NEC

System V Application Binary Interface

VE Architecture Processor Supplement

SX-Aurora TSUBASA

Contents

Chapter1　Introduction

This document describe Application Binary Interface (ABI) for VE architecture.

Remarks

- All product, brand, or trade names in this publication are the trademarks or registered trademarks of their respective owners.

Chapter2　Software Installation

This document does not specify how software must be installed on a VE architecture machine.

Chapter3 Low Level System Information

3.1 Machine Interface

3.1.1 Processor Architecture

3.1.2 Data Representation

Within this specification, the term *byte* refers to an 8-bit object, the term *twobyte* refers to a 16-bit object, the term *fourbyte* refers to a 32-bit object, the term *eightbyte* refers to a 64-bit object, and the term *sixteenbyte* refers to a 128-bit object.

3.1.2.1 Fundamental Types

Table 3-1 shows the correspondence between ISO C's scalar types and the processor's.
A null pointer (for all types) has the value zero.
The type `size_t` is defined as unsigned long.
Booleans, when stored in a memory object, are stored as single byte objects the value of which is always 0 (`false`) or 1 (`true`). When stored in registers, all 8 bytes of the register are significant; any nonzero value is considered `true`.
Integral types which are shorter than 8 bytes, when stored in a memory object, are stored as their appropriate size. When stored in registers, all 8 bytes of the register are significant; when signed types, the sign should be extended upper area and when unsigned types, zero should be filled into upper area.

Table 3-1 Scalar Types

Type	**C**	**Sizeof**	**Alignment (bytes)**	**VE Architecture**
Integral	_Bool	1	1	Boolean
	char signed char	1	1	Signed byte
	unsigned char	1	1	Unsigned byte
	short signed short	2	2	Signed twobyte
	unsigned short	2	2	Unsigned twobyte
	int signed int enum	4	4	Signed fourbyte

Type	C	Sizeof	Alignment (bytes)	VE Architecture
	unsigned int	4	4	Unsigned fourbyte
	long signed long long long signed long long	8	8	Signed eightbyte
	unsigned long unsigned long long	8	8	Unsigned eightbyte
Pointer	any-type * any-type (*)()	8	8	Unsigned eightbyte
Floating-point	float	4	4	Single (IEEE-754)
	double	8	8	Double (IEEE-754)
	long double	16	16	128-bit extended (IEEE-754)

Aggregates and Unions

Structures and unions assume the alignment of their most strictly aligned component. Each member is assigned to the lowest available offset with the appropriate alignment. Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding are undefined.

3.1.2.2 Bit-Fields

C struct and union definitions may include bit-fields that define integral values of a specified size.

Bit-field Type	Width w	Range
signed char char unsigned char	1 to 8	-2^{w-1} to $2^{w-1} - 1$ -2^{w-1} to $2^{w-1} - 1$ 0 to $2^{w} - 1$
signed short short unsigned short	1 to 16	-2^{w-1} to $2^{w-1} - 1$ -2^{w-1} to $2^{w-1} - 1$ 0 to $2^{w} - 1$
signed int int unsigned int	1 to 32	-2^{w-1} to $2^{w-1} - 1$ -2^{w-1} to $2^{w-1} - 1$ 0 to $2^{w} - 1$
singed long	1 to 64	-2^{w-1} to $2^{w-1} - 1$

Bit-field Type	Width w	Range
long		-2^{w-1} to $2^{w-1} - 1$
unsigned long		0 to 2^{w} - 1

Bit-fields obey the same size and alignment rules as other structure and union members.

Also:

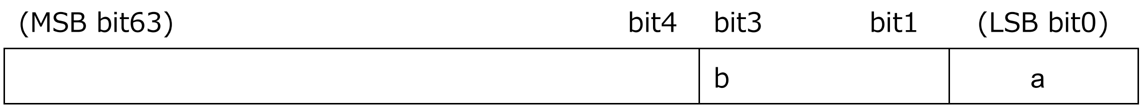
- Bit-fields are allocated from LSB to MSB (right to left)
- Bit-fields must be contained in a storage unit appropriate for its declared type
- Bit-fields may share a storage unit with other struct/union members

Unnamed bit-fields' types do not affect the alignment of a structure or union.

Figure 3-1-1 Example of Bit-filed

```
struct {
  unsigned int a : 1;
  unsigned int b : 3;
} status2;
```

Figure 3-1-2 Allocated image of Bit-filed



3.2 Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, parameter passing and so on.

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions. Nevertheless, it is recommended that all functions use the standard calling sequence when possible.

3.2.1 Registers and the Stack Frame

The VE architecture provides 64 scalar 64-bit registers (`%s0 - %s63`). In addition, the architecture provides 64 vector registers (`%v0 - %v63`), each 64-bit wide 256 elements, and 16 vector mask registers (`%vm0 - %vm15`), each 256-bit wide. All of these registers are global to all functions active for a given thread.

This subsection discusses usage of each register. Registers `%s18` through `%s33` “belong” to the calling function and the called function is required to preserve their values. In other words, a called function must preserve these registers’ values for its caller. Remaining registers “belong” to the called function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame. In addition to scalar registers, all vector registers and vector mask registers “belong” to the called function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

Register	Alias	Usage	Preserved across function calls
%s0-%s7		Used to pass 1st to 8th arguments to functions; return registers	No
%s8	%sl	Stack limit	Yes
%s9	%fp	Frame pointer	Yes
%s10	%lr	Link register, used for pointing return address of calling function	No
%s11	%sp	stack pointer	Yes
%s12		Outer register, used for pointing start address of called function	No
%s13		Used to pass identification of function to dynamic linker	No
%s14	%tp	Thread pointer	Yes
%s15	%got	Global Offset Table register	Yes
%s16	%plt	Procedure Linkage Table register	Yes
%s17		Linkage-area register, used for pointing	Yes

Register	Alias	Usage	Preserved across function calls
		linkage-area	
%s18-%s33		Callee-saved registers	Yes
%s34-%s63		Temporary registers	No
%v0-%v63		Vector registers	No
%vm0		Vector mask register	Inalterable
%vm1-%vm15		Vector mask registers	No

3.2.2 The Stack Frame

In addition to registers, each function has a frame on a run-time stack. This stack grows downwards from high addresses. The top address of stack needs to be a multiple of 16 and the stack must be aligned on 16 byte boundary.

