ELF Application Binary Interface s390x Supplement

Version 1.6.1

Martin Schwidefsky Ulrich Weigand Andreas Arnez
Andreas Krebbel

February 6, 2024

IBM® Corporation

ELF Application Binary Interface s390x Supplement
Version 1.6.1

Contents

List of Figures

List of Tables

Listings

About This Book

The s390x supplement to the Executable and Linkage Format Application Binary Interface (or ELF ABI) defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on Linux® for z/Architecture® systems.

This book is a supplement to the generic "System V Application Binary Interface" and should be read in conjunction with it.

History

1.6.1 Minor release to pick up a few fixes. Published at https://github.com/ibm/s390x-abi, January 2024.

1.6 Describe vector types and registers (based on input from Andreas Krebbel); mention condition code and program mask; enhance exception handling information. Published at https://github.com/ibm/s390x-abi, November 2021.

1.5 *"ELF Application Binary Interface s390x Supplement"* – Conversion to LaTeX; various corrections to revision 1.02. Edited by Andreas Arnez. Published at https://github.com/ibm/s390x-abi, January 2021.

1.02 *"zSeries ELF Application Binary Interface Supplement"* – Revised edition. Published under the GNU Free Documentation License 1.1 by The Linux Foundation as a "referenced specification" at http://refspecs.linuxbase.org/, November 2002.

1.0 *"LINUX for zSeries: ELF Application Binary Interface Supplement"* – First edition. Published by IBM as LNUX-1107-01, March 2001.

1. Low-Level System Information

1.1. Machine Interface

This section describes the processor-specific information for z/Architecture processors.

1.1.1. Processor Architecture

[5] (SA22-7832) defines the z/Architecture.

Programs intended to execute directly on the processor use the z/Architecture instruction set and the instruction encoding and semantics of the architecture.

An application program can assume that all instructions defined by the architecture and that are neither privileged nor optional, exist and work as documented.

To be ABI conforming, the processor must implement the instructions of the architecture, perform the specified operations, and produce the expected results. The ABI neither places performance constraints on systems nor specifies what instructions must be implemented in hardware. A software emulation of the architecture can conform to the ABI.

In z/Architecture a processor runs in big-endian mode. (See section 1.1.2.1.)

1.1.2. Data Representation

1.1.2.1. Byte Ordering

The architecture defines an 8-bit byte, a 16-bit halfword, a 32-bit word, a 64-bit doubleword, and a 128-bit quadword. Byte ordering defines how the bytes that make up halfwords, words, doublewords, and quadwords are ordered in memory. Most significant byte (MSB) ordering, also called “big-endian,” means that the most significant byte of a structure is located in the lowest addressed byte position in a storage unit (byte 0). By contrast, least significant byte (LSB) ordering, or “little-endian,” refers to the reverse byte order, where the lowest addressed byte position holds the least significant byte.

Figures 1.1 to 1.4 illustrate the conventions for bit and byte numbering within storage units of various widths. These conventions apply to both integer data and floating-point data, where the most significant byte of a floating-point value holds the sign and the exponent (or at least the start of the exponent). The figures show big-endian byte numbers in the upper left corners and bit numbers in the lower corners.

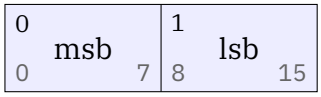

Figure 1.1.: Bit and byte numbering in halfwords

Figure 1.2.: Bit and byte numbering in words

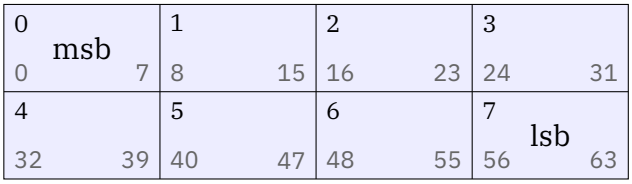

Figure 1.3.: Bit and byte numbering in doublewords

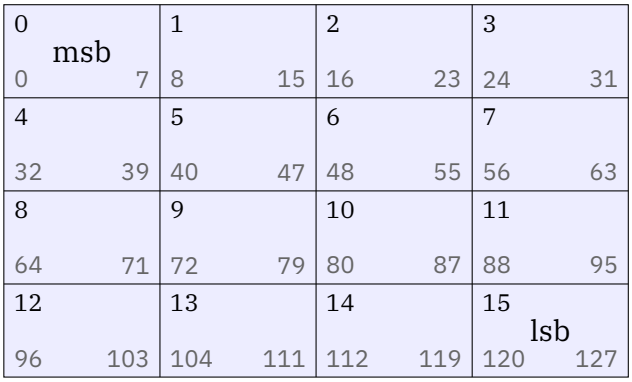

Figure 1.4.: Bit and byte numbering in quadwords

```
struct {
    char c;
};
```

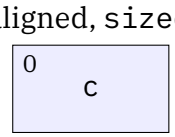

Figure 1.5.: Structure smaller than a word

1.1.2.2. Fundamental Types

Table 1.1 shows how ISO C scalar types correspond to those of a z/Architecture processor. To comply with this ABI, objects stored in memory must be aligned as indicated, even though the architecture permits unaligned storage operands for most instructions.

