standard ELFs. The relocation entries are in two different format, either an 8 byte "short" entry or a 12 byte "long" entry. You are allowed to mix and match "short" and "long" entries, but that is not recommended for alignment reasons. The entire relocation segment is just a packed array of these entries.

In the short entry, the offset is stored partially in the first word and partially in the second word. It also has a 12-bit addend. In the long entry, there is support for two relocations on the same data. The open toolchain does not have to implement this. Long entries have 32-bit addends.

- `r_format` determines the entry format. Currently two are supported: `0x0` is "long entry" and `0x1` is "short entry". In previous versions of this document, this field was called `r_short`.

- `r_symseg` is the index of the *program segment* containing the data to point to. Previous versions of this documented noted that if this value is 0xF, then `0x0` is used as the base address. This is *no longer true* in recent system versions.

- `r_code` is the relocation code defined in ARM ELF[3]

- `r_datseg` is the index of the *program segment* containing the pointer that is to be relocated.

- `r_offset` is the offset into the segment indexed by `r_datseg`. This is the pointer to relocate.

- `r_addend` is the offset into the segment indexed by `r_symseg`. This is what is written to the relocated pointer.

```c
// assuming LSB of bitfield is listed first
union {
    Elf32_Word r_format : 4;
    struct {
        Elf32_Word r_format : 4; // 0x1
        Elf32_Word r_symseg : 4;
        Elf32_Word r_code : 8;
        Elf32_Word r_datseg : 4;
        Elf32_Word r_offset_lo : 12;
        Elf32_Word r_offset_hi : 20;
        Elf32_Word r_addend : 12;
    } r_short_entry;
    struct {
        Elf32_Word r_format : 4; // 0x0
        Elf32_Word r_symseg : 4;
        Elf32_Word r_code : 8;
        Elf32_Word r_datseg : 4;
        Elf32_Word r_code2 : 8;
        Elf32_Word r_dist2 : 4;
        Elf32_Word r_addend;
        Elf32_Word r_offset;
    } r_long_entry;
} SCE_Rel;
```

Figure 4: SCE relocation entry

2.2.2 Relocation Operations

Only the following ARM relocation types are supported on the PS Vita™:

The toolchain is required to only output relocations of these types. Refer to that ARM ELF[3] manual for information on how the value is formed. The definitions of the variables for relocation is as follows.

Segment start = Base address of segment indexed at `r_datseg`

Symbol start = Base address of segment indexed at `r_symseg`

P = segment start + `r_offset`

S = symbol start

A = `r_addend`

2.3 SCE Dynamic Section

ELF Dynamic sections are not used (a change from PSP). Instead all dynamic linking information is stored as part of the export and import sections, which are `SHT_PROGBITS` sections.

Code	Name	Operation
0	`R_ARM_NONE`	
2	`R_ARM_ABS32`	$S + A$
3	`R_ARM_REL32`	$S + A - P$
10	`R_ARM_THM_CALL`	$S + A - P$
28	`R_ARM_CALL`	$S + A - P$
29	`R_ARM_JUMP24`	$S + A - P$
38	`R_ARM_TARGET1` (same as `R_ARM_ABS32`)	$S + A$
40	`R_ARM_V4BX` (same as `R_ARM_NONE`)	
41	`R_ARM_TARGET2` (same as `R_ARM_REL32`)	$S + A - P$
42	`R_ARM_PREL31`	$S + A - P$
43	`R_ARM_MOVW_ABS_NC`	$S + A$
44	`R_ARM_MOVT_ABS`	$S + A$
47	`R_ARM_THM_MOVW_ABS_NC`	$S + A$
48	`R_ARM_THM_MOVT_ABS`	$S + A$

Figure 5: SCE specific ELF section types

2.3.1 NIDs

Instead of using symbols, SCE ELF linking depends on NIDs. These are just 32-bit integers created from hashing the symbol name. The formula for generating them does not matter as long as they match up. For our purposes, we will make sure our open SDK recognizes NIDs for imported functions and when NIDs are created, they can be done so in an implementation defined way.

2.3.2 Module Information

The first SCE specific section `.sceModuleInfo.rodata` is located in the same program segment as `.text`. It contains metadata on the module[1]

Some fields here are optional and can be set to zero. The other fields determine how this module is loaded and linked. All offset fields are formatted as follows: top 2 bits is an index to the segment to start at and bottom 30 bits is an offset from the segment. Currently, the segment start index must match the segment that the module information structure is in.