The stack pointer, `%sp`, always points to the end of the latest allocated stack frame.

Position	Contents	Frame
	Locals and Temporaries	caller
176(%fp)	Parameter Area for callee	
19*8+16(%fp)	Register Save Area (RSA) for callee	
16(%fp)		
8(%fp)	return address	
0(%fp)	Frame pointer of callee	
	Local and Temporaries (for callee)	callee
	Parameter Area	
	Register Save Area (RSA)	
	return address	
0(%sp)	Frame pointer	

The detail of Register Save Area (RSA) is following:

Position	Contents	
%fp+168	Largest number of callee-saved register (`%s33`)	Higher address
	...	
%fp+48	Smallest number of callee-saved register (`%s18`)	
%fp+40	Linkage Area Register (`%s17`)	
%fp+32	Procedure Linkage Table Register (`%plt`)	
%fp+24	Global Offset Table Register (`%got`)	
%fp+16	Thread Pointer Register (`%tp`)	Lower Address

3.2.3 Parameter Passing

After the argument values have been computed, they are placed both in registers and pushed on the stack. The way how values are passed is described in the following sections.

Definitions We first define a number of classes to classify arguments. The classes are corresponding to VE register classes and defined as:

REGISTER	This class consists of basic types that fit into one of the general purpose registers.
REFERENCE	This class consists of types that will be passed and returned by memory.
BOTH	This class consists of types that will be passed in both registers and memory.

Classification The size of each argument gets rounded up to 8 bytes in the same manner described in 3.1.2.1. The basic types are assigned their natural classes:

- Arguments of types (signed and unsigned) _Bool, char, short, int, long, long long, _Imaginary and pointers are in the REGISTER class.
- Arguments of types float and double are in the REGISTER class.
- Arguments of types long double are split into two halves. The both are in the REGISTER class.
- Arguments of types T _Complex where T is one of types float or double are split into real and imaginary parts. The both are in the REGISTER class.

- Arguments of types long double _Complex are split into real and imaginary parts and then each part are split into two halves. The four parts are in the REGISTER class.
- The classification of aggregate (structures and arrays) and union types are in the REFERENCE class.
- The classification of arguments of variadic (variable number of arguments) and prototype-less functions are in the BOTH class.

Passing Once arguments are classified, the registers get assigned (in left-to-right) for passing as follows:

(1) If the class is REFERENCE, the caller provides temporary space on its stack to hold its value and assign the next available register of the sequence `%s0-%s7` to hold its address.

(2) If the class is REGISTER, the next available register of the sequence `%s0-%s7` is used.

(3) If the class is BOTH, the next available register of the sequence `%s0-%s7` is used and its value is stored in the parameter area on the stack.

If there are no registers available for arguments, the remaining arguments are passed by the parameter area on the stack.

The argument data is stored into responsible parameter area as the same data format as in register. It means:

- When signed integral value, sign-extended 8-byte value is stored
- When unsigned integral value, stored zero-extended 8-byte value is stored
- When single floating-point value, trailing 4-byte is zero-padded.

Following examples shows how arguments are passed in more details. These examples shows the argument passing in case of "BOTH" if not otherwise specified. In case of "REGISTER", the arguments from 1st to 8th are accessible by only registers.

Example 1. Basic case

```
void func(int a, short b, char c, unsigned int d, unsigned short e,
unsigned char f, float g, void *h, long i, double j);
```

Register	Argument Value		Access by Caller	Access by Callee
none	j		%sp+176+8*9	%fp+176+8*9
none	i		%sp+176+8*8	%fp+176+8*8
s7	h		%sp+176+8*7	%fp+176+8*7
s6	g	zero	%sp+176+8*6	%fp+176+8*6
s5	zero	f	%sp+176+8*5	%fp+176+8*5
s4	zero	e	%sp+176+8*4	%fp+176+8*4
s3	zero	d	%sp+176+8*3	%fp+176+8*3
s2	sign	c	%sp+176+8*2	%fp+176+8*2
s1	sign	b	%sp+176+8*1	%fp+176+8*1
s0	sign	a	%sp+176+8*0	%fp+176+8*0
	MSB	LSB		

In case of some types, special handling is necessary.

When the argument type is `float _Complex` or `double _Complex`, two consecutive registers are used. The younger number register holds real part and elder number register holds the imaginary part.

When the argument type is `long double` or `long double _Imaginary`, two consecutive registers which starts with ever number register are used. If the next available register number is odd, then this odd number register is skipped. The even number register holds upper part and odd number register holds lower part. When they are passed by the parameter area on the stack, upper part is stored in higher address and lower part is stored in lower address.

When the argument type is `long double _Complex`, four consecutive registers which starts with even number register are used. If the next available register number is odd, then this odd number register is skipped. The first even number register holds upper part of real part, the first odd number register holds lower part of real part, the second even number register holds upper part of imaginary part and the second odd number register holds lower part of imaginary part. When they are passed by the parameter area on the stack, upper part is stored in higher address and lower part is stored in lower address respectively.