For all types, a null pointer has the value zero (binary).

A Boolean object is represented in memory as a single byte with a value of 0 or 1. If a byte with any other value is evaluated as a Boolean, the behavior is undefined.

For each binary floating-point type, there is a corresponding complex type. It is represented as a two-element array with the real part as its first and the imaginary part as its second element.

Some C dialects permit enumeration constants that exceed the range of an `int`. Then the enumeration type shall be encoded as the smallest unsigned or signed C integer type that can represent all of its enumeration constants and is not smaller than `int`.

1.1.2.3. Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component—that is, the component with the largest alignment. The size of any object, including aggregates and unions, is always a multiple of the alignment of the object. An array uses the same alignment as its elements. Structure and union objects may require padding to meet these size and alignment constraints:

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require internal padding, depending on the previous member.
- If necessary, a structure's size is increased to make it a multiple of the structure's alignment. This may require tail padding if the last member does not end on the appropriate boundary.

In the examples shown in figures 1.5 to 1.9, member byte offsets (for the big-endian implementation) appear in the upper left corners.

1.1.2.4. Bit-Fields

C struct and union definitions may have "bit-fields," defining integral objects with a specified number of bits (see table 1.2).

Type	ISO C	Size in bytes	Align-ment	z/Architecture type
Unsigned integer	`_Bool`	1	1	*n*-bit unsigned binary integer[†]
	`unsigned char`	1	1	
	`char`	1	1	
	`unsigned short`	2	2	
	`unsigned int`	4	4	
	`unsigned long`	8	8	
	`unsigned long long`	8	8	
	`unsigned __int128`[††]	16	8	
Signed integer	`signed char`	1	1	*n*-bit signed binary integer[†]
	`signed short`	2	2	
	`short`	2	2	
	`signed int`	4	4	
	`int`	4	4	
	`enum`	4	4	
	`signed long`	8	8	
	`long`	8	8	
	`signed long long`	8	8	
	`long long`	8	8	
	`__int128`[††]	16	8	
	`signed __int128`[††]	16	8	
Pointer	*any-type* `*`	8	8	64-bit address
	any-type `(*) ()`	8	8	
Binary floating-point	`float`	4	4	short BFP
	`double`	8	8	long BFP
	`long double`	16	8	extended BFP
Decimal floating-point	`_Decimal32`[††]	4	4	short DFP
	`_Decimal64`[††]	8	8	long DFP
	`_Decimal128`[††]	16	8	extended DFP

[†] Here *n* denotes the bit size, which equals the byte size multiplied by 8.
[††] These types are an extension to C (ISO/IEC 9899:2011).

Table 1.1.: Scalar types

```
struct {
    char c;
    char d;
    short s;
    int n;
};
```

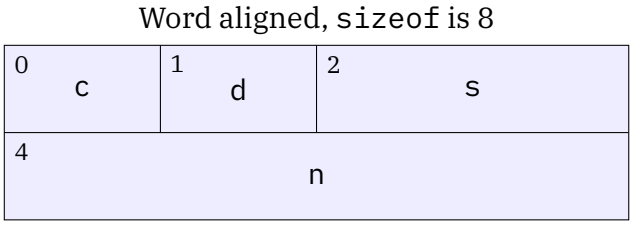

Figure 1.6.: No padding

```
struct {
    char c;
    short s;
};
```

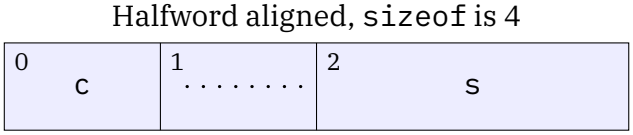

Figure 1.7.: Internal padding

```
struct {
    char c;
    double d;
    short s;
};
```

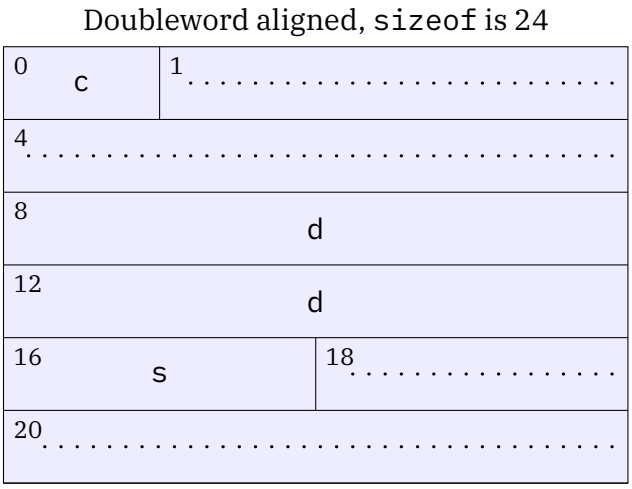

Figure 1.8.: Internal and tail padding

```
union {
    char c;
    short s;
    int j;
};
```