- `version`: Set to `0x0101`
- `name`: Name of the module
- `type`: `0x0` for executable, `0x6` for PRX

[1] Previous versions of this document swapped the usage of the term "library" and "module". This change is to be more consistent with Sony's usage of the term.

```c
struct {
    u16_t   attributes;
    u16_t   version;
    char    name[27];
    u8_t    type;
    void    *gp_value;
    u32_t   export_top;
    u32_t   export_end;
    u32_t   import_top;
    u32_t   import_end;
    u32_t   module_nid;
    u32_t   field_38;
    u32_t   field_3C;
    u32_t   field_40;
    u32_t   module_start;
    u32_t   module_stop;
    u32_t   exidx_top;
    u32_t   exidx_end;
    u32_t   extab_top;
    u32_t   extab_end;
} sce_module_info;
```

Figure 6: SCE module information

- `export_top`: Offset to start of export table.
- `export_end`: Offset to end of export table.
- `import_top`: Offset to start of import table.
- `import_end`: Offset to start of import table.
- `module_nid`: NID of this module. Can be a random unique integer. This can freely change with version increments. It is not used for imports.
- `module_start`: Offset to function to run when module is started. Set to 0 to disable.
- `module_stop`: Offset to function to run when module is exiting. Set to 0 to disable.
- `exidx_top`: Offset to start of ARM EXIDX (optional)
- `exidx_end`: Offset to end of ARM EXIDX (optional)
- `extab_top`: Offset to start of ARM EXTAB (optional)
- `extab_end`: Offset to end of ARM EXTAB (optional)

2.3.3 Module Exports

Each module can export one or more *libraries*. To get the start of the export table, we add `export_top` to the base of the segment address. To iterate through the export tables, we read the size field of each entry and increment by the size until we reach `export_end`.

```
struct {
    u16_t    size;
    u16_t    version;
    u16_t    flags;
    u16_t    num_syms_funcs;
    u16_t    num_syms_vars;
    u16_t    unk_1;
    u32_t    unk_2;
    u32_t    library_nid;
    char     *library_name;
    u32_t    *nid_table;
    void     **entry_table;
} sce_library_exports;
```

Figure 7: SCE library export

- `size`: Set to `0x20`. There are other sized export tables that follow different formats. We will not support them for now.
- `version`: Set to `0x1` for a normal export or `0x0` for the main module export.
- `flags`: An OR mask of valid flag values.
- `num_syms_funcs`: Number of function exports.
- `num_syms_vars`: Number of variable exports. Must be zero for kernel library export to user.
- `library_nid`: NID of this library. Can be a random unique integer that is consistent. Importers will use this NID so it should only change when library changes are not backwards compatible.
- `library_name`: Pointer to name of this exported library. For reference only and is not used in linking.
- `nid_table`: Pointer to an array of 32-bit NIDs to export.
- `entry_table`: Pointer to an array of data pointers corresponding to each exported NID (of the same index).

Flag	Mask	Description
Importable	`0x1`	Should be set unless it is the main export.
User Importable	`0x4000`	In kernel modules only. Allow syscall imports.
Main Export	`0x8000`	Set for main export.
Unknown	`0x10000`	

Figure 8: Export flags

Note that since pointers are used, the `.sceLib.ent` section containing the export tables can be relocated. The data pointed to (name string, NID array, and data array) are usually stored in a section `.sceExport.rodata`. The order in the arrays (NID and data) is: function exports followed by data exports followed by the unknown exports (the open toolchain should define no such entries). The `.sceExport.rodata` section can also be relocated.

For all executables, a library with NID *0x00000000* and attributes *0x8000* exports the `module_start` and `module_stop` functions along with a pointer to the module information structure as a function export. The NIDs for these exports are as follows:

Name	Type	NID
module_stop	Function	`0x79F8E492`
module_exit	Function	`0x913482A9`
module_start	Function	`0x935CD196`
module_info	Variable	`0x6C2224BA`

Figure 9: Required module export

2.3.4 Module Imports

Each module also has a list of imported libraries. The format of the import table is very similar to the format of the export table.