Example 2. Special cases 1

```
void func(struct tag a, long double b, double _Complex c, float
_Complex d);
```

Register	Argument Value		Access by Caller	Access by Callee
s7	Imag. part of d	zero	%sp+176+8*7	%fp+176+8*7
s6	Real part of d	zero	%sp+176+8*6	%fp+176+8*6
s5	Imag. part of c		%sp+176+8*5	%fp+176+8*5
s4	Real part of c		%sp+176+8*4	%fp+176+8*4
s3	Lower part of b		**%sp+176+8*2**	**%fp+176+8*2**
s2	Upper part of b		**%sp+176+8*3**	**%fp+176+8*3**
s1	N/A		%sp+176+8*1	%fp+176+8*1
s0	Address of the copy of a		%sp+176+8*0	%fp+176+8*0
	MSB	LSB		

Example 3. Special cases 2

```
void func(long double _Complex a);
```

Register	Argument Value		Access by Caller	Access by Callee
s3	Lower part of Imag. a		**%sp+176+8*2**	**%fp+176+8*2**
s2	Upper part of Imag. a		**%sp+176+8*3**	**%fp+176+8*3**
s1	Lower part of Real a		**%sp+176+8*0**	**%fp+176+8*0**
s0	Upper part of Real a		**%sp+176+8*1**	**%fp+176+8*1**
	MSB	LSB		

Returning of Values The returning of values is done according to the following algorithm:

(1) Classify the return type with the classification algorithm except BOTH.

(2) If the type has class REFERENCE, then the caller provides space for the return value on its stack and passes the address of this storage in `%s0` as if it were the first argument to the function. In effect, this address becomes a "hidden" first argument. On return `%s0` will contain the address where return values are stored.

(3) If the class is REGISTER, the next available register of the sequence `%s0`-`%s7` is used.

Once return values are classified, the registers get assigned for passing are as as follows:

(1) When the return type is `long double` or `long double _Imaginary`, upper part is hold by `%s0` and lower part is hold by `%s1`.

(2) When the return type is `float _Complex` or `double _Complex`, real part is hold by `%s0` and imaginary part is hold by `%s1`.

(3) When the return type is `long double _Complex`, upper part of real part is hold by `%s0`, lower part of real part is hold by `%s1`, upper part of imaginary part is hold by `%s2` and lower part of imaginary part is hold by `%s3`

(4) When the return type is struct or union

- A return value is stored to the caller provided space which address is passed by “hidden” first argument.
- The address of this storage is hold by %s0.

(5) Other than the above is hold by `%s0`.

Example 4. Return aggregate or union type

```
struct foo func(long a, double b);
```

Register	**Argument Value**	**Access by Caller**	**Access by Callee**
s2	b	%sp+176+8*2	%fp+176+8*2
s1	a	%sp+176+8*1	%fp+176+8*1
s0	Address of return value area	%sp+176+8*0	%fp+176+8*0
	MSB LSB		

3.3 Coding Examples

This section discusses example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. The information here illustrates how operations may be done, not how they must be done.

Examples use the ANSI C language. Other programming languages may use the same conventions displayed below, but failure to do so does not prevent a program from conforming to the ABI. Two main object code models are available.

- Absolute code. Instructions can hold absolute addresses under this model. To execute properly, the program must be loaded at a specific virtual address, making the program's absolute addresses coincide with the process's virtual addresses.
- Position-independent code. Instructions under this model hold relative addresses, not absolute addresses. Consequently, the code is not tied to a specific load address, allowing it to execute properly at various positions in virtual memory.

Following sections describe the differences between these models. Code sequences for the models (when different) appear together, allowing easier comparison.

Note

- Examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences nor to reproduce compiler output.
- When other sections of this document show assembly language code sequences, they typically show only the absolute versions. Information in this section explains how position-independent code would alter the examples.

3.3.1 **Architectural Constraints**

When the system creates a process image, the executable file portion of the process has fixed addresses, and the system chooses shared object library virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared objects conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without having to change the segment images. Thus multiple processes can share a single shared object text segment, even though the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques.

- Control transfer instructions hold addresses relative to the instruction counter (IC). An IC-relative branch computes its destination address in terms of the current IC, not relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in the instructions, the compiler generates code to calculate an absolute address during execution.

Because the VE architecture provides IC-relative call and branch instructions, compilers can satisfy the first condition easily.

A global offset table provides information for address calculation. Position independent object files (executable and shared object files) have this table in their data segment. When the system creates the memory image for an object file, the table entries are relocated to reflect the absolute virtual addresses as assigned for an individual process. Because data segments are private for each process, the table entries can change-unlike text segments, which multiple processes share.

3.3.2 Conventions

In this document some special assembler symbols are used in the coding examples and discussion. They are:

- name@HI: specifies upper 32-bit of the address of the symbol name.
- name@LO: specifies lower 32-bit of the address of the symbol name.
- name@GOT32: specifies the offset to the GOT entry for the symbol name from the base of GOT.
- name@GOT_HI: specifies upper 32-bit of the offset to the GOT entry for the symbol name from the base of GOT.
- name@GOT_LO: specifies lower 32-bit of the offset to the GOT entry for the symbol name from the base of GOT.
- name@GOTOFF32: specifies the offset to the symbol name from the base of GOT.
- name@GOTOFF_HI: specifies upper 32-bit of the offset to the symbol name from the base of GOT.
- name@GOTOFF_LO: specifies lower 32-bit of the offset to the symbol name from the base of GOT.

- name@PLT32: specifies the offset to the PLT entry for the symbol name from the current code location.
- name@PLT_HI: specifies upper 32-bit of the offset to the PLT entry for the symbol name from the current location.
- name@PLT_LO: specifies lower 32-bit of the offset to the PLT entry for the symbol name from the current location.
- name@PC_HI: specifies upper 32-bit of the offset to the symbol name from the current location.
- name@PC_LO: specifies lower 32-bit of the offset to the symbol name from the current location.
- name@CALL_HI: specifies upper 32-bit of the address of the symbol name.
- name@CALL_LO: specifies lower 32-bit of the address of the symbol name.
- _GLOBAL_OFFSET_TABLE_: specifies the address of the base of GOT.
- _PROCEDURE_LINKAGE_TABLE_: specifies the address of the base of PLT.

3.3.3 Function Prologue and Epilogue

This example shows the codes when all callee-saved registers are used in this function. If only some of them are used, you should store/restore only ones you use. These storing/restoring codes may be at any place on the condition that the value of callee-saved register should be stored in RSA and it should be restore to callee-saved register before returning function.

Figure 3-1 Function Prologue and Epilogue

C	Assembly
int func(int a,int b,int c)	`        .text`
{	`        .globl   func`
…	`        .type    func,@function`
}	`func:`
	`        st     %fp,0x0(,%sp)`
	`        st     %lr,0x8(,%sp)`
	`        st   %got,0x18(,%sp)`
	`        st   %plt,0x20(,%sp)`
	`        or     %fp,0,%sp`
	`        …`
	`        # when callee-saved register will be used,`
	`        # its value must be stored in RSA (if necessary)`

C	Assembly
	```
	    …
	    lea  %s13, (*needed stack size for func*)&0xffffffff
	    and %s13,%s13,(32)0
	    lea.sl %sp, (*needed stack size for*
	*func)*>>32(%fp,%s13)
	    brge.l.t   %sp,%sl,.L1.EoP
	    ld   %s61,0x18(,%tp)  # load param area
	    or   %s62,0,%s0    # spill the value of %s0
	    lea  %s63,0x13b    # syscall # of grow
	    shm.l  %s63,0x0(%s61)# stored at addr:0
	    shm.l  %sl,0x8(%s61)# old limit at addr:8
	    shm.l  %sp,0x10(%s61)# new limit at addr:16
	    monc
	    or   %s0,0,%s62    # restore the value of %s0
	.L1.EoP:
	    <operations of func>
	    # the values of arguments are accessible via:
	    # a : %s0, b : %s1, c : %s2
	    …
	    or   %s0,0, <*return value*>
	    …
	    #  only if callee-saved registers are stored at
	    # prologue, its value must be restored in register
	    …
	    or    %sp,0,%fp
	    ld   %got,0x18(,%sp)
	    ld   %plt,0x20(,%sp)
	    ld     %lr,0x8(,%sp)
	    ld     %fp,0x0(,%sp)
	    b.l     (,%lr)
	```

3.3.4 Position-Independent Function Prologue

This section describes the function prologue for position-independent code. A function's prologue sets register `%got` to the global offset table's address. When the global function is being called, a function's prologue sets register `%plt` to the procedure linkage table's address. Because `%got` and `%plt` are private for each function and preserved across function calls, a function calculates its value once at the entry.

C	Assembly
int func(int a,int b,int c) { …	.text .globl func .type func,@function

C	Assembly

```
}                 func:
                      st     %fp,0x0(,%sp)
                      st     %lr,0x8(,%sp)
                      st   %got,0x18(,%sp)
                      st   %plt,0x20(,%sp)
                      lea %got,_GLOBAL_OFFSET_TABLE_@PC_LO(-24)
                      and   %got,%got,(32)0
                      sic   %plt

                  lea.sl %got,_GLOBA_OFFSET_TABLE_@PC_HI(%got,%plt
                  )
                      or     %fp,0,%sp
                      …
                      # when callee-saved register will be used,
                      # its value must be stored in RSA (if necessary)
                      …
                      lea  %s13,(needed stack size for func)&0xffffffff
                      and %s13,%s13,(32)0
                      lea.sl %sp, (needed stack size for
                  func)>>32(%fp,%s13)
                      brge.l.t   %sp,%sl,.L1.EoP
                      ld   %s61,0x18(,%tp)  # load param area
                  or   %s62,0,%s0    # spill the value of %s0
                      lea  %s63,0x13b    # syscall # of grow
                      shm.l  %s63,0x0(%s61) # stored at addr:0
                      shm.l  %sl,0x8(%s61)  # old limit at addr:8
                      shm.l  %sp,0x10(%s61) # new limit at addr:16
                      monc
                      or   %s0,0,%s62    # restore the value of %s0
                  .L1.EoP:
                      <operations of func>
                      # the values of arguments are accessible via:
                      # a : %s0, b : %s1, c : %s2
                      …
                      or   %s0,0, <return value>
                      …
                      #  only if callee-saved registers are stored at
                      # prologue, its value must be restored in register
                      …
                      or     %sp,0,%fp
                      ld   %got,0x18(,%sp)
                      ld   %plt,0x20(,%sp)
                      ld     %lr,0x8(,%sp)
                      ld     %fp,0x0(,%sp)
```

C	Assembly
	b.l　(,%lr)

3.3.5　Data Objects

This section describes only objects with static storage. Stack-resident objects are excluded since programs always compute their virtual address relative to the stack or frame pointers.
Because VE instructions cannot hold 64-bit addresses directly, a program normally computes an address into a register and accesses memory through the register.

Figure 3-2 Access to the data in absolute code

C	Assembly
extern int src;	
extern int dst;	
extern int *ptr;	
ptr = &dst;	lea　%s63,dst@LO
	and　%s63,%s63,(32)0
	lea.sl　%s63,dst@HI(,%s63)
	lea　%s62,ptr@LO
	and　%s62,%s62,(32)0
	lea.sl　%s62,ptr@HI(,%s62)
	st　%s63, (,%s62)
*ptr = src;	lea　%s60,src@LO
	and　%s60,%s60,(32)0
	lea.sl　%s60,src@HI(,%s60)
	ldl.sx　%s60,(,%s60)
	lea　%s59,ptr@LO
	and　%s59,%s59,(32)0
	lea.sl　%s59,ptr@HI(,%s59)
	ld　%s59,(,%s59)
	stl　%s60,(,%s59)

Position-independent code cannot contain absolute address. To access a global symbol the address of the symbol has to be loaded from the Global Offset Table.