- `size`: Set to `0x34`. There are other sized import tables that follow different formats. We will not support them for now.
- `version`: Set to `0x1`.
- `flags`: Set to `0x0`.
- `num_syms_funcs`: Number of function imports.
- `num_syms_vars`: Number of variable imports.
- `library_nid`: NID of library to import. This is used to find what module to import from and is the same NID as the `library_nid` of the library from the exporting module.

```c
struct {
    u16_t    size;
    u16_t    version;
    u16_t    flags;
    u16_t    num_syms_funcs;
    u16_t    num_syms_vars;
    u16_t    num_syms_unk;
    u32_t    reserved1;
    u32_t    library_nid;
    char     *library_name;
    u32_t    reserved2;
    u32_t    *func_nid_table;
    void     **func_entry_table;
    u32_t    *var_nid_table;
    void     **var_entry_table;
    u32_t    *unk_nid_table;
    void     **unk_entry_table;
} sce_module_imports;
```

Figure 10: SCE module import

- `library_name`: Pointer to name of the imported library. For reference only and is not used for linking.
- `func_nid_table`: Pointer to an array of function NIDs to import.
- `func_entry_table`: Pointer to an array of stub functions to fill.
- `var_nid_table`: Pointer to an array of variable NIDs to import.
- `var_entry_table`: Pointer to an array of data pointers to write to.

The import tables are stored in the same way as export tables in a section `.sceLib.stubs` which can be relocated. The data pointed to are usually found in `.sceImport.rodata`. The function NIDs to import (for all imported libraries) is usually stored in section `.sceFNID.rodata` and the corresponding stub functions are in `.sceFStub.rodata`. The stub functions are found in `.text` and can be any function that is 12 bytes long (however, functions are usually aligned to 16 bytes, which is fine too). Upon dynamic linking, the stub function is either replaced with a jump to the user library or a syscall to an imported kernel module. The suggested stub function is:

Imported variable NIDs can be stored in section `.sceVNID.rodata` and the data table in `.sceVNID.rodata`.

```
mvn r0, #0x0
bx lr
mov r0, r0
```

Figure 11: Stub function code

2.3.5 New Import Format

Newer firmware versions have modules with import tables of size `0x24` bytes instead of the usual `0x34` byte structure defined above. The only difference is that some fields are dropped. For reference, the new structure is defined below. However, support for it is optional. Software can differentiate the two versions by looking at the size field. All import entries must be the same size!

```
struct {
    u16_t   size;
    u16_t   version;
    u16_t   flags;
    u16_t   num_syms_funcs;
    u32_t   reserved1;
    u32_t   library_nid;
    char    *library_name;
    u32_t   *func_nid_table;
    void    **func_entry_table;
    u32_t   unk1;
    u32_t   unk2;
} sce_module_imports_new;
```