Figure 3-3 Access to the data in position-independent code

C	Assembly
extern int src;	
extern int dst;	
extern int *ptr;	
ptr = &dst;	lea %s63,dst@GOT_LO
	and %s63,%s63,(32)0
	lea.sl %s63,dst@GOT_HI(%s63,%got)
	ld %s63,(,%s63)
	lea %s62,ptr@GOT_LO
	and %s62,%s62,(32)0
	lea.sl %s62,ptr@GOT_HI(%s62,%got)
	ld %s62,(,%s62)
	st %s63, (,%s62)
*ptr = src;	lea %s60,src@GOT_LO
	and %s60,%s60,(32)0
	lea.sl %s60,src@GOT_HI(%s60,%got)
	ld %s60,(,%s60)
	ldl.sx %s60,(,%s60)
	lea %s59,ptr@GOT_LO
	and %s59,%s59,(32)0
	lea.sl %s59,ptr@GOT_HI(%s59,%got)
	ld %s59,(,%s59)
	ld %s59,(,%s59)
	stl %s60,(,%s59)

Position-independent references to static data may be optimized. Because `%got` holds a known address, the global offset table, a program may use it as a base register.

Figure 3-4 Access to the static data in position-independent code

C	Assembly
static int src;	.local src
	.comm src,4,4
static int dst;	.local dsr
	.comm dst,4,4
static int *ptr;	.local ptr
	.comm ptr,8,8
ptr = &dst;	lea %s63,dst@GOTOFF_LO
	and %s63,%s63,(32)0
	lea.sl %s63,dst@GOTPFF_HI(%s63,%got)

C	Assembly
	lea %s62,ptr@GOTOFF_LO and %s62,%s62,(32)0 lea.sl %s62,ptr@GOTOFF_HI(%s62,%got) st %s63, (,%s62)
*ptr = src;	lea %s60,src@GOTOFF_LO and %s60,%s60,(32)0 lea.sl %s60,src@GOTOFF_HI(%s60,%got) ldl.sx %s60,(,%s60) lea %s59,ptr@GOTOFF_LO and %s59,%s59,(32)0 lea.sl %s59,ptr@GOTOFF_HI(%s59,%got) ld %s59,(,%s59) stl %s60,(,%s59)

3.3.6 Function Calls

Programs use the `bsic` instruction to make direct function calls. A `bsic` instruction's destination is an absolute address. Even when the code for a function resides in a shared object, the caller uses the same assembly language instruction sequence, although in that case control passes from the original call, through an indirection sequence, to the desired destination. See "5.1.4 Procedure Linkage Table" for more information on the indirection sequence.

Figure 3-5 Absolute Direct Function Call

C	Assembly
extern void function();	
function();	lea %s12,function@CALL_LO and %s12,%s12,(32)0 lea.sl %s12,function@CALL_HI(,%s12) bsic %lr,(,%s12)

Dynamic linking may redirect a function call outside the current object file's scope; So position-independent calls should use the procedure linkage table explicitly.

Figure 3-6 Position-Independent Direct Function Call

C	Assembly
extern void function();	

C	Assembly
function();	lea %s12,function@PLT_LO(-24) and %s12,%s12,(32)0 sic %s63 lea.sl %s12,function@PLT_HI(%s12,%s63) bsic %lr,(,%s12)

Indirect function calls use the indirect *bsic* instruction.

Figure 3-7 Absolute Indirect Function Call

C	Assembly

extern void (*ptr)();
extern void name();	

ptr = name;	lea %s63,name@LO and %s63,%s63,(32)0 lea.sl %s63,name@HI(,%s63) lea %s62,ptr@LO and %s62,%s62,(32)0 lea.sl %s62,ptr@HI(,%s62) st %s63,(,%s62)
(*ptr)();	lea %s61,ptr@LO and %s61,%s61,(32)0 lea.sl %s61,ptr@HI(,%s61) ld %s61,(,%s61) or %s12,%s61,(0)1 bsic %lr,(,%s12)

For position-independent code, the global offset table supplies absolute addresses for all required symbols, whether the symbols name objects or functions.

Figure 3-8 Position-Independent Indirect Function Call

C	Assembly

extern void (*ptr)();
extern void name();	

ptr = name;	lea %s63,name@GOT_LO and %s63,%s63,(32)0 lea.sl %s63,name@GOT_HI(%s63,%got)

C	Assembly
	ld %s63,(,%s63) lea %s62,ptr@GOT_LO and %s62,%s62,(32)0 lea.sl %s62,ptr@GOT_HI(%s62,%got) ld %s62,(,%s62) st %s63,(,%s62)
(*ptr)();	lea %s61,ptr@GOT_LO and %s61,%s61,(32)0 lea.sl %s61,ptr@GOT_HI(%s61,%got) ld %s61,(,%s61) ld %s61,(,%s61) or %s12,%s61,(0)1 bsic %lr,(,%s12)

The static function calls of position-independent call gets the absolute address from relative address of the symbol and the `sic` instruction.

Figure 3-9 Position-Independent Static Function Call

C	Assembly
static void function();	.type function,@function function:
function();	lea %s12,function@PC_LO(-24) and %s12,%s12,(32)0 sic %s63 lea.sl %s12,function@PC_HI(%s12,%s63) bsic %lr,(,%s12)

3.3.7 Branching

Programs use branch instructions to control their execution flow.

If the target addresses are within 2GB, no special care has to be taken when implementing branch instructions.

Figure 3-10 Branching Code within 2GB

C	Assembly
label: …	label: …

C	Assembly
goto label;	br.l label

If the target addresses are over 2GB, a branch target address is calculated. For absolute objects:

Figure 3-11 Absolute Branching Code over 2GB

C	Assembly
label: … goto label;	label: … lea %s63,label@LO and %s63,%s63,(32)0 lea.sl %s63,label@HI(,%s63) b.l (,%s63)

For position-independent objects:

Figure 3-12 Position-Independent Branching Code over 2GB

C	Assembly
label: … goto label;	label: … lea %s63,label@PC_LO(-24) and %s63,%s63,(32)0 sic %s62 lea.sl %s63,label@PC_HI(%s63,%s62) b.l (,%s63)

3.3.8 Variable Argument List

The full parameter list is known by the caller. So its save area must be ensured on the caller stack.

Example is not create yet.

3.3.9 Initialized static variables (global variables and static local variables)

Initialized static variables are allocated to the .data section

C	Assembly
int mos = 8;	.data

C	Assembly
	.balign 16 .global mos .type mos,@object .size mos,4 mos: .int 8

3.3.10 Uninitialized static variables (global variables and static local variables)

Uninitialized static variables are allocated to the .bss section using .comm.

C	Assembly
int mos;	.comm mos, 4, 4

3.4 DWARF Definition

This section defines the Debug With Arbitrary Record Format (DWARF) debugging format for the VE processor. The VE ABI does not define a debug format. However, all systems that do implement DWARF on VE shall use the following definitions.

DWARF is a specification developed for symbolic, source-level debugging. The debugging information format does not favor the design of any compiler or debugger. For more information on DWARF, see *DWARF Debugging Information Format*, Version 4, June 2010, DWARF Debugging Information Format Committee. It is available at: http://www.dwarfstd.org/.

3.4.1 DWARF Release Number

The DWARF definition requires some machine-specific definitions. The register number mapping needs to be specified for the VE registers.

3.4.2 DWARF Register Number Mapping

Table 3-2 outlines the register number mapping for the VE processor.

Table 3-2 DWARF Register Number Mapping

Register Name	Number	Abbreviation
Scalar Register 0-7	0-7	%s0-%s7
Stack Limit Register	8	%sl (or %s8)
Frame Pointer Register	9	%fp (or %s9)
Link Register	10	%lr (or %s10)
Stack Pointer Register	11	%sp (or %s11)
Outer Register	12	%s12
Scalar Register 13	13	%s13
Thread Pointer Register	14	%tp (or %s14)
Global Offset Table Register	15	%got (or %s15)
Procedure Linkage Table Register	16	%plt (or %s16)
Linkage-area Register	17	%s17
Scalar Register 18-63	18-63	%s18-%s63
Vector Register 0-63	64-127	%v0-%v63
Vector Mask Register 0-15	128-143	%vm0-%vm15

Chapter4　Object Files

4.1　ELF Header

4.1.1　Machine Information

For file identification in `e_ident`, the VE architecture requires the following values.

Table 4-1 Machine Information

Position	Value
e_ident[EI_CLASS]	ELFCLASS64
e_ident[EI_DATA]	ELFDATA2LSB

Processor identification resides in the ELF header's e_machine member and must have the `EM_VE`[1].

The ELF header's `e_flags` member holds bit flags associated with the file. The VE architecture defines no flags; so this member contains zero.

4.1.2　Number of Program Headers

The `e_phnum` member contains the number of entries in the program header table. The product of `e_phentsize` and `e_phnum` gives the table's size in bytes. If a file has no program header table, `e_phnum` holds the value zero.

If the number of program headers is greater than or equal to `PN_XNUM` (0xffff), this member has the value `PN_XNUM` (0xffff). The actual number of program header table entries is contained in the `sh_info` field of the section header at index 0. Otherwise, the `sh_info` member of the initial entry contains the value zero.

4.2　Sections

4.2.1　Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

[1] The value of this identifier is 251.

Table 4-2 Special sections

Name	Type	Attribute
.dynamic	SHT_DYNAMIC	SHF_ALLOC+SHF_WRITE
.got	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.plt	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR

Special sections are described below:

`.dynamic`	This section holds dynamic linking information.
`.got`	This section holds the global offset table. See “5.1.2 Global Offset Table” for more information.
`.plt`	This section holds the procedure linkage table. See “5.1.4 Procedure Linkage Table” for more information.

4.3 Symbol Table

4.3.1 Symbol Values

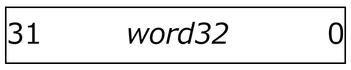
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for that file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This informs the dynamic linker that the symbol definition for that function is not contained in the executable file itself. If that symbol has been allocated a procedure linkage table entry in the executable file, and the st_value member for that symbol table entry is non-zero, the value will contain the virtual address of the first instruction of that procedure linkage table entry. Otherwise, the st_value member contains zero. This procedure linkage table entry address is used by the dynamic linker in resolving references to the address of the function. See “5.1.3 Function Addresses” for details.

4.4 Relocation

4.4.1 Relocation Types

Figure 4-1 shows the allowed relocatable fields.

Figure 4-1 Relocatable Fields



63	*word64*	0

word32	This specifies a 32-bit field occupying 4 bytes, the alignment of which is 4 bytes.
word64	This specifies a 64-bit field occupying 8 bytes, the alignment of which is 8 bytes.

The following notations are used for specifying relocations in **Table 4-3**.

A	Represents the addend used to compute the value of the relocatable field.
B	Represents the base address at which a shared object has been loaded into memory during execution. Generally, a shared object is built with a 0 base virtual address, but the execution address will be different.
G	Represents the offset into the global offset table at which the relocation entry's symbol will reside during execution.
GOT	Represents the address of the global offset table.
L	Represents the place (section offset or address) of the Procedure Linkage Table entry for a symbol.
P	Represents the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).
S	Represents the value of the symbol whose index resides in the relocation entry.

The VE ABI architectures uses only `Elf64_Rela` relocation entries with explicit addends. The `r_addend` member serves as the relocation addend.