Figure 12: New SCE module import

2.3.6 Diagram

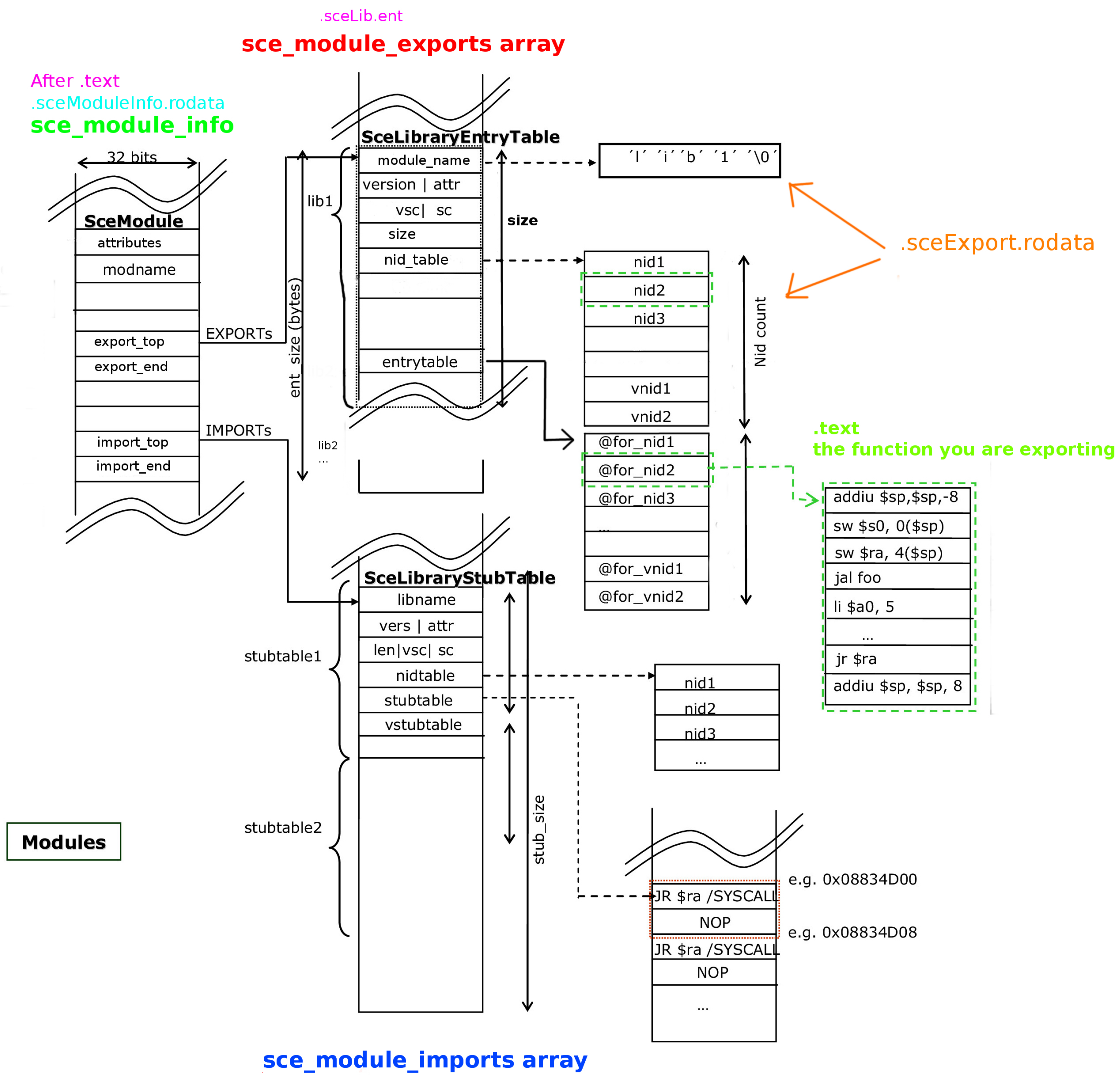

Figure 13: Visual representation of all the parts in the SCE sections. Credits to Anissian and xerpi.

2.4 ELF Segments

SCE ELFs define additional program segment types for the `p_type` field.

Name	Value	Meaning
PT_SCE_RELA	0x60000000	SCE Relocations
PT_SCE_COMMENT	0x6FFFFF00	Unused
PT_SCE_VERSION	0x6FFFFF01	Unused
PT_SCE_UNK	0x70000001	Unknown
PT_SCE_PSPRELA	0x700000A0	Unused (PSP ELF only)
PT_SCE_PPURELA	0x700000A4	Unused (SPU ELF only)

Figure 14: SCE specific ELF program segment types

The toolchain is only required to support `PT_SCE_RELA`. This program segment is essentially just a composition of all `SHT_SCE_RELA` sections.

2.4.1 Module Information Location

For `ET_SCE_EXEC` executables, the module information is stored in the first segment (where the code is loaded). The location of the `sce_module_info` structure is at `p_paddr` offset from the start of the ELF file. Once the ELF is loaded into memory, the location is segment base address + `p_paddr` - `p_offset`.

For `ET_SCE_RELEXEC` executables, the segment containing `sce_module_info` is indexed by the upper two bits of `e_entry` of the ELF header. The structure is stored at the base of the segment plus the offset defined by the bottom 30 bits of `e_entry`.

3 Open SDK Format

We will first specify a format for defining a database of NIDs to symbol name mappings in JSON. The motivation behind this is that most ELF tools deal with Linux APIs and symbols. We should not have to write our own linker but instead have a tool that converts a linked executable to the SCE ELF format. This database will be built by reverse engineers who will extract NIDs and figure out how the APIs work.

3.1 JSON NID Database

Let's start with a motivating example for what a typical API export would look like.

```
{
  "SceLibKernel": {
    "nid": 1237592384,
    "modules": {
      "SceLibKernel": {
        "nid": 3404311782,
        "kernel": false,
        "functions": {
          "sceKernelPuts": 37661282,
          "sceKernelGetThreadId": 263811833,
          "sceIoDevctl": 78843058,
          ...
        },
        "variables": {
          "SceKernelStackGuard": 1146666227,
          ...
        }
      },
      "SceLibGcc": {
        "nid": 1450899878,
        "kernel": false,
        "functions": {
          ...
        },
        "variables": {
          ...
        }
      },
      ...
    }
  },
```

```
  "SceIoFilemgr": {
    "nid": 1042566167,
    "modules": {
      ...
    }
  },
  ...
}
```

We start out with an array of module definitions. Each module has an associated module NID (value from `module_nid` field of `sce_module_info` structure) and an array of library exports. This NID can change across firmware versions, but should be consistent in our database. This is because the Vita does not use this NID but we will use it in linking. We can assign the module NID to be any valid and unique value that is consistent across database updates. Each library has an library NID (value from `library_nid` field of each export table). If the library can only be accessed in kernel (no syscalls are exported), then "kernel" is set to true, otherwise if the library is in userspace or has syscall exports, it is set to false. Each library has an array of functions and an array of variables which maps the symbol name to an NID.