Table 4-3 Relocation Types

Name	Value	Fields	Calculation
R_VE_NONE	0	none	none
R_VE_REFLONG	1	word32	S + A
R_VE_REFQUAD	2	word64	S + A
R_VE_SREL32	3	word32	S + A – P
R_VE_HI32	4	word32	(S + A) >> 32
R_VE_LO32	5	word32	(S + A) & 0xFFFFFFFF
R_VE_PC_HI32	6	word32	(S + A – P) >> 32
R_VE_PC_LO32	7	word32	(S + A – P) & 0xFFFFFFFF
R_VE_GOT32	8	word32	G + A
R_VE_GOT_HI32	9	word32	(G + A) >> 32
R_VE_GOT_LO32	10	word32	(G + A) & 0xFFFFFFFF
R_VE_GOTOFF32	11	word32	S + A – GOT

Name	Value	Fields	Calculation
R_VE_GOTOFF_HI32	12	word32	(S + A – GOT) >> 32
R_VE_GOTOFF_LO32	13	word32	(S + A – GOT) & 0xFFFFFFFF
R_VE_PLT32	14	word32	L + A – P
R_VE_PLT_HI32	15	word32	(L + A – P) >> 32
R_VE_PLT_LO32	16	word32	(L + A – P) & 0xFFFFFFFF
R_VE_RELATIVE	17	word64	B + A
R_VE_GLOB_DAT	18	word64	S
R_VE_JUMP_SLOT	19	word64	S
R_VE_COPY	20	-	-
R_VE_CALL_HI32	35	word32	(S + A) >> 32
R_VE_CALL_LO32	36	word32	(S + A) & 0xFFFFFFFF

Relocation types with special semantics are described below.

```
R_VE_HI32 / R_VE_CALL_HI32
```

This relocation type uses to compute the high order 32 bits of the symbol's address.

```
R_VE_LO32 / R_VE_CALL_LO32
```

This relocation type uses to compute the low order 32 bits of the symbol's address.

```
R_VE_PC_HI32
```

This relocation type uses to compute the high order 32 bits of the distance from the current code location to the location of the symbol.

```
R_VE_PC_LO32
```

This relocation type uses to compute the low order 32 bits of the distance from the current code location to the location of the symbol.

R_VE_GOT32[2]

This relocation type uses to compute the distance from the base of the global offset

[2] These have not been supported yet.

table to the symbol's global offset table entry. It additionally instructs the link editor to build a global offset table.

This relocation type can use when the distance from the base of the global offset table to the symbol's global offset table entry is in the range from -2147483648 bytes to 2147483647 bytes.

```
R_VE_GOT_HI32
```

This relocation type uses to compute the high order 32 bits of the distance from the base of the global offset table to the symbol's global offset table entry. It additionally instructs the link editor to build a global offset table.

```
R_VE_GOT_LO32
```

This relocation type uses to compute the low order 32 bits of the distance from the base of the global offset table to the symbol's global offset table entry. It additionally instructs the link editor to build a global offset table.

R_VE_GOTOFF32[3]

This relocation type uses to compute the distance from the base of the global offset table to the location of the symbol. It additionally instructs the link editor to build a global offset table.

This relocation type can use when the size of the data segment is less than 4294967296 bytes.

```
R_VE_GOTOFF_HI32
```

This relocation type uses to compute the high order 32 bits of the distance from the base of the global offset table to the location of the symbol. It additionally instructs the link editor to build a global offset table.

```
R_VE_GOTOFF_LO32
```

This relocation type uses to compute the low order 32 bits of the distance from the base of the global offset table to the location of the symbol. It additionally instructs the link editor to build a global offset table.

[3] These have not been supported yet.

```
R_VE_PLT32
```
[4]

This relocation type uses to compute the distance from the current code location to the symbol's procedure linkage table entry. It additionally instructs the link editor to build a procedure linkage table.

This relocation type can use when the size of the text segment is less than 4294967296 bytes.

```
R_VE_PLT_HI32
```

This relocation type uses to compute the high order 32 bits of the distance from the current code location to the symbol's procedure linkage table entry. It additionally instructs the link editor to build a procedure linkage table.

```
R_VE_PLT_LO32
```

This relocation type uses to compute the low order 32 bits of the distance from the current code location to the symbol's procedure linkage table entry. It additionally instructs the link editor to build a procedure linkage table.

```
R_VE_RELATIVE
```

The link editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

```
R_VE_GLOB_DAT
```

The link editor creates this relocation type for dynamic linking. This relocation type is used to set a global offset table entry to the address of the specified symbol. The special relocation type allows one to determine the correspondence between symbols and global offset table entries.

```
R_VE_JUMP_SLOT
```

The link editor creates this relocation type for dynamic linking. Its offset member refers

[4] These have not been supported yet.

to a global offset table entry to which a procedure linkage table entry is referring. The dynamic linker modifies a global offset table entry to transfer control to the designated symbol's address.

```
R_VE_COPY
```

The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset. The other objects referring to this symbol in the process refer to this copy.

Chapter5 Program Loading and Dynamic Linking

5.1 Dynamic Linking

5.1.1 Dynamic Section

Dynamic section entries give information to the dynamic linker.

```
DT_PLTGOT
```

This entry's d_ptr member gives the address of the first byte in the procedure linkage table.

5.1.2 Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries (See "4.4 Relocation"). After the system creates memory segments for a loadable object file, the dynamic linker processes the relocation entries, some of which will be type R_VE_GLOB_DAT referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The first entry (number zero) in the global offset table is reserved to hold the address

of the dynamic structure, referenced with the symbol `_DYNAMIC`. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image.
The second entry (number one) in the global offset table is reserved for storing the address to pass control to the dynamic linker (See "5.1.4 Procedure Linkage Table").
The global offset table exists in .got section. The symbol `_GLOBAL_OFFSET_TABLE_` can use to access the global offset table. The symbol `_GLOBAL_OFFSET_TABLE_` indicates the first entry (number zero) in the global offset table. The type of symbol is an array of `Elf64_Addr`.

Figure 5-1 Global Offset Table

```
extern Elf64_Addr _GLOBAL_OFFSET_TABLE_[];
```

The symbol `_GLOBAL_OFFSET_TABLE_` may reside in the middle of the `.got` section, allowing both negative and non-negative offsets into the array of addresses.

5.1.3 Function Addresses

References to the address of a function from an executable file and the shared objects associated with it might not resolve to the same value. References from within shared objects will normally be resolved by the dynamic linker to the virtual address of the function itself. References from within the executable file to a function defined in a shared object will normally be resolved by the link editor to the address of the procedure linkage table entry for that function within the executable file.
To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry (See "4.3.1 Symbol Values"). The dynamic linker treats such symbol table entries specially. If the dynamic linker is searching for a symbol, and encounters a symbol table entry for that symbol in the executable file, it normally follows the rules below.