3.2 Header Files

The header files written should be commented with Doxygen syntax. API documentation will be generated by Doxygen.

3.3 Library Files

Library stub files for static linking will be generated by the vita-libs-gen tool which uses the JSON API database to create temporary libraries to statically link to.

3.4 Export Configuration Files

For those who wish to build dynamically linkable modules (`.suprx` or `.skprx`), a YAML based configuration file is specified for setting up exports. Just like above, we begin with a motivating example.

```
MyPlugin:
  attributes: 0x1000     # optional: default to 0x1000
  version:               # optional
    major: 1             #   default to 1
    minor: 5             #   default to 0
  nid: 0xeeeeeeee        # optional: can set nid manually
  main:                  # optional
    start: module_start  #   default to ""
```

```yaml
    stop: module_stop     #   default to ""
    exit: module_exit     #   default to ""
  modules:
    MyPluginForUser:
      kernel: false       # optional: default is false
      functions:
        - myPlgFunc1
        - myPlgFunc2
        - myPlgFunc3
      variables:          # only supported in user type
        - someVar1
        - someVar2
    MyPluginForKernel:
      kernel: true
      functions:
        - myPlgFunc1
        - myPlgFunc3
        - myPlgSecretFunc
    MyPluginForDriver:
      kernel: true
      nid: 0xdeadbeef     # optional: set nid manually
      functions:
        - myPlgSecretFunc
        - myPlgDriverFunc
```

The format matches how the JSON database for NID imports look (except in YAML instead of JSON) with the exception that function and variable NIDs are unspecified. The reason for using YAML instead of JSON is that YAML is easier to read and write (for developers) while JSON is easier to parse (for linking). The dichotomy in formats also helps prevents mistakes of using an import file (which has NIDs specified) instead of an export file (which has NIDs unspecified). Library and module NIDs can optionally be set to an user-specified value. If left unspecified, then it will be determined with the algorithm below. Function and variable NIDs must be generated with the algorithm. Note that this algorithm is *not* the same as SCE's own algorithm, but the details should not matter for interoperability.

The hash for module NID should be unique for a given module source and compiler configuration. An example would be to hash the ELF file before converting it to a SCE ELF. The hash for library name and function/variable name is variable length and does not include the null terminator. The SHA256-32 algorithm takes the full 32 byte hash and only returns the first four bytes as a little endian integer.

Since module NID is expected to be different in each version. However, other NIDs must stay consistent regardless of version. Changing those NID will break compatibility with anything that uses the module.

Type	Algorithm
Module NID	SHA256-32(unique data)
Library NID	SHA256-32(export_info.library_name)
Main Export NID	See 2.3.3
Other NID	SHA256-32(name)

Figure 15: NID generation algorithm for Open SDK exports

The “kernel” field of a library export is used only for `skprx`. It is not permitted to set “kernel” true for user modules. In a kernel module, setting “kernel” true will only allow the library to be imported by another kernel module. Setting it to false (the default value) will allow user modules to import it through a syscall.

The “main” field allows you to manually specify the symbol for each of the main export fields. If “start” is blank, that means the ELF entry point will be used as the start. If stop or exit are blank, then they will not be set in the resultant SCE ELF. To manually specify the start, stop, or exit function, set the field to the ELF symbol name.

All export names for functions and variables refer to the input ELF symbols.

4 Toolchain

We will try to use as much of the existing publicly available and open source ARM cross-compile build system as possible. We only need to build two tools: a library stubs generator and a ELF to SCE ELF converter. That way, the build system is agnostic of compiler (GCC, clang, etc) and host platform. The tools can also be integrated into a Makefile build process by including vita-elf-create as the last step in producing a PS Vita™ executable. The only requirement for these tools is that they work across Linux, OSX, and Windows.

4.1 vita-libs-gen

For each library as defined by the JSON NID database, we will generate a static object archive with the name of the SCE library. For example, "SceLibKernel" was defined in our JSON database so we produce "libSceLibKernel.a".

The contents of each library is a collection of object files, one for each exporting library. These object files are assembled from assembly code generated by this tool. For example, the entry for "SceLibGcc" will generate "SceLibGcc.S" which gets assembled into "SceLibGcc.o".

The assembly code for each exported library contains an exported symbol for each exported function or variable. The functions/variables will be placed into a section defined as `.vitalink.fstubs` (for functions) or `.vitalink.vstubs` (for variables) so the vita-elf-create tool can find it. For each symbol that is exported, we store three integers: the module NID, the library NID, and the function/variable NID in place of any actual code.

Below is an example of the assembly code generated for "SceLibKernel.S"

```
  .arch armv7-a
@ export functions
  .section  .vitalink.fstubs,"ax",%progbits
  .align  2
@ export sceKernelPuts
  .global sceKernelPuts
  .type sceKernelPuts, %function
sceKernelPuts:
  .word 0x49C42940
  .word 0xCAE9ACE6
  .word 0x023EAA62
  .align 4
@ export sceKernelGetThreadId
  .global sceKernelGetThreadId
  .type sceKernelGetThreadId, %function
sceKernelGetThreadId:
  .word 0x49C42940
  .word 0xCAE9ACE6
  .word 0x0FB972F9
```

```
  .align 4
@ export sceIoDevctl
  .global sceIoDevctl
  .type sceIoDevctl, %function
sceIoDevctl:
  .word 0x49C42940
  .word 0xCAE9ACE6
  .word 0x04B30CB2
  .align 4
@ ... export all other functions
@ export variables
  .section  .vitalink.vstubs,"awx",%progbits
  .align  2
@ export SceKernelStackGuard
  .global SceKernelStackGuard
  .type SceKernelStackGuard, %object
SceKernelStackGuard:
  .word 0x49C42940
  .word 0xCAE9ACE6
  .word 0x4458BCF3
  .align 4
@ ... export all other variables
```

4.2 vita-elf-create

The purpose of this tool is to convert an executable ELF linked with the static libraries generated by vita-libs-gen and produce a SCE ELF. An optional YAML export configuration can be passed in to specify export information.

1. Read the `.vitalink.fstubs` and `.vitalink.vstubs` sections of the input ELF. Build a list of imports from each library required.

2. Create the `.sceModuleInfo.rodata` section by generating a module info for the input ELF.

3. Create the export tables with the optional YAML configuration.

4. Create the import tables with the list from step 1 and the NID JSON database.

5. Convert all non-supported relocations to a supported type (optionally, if the linker was patched to only produce supported relocation types, we can skip this)

6. Open a new ELF with type `ET_SCE_EXEC` or `ET_SCE_RELEXEC` for writing.

7. Build the output SCE ELF by copying over the first loadable program segment and then writing all the module info, export, and import data to the end of the segment, extending the size of the segment. Make sure the offsets and pointers in the SCE sections are updated to match its new location.

8. Update `p_paddr` of the first segment to point to the module info (for `ET_SCE_EXEC` types) or `e_entry` to point to the module info (for `ET_SCE_RELEXEC` types).

9. Write import stubs over the temporary entries in `.vitalink.fstubs` and `.vitalink.vstubs`.

10. Next copy over the other program segments (if needed).

11. Finally create a new program segment of type `PT_SCE_RELA` and create SCE relocation entries based on the ELF relocation entries of the input ELF.

4.3 vita-elf-export

This tool, along with vita-elf-create allows the developer to create shared modules (`suprx`) and kernel modules (`skprx`). Once the SCE ELF is created, it can be distributed along with a JSON NID import database (generated by this tool), stubs, and user defined header files. This tool converts the YAML export configuration into a JSON import database. It's main task is creating the NIDs through the defined algorithm and then generating the JSON database.

The full flow for creating a user or kernel module is as follows

1. Build the module with vita-elf-create along with the export YAML configuration.

2. Build the JSON NID import database with vita-elf-export on the same YAML configuration.

3. Build the stubs with vita-libs-gen on the JSON import database generated from step 2.

4. Package the user created header files along with the results of step 1, 2, and 3 for distribution.

To use a user shared module in another application of module

1. Compile the code and link with the provided stubs.

2. Create the application with vita-elf-create and the additional JSON NID database.

3. Distribute the application along with the shared module if needed.

References

[1] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2* https://refspecs.linuxbase.org/elf/elf.pdf

[2] *Executable and Linkable Format (ELF)* http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf

[3] *ARM IHI 0044E: ELF for the ARM Architecture* http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044e/IHI0044E_aaelf.pdf