(1) If the `st_shndx` member of the symbol table entry is not `SHN_UNDEF`, the dynamic linker has found a definition for the symbol and uses its `st_value` member as the symbol's address.

(2) If the `st_shndx` member is `SHN_UNDEF` and the symbol is of type `STT_FUNC` and the `st_value` member is not zero, the dynamic linker recognizes this entry as special and uses the `st_value` member as the symbol's address.

(3) Otherwise, the dynamic linker considers the symbol to be undefined within the executable file and continues processing.

Some relocations are associated with procedure linkage table entries. These entries are used for direct function calls rather than for references to function addresses. These relocations are not treated in the special way described above because the dynamic linker must not redirect procedure linkage table entries to point to themselves.

5.1.4 Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. Procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and shareability of the program's text. Executable files and shared object files have separate procedure linkage tables (See Figure 5-2, Figure 5-3).

Figure 5-2 Absolute Procedure Linkage Table Sample Entries

```
_PROCEDURE_LINKAGE_TABLE_:
 lea %s62, _GLOBAL_OFFSET_TABLE_@LO
 and %s62, %got, (32)0
 lea.sl %s62, _GLOBAL_OFFSET_TABLE_@HI(,%s62)
 ld  %s63, 8(,%s62)
 b.l.t  (,%s63)
.PLT1:
 lea %s13, func1@LO
 and %s13, %s13, (32)0
 lea.sl %s13, func1@HI(,%s13)
 ld  %s13, (,%s13)
 b.l.t  (,%s13)
 lea %s13, [index of relocation entry of symbol]
 br.l.t _PROCEDURE_LINKAGE_TABLE_
```

Figure 5-3 Position-Independent Procedure Linkage Table Sample Entries

```
_PROCEDURE_LINKAGE_TABLE_:
       or     %62,0,%got
ld  %s63, 8(,%got)
b.l.t   (,%s63)
.PLT1:
lea %s13, func1@GOT_LO
and %s13, %s13, (32)0
lea.sl  %s13, func1@GOT_HI(%s13,%got)
ld  %s13, (,%s13)
b.l.t   (,%s13)
lea %s13, [index of relocation entry of symbol]
br.l.t   _PROCEDURE_LINKAGE_TABLE_
```

Following the steps below, the dynamic linker and the program cooperate to resolve symbolic references through the procedure linkage table and the global offset table. Again, the steps described below are for explanation only. The precise execution-time behavior of the dynamic linker is not specified.

(1) When memory image of program is created first, the dynamic linker modifies memory image as follows.

- Set address to pass control of the dynamic linker to the second entry (number one) in the global offset table.
- As for symbols of the relocation type of R_VE_JUMP_SLOT, set value at offset location plus base address of shared object to the offset location of relocation entry.

(2) Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file.

(3) For illustration, assume the program calls func1, which transfers control to the label `.PLT1`.

Note that when object file is shared object, it's necessary to set the address of the first entry of the global offset table to the register `%s15` and to set the address of the first entry of procedure linkage table to the register `%S16` before calling the procedure linkage table entry.

(4) It jumps to the address in the global offset table entry for `func1`. Initially the global offset table holds the address of the following lea instruction, not the real

address of func1.

(5) It sets index of relocation entry to the register %s16. The index of relocation entry is non-negative index. The designated relocation entry will have type R_VE_JUMP_SLOT, and its offset will specify address of the global offset table entry. The relocation entry contains a symbol table index that will reference the appropriate symbol, func1 in the example.

(6) The program jumps to _PROCEDURE_LINKAGE_TABLE_, the first entry in the procedure linkage table. In this entry, the program jumps to the address in the second global offset table entry, which transfers control to the dynamic linker.

(7) When the dynamic linker receives control, looks at the designated relocation entry by the register %s16, finds the symbol's value, stores the "real" address for func1 in its global offset table entry, and transfers control to the desired destination.

(8) Subsequent executions of the procedure linkage table entry will transfer directly to func1, without calling the dynamic linker a second time. That is, the b.l.t instruction at .PLT1 will transfer to func1, instead of "falling through" to the lea instruction.

5.1.5 Program Interpreter

There is one valid program interpreter for programs conforming to the VE ABI:

```
/opt/nec/ve/lib/ld.so.1
```
[5]

5.1.6 Initialization and Termination Functions

The implementation is responsible for executing the initialization functions specified by DT_INIT, DT_INIT_ARRAY, and DT_PREINIT_ARRAY entries in the executable file and shared object files for a process, and the termination (or finalization) functions specified by DT_FINI and DT_FINI_ARRAY, as specified by the System V ABI. The user program plays no further part in executing the initialization and termination functions specified by these dynamic tags.

[5] This is provisional. This is currently under consideration by the OS group.

Chapter6　Conventions

6.1　C++

To be determined.

6.2　Fortran

To be determined.

6.3　Thread-Local Storage

The VE ABI related with Thread-Local Storage is described the separated document. See ELF Handling For Thread-Local Storage VE Architecture Processor Supplement.

Appendix A　Code Segment

A.1　.text

Position	Contents
	code area

VE require the .text to be 16-byte aligned. And each function has a frame on the run-time stack.

(1)　Code area are placed after 16 byte, it contains machine instructions.

Appendix B　Data Segment

B.1　.data

Position	Contents
.data	

VE require the .data to be 16-byte aligned and zero out.

B.2　.bss

Position	Contents
.bss	

VE require the .bss to be 16-byte aligned and zero cleared.

Appendix C History

C.1 History table

Feb. 2018	Rev. 1.0	Create a new entry.
Mar. 2018	Rev. 1.1	Remarks is added.
Oct. 2018	Rev. 2.0	The design of document is changed.
Dec. 2018	Rev. 2.1	CALL_HI/CALL_LO which is kind of relocation is added.

C.2 Change notes

The following changes are done in this edition.

- CALL_HI/CALL_LO which is kind of relocation is added.