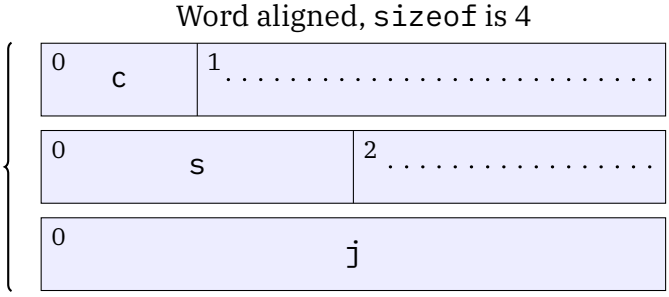

Figure 1.9.: Union padding

Bit-field type	Width n	Range
`signed char`		$-2^{n-1} \ldots 2^{n-1}-1$
`char`	1…8	$0 \ldots 2^{n}-1$
`unsigned char`		$0 \ldots 2^{n}-1$
`signed short`		$\text{-}2^{n-1} \ldots 2^{n-1}-1$
`short`	1…16	$\text{-}2^{n-1} \ldots 2^{n-1}-1$
`unsigned short`		$0 \ldots 2^{n}-1$
`signed int`		$-2^{n-1} \ldots 2^{n-1}-1$
`int`	1…32	$-2^{n-1} \ldots 2^{n-1}-1$
`unsigned int`		$0 \ldots 2^{n}-1$
`signed long`		$-2^{n-1} \ldots 2^{n-1}-1$
`long`	1…64	$-2^{n-1} \ldots 2^{n-1}-1$
`unsigned long`		$0 \ldots 2^{n}-1$
`signed long long`		$-2^{n-1} \ldots 2^{n-1}-1$
`long long`	1…64	$-2^{n-1} \ldots 2^{n-1}-1$
`unsigned long long`		$0 \ldots 2^{n}-1$

Table 1.2.: Bit-fields

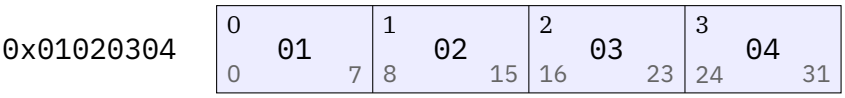

Figure 1.10.: Bit numbering

Bit-fields have the signedness of their underlying type. For example, a bit-field of type `long` is signed, whereas a bit-field of type `char` is unsigned.

Bit-fields obey the same size and alignment rules as other structure and union members, with the following additions:

- Bit-fields are allocated from left to right (most to least significant).
- A bit-field must entirely reside in a storage unit appropriate for its declared type. Thus, a bit-field never crosses its unit boundary.
- Bit-fields must share a storage unit with other structure and union members (either bit-field or non-bit-field) if and only if there is sufficient space within the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union, although an individual bit-field's member offsets obey the alignment constraints. An unnamed, zero-width bit-field shall prevent any further member, bit-field or other, from residing in the storage unit corresponding to the type of the zero-width bit-field.

The examples in figures 1.10 to 1.15 show structure and union member byte offsets in the upper left corners. Bit numbers appear in the lower corners.

```
struct {
     int j:5;
     int k:6;
     int m:7;
};
```

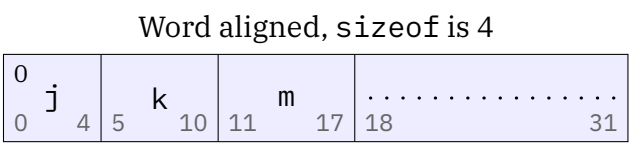

Figure 1.11.: Left-to-right allocation

```
struct {
     short s:9;
     int   j:9;
     char  c;
     short t:9;
     short u:9;
     char  d;
};
```

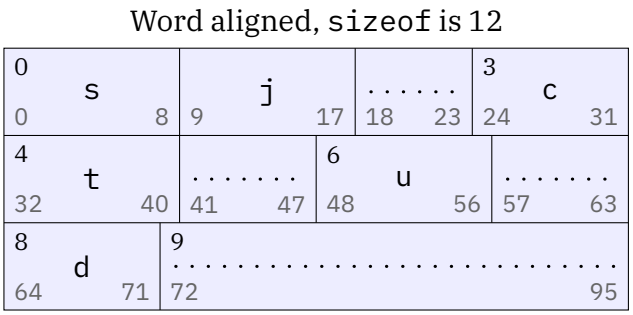

Figure 1.12.: Boundary alignment

```
struct {
     char  c;
     short s:8;
};
```

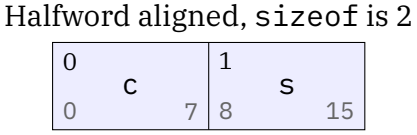

Figure 1.13.: Storage unit sharing

```
union {
     char  c;
     short s:8;
};
```

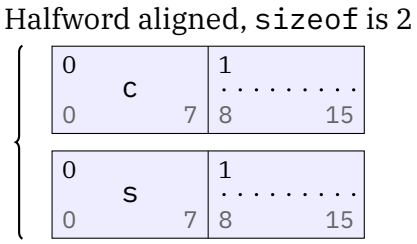

Figure 1.14.: Union allocation

```
struct {
     char  c;
     int   :0;
     char  d;
     short :9;
     char  e;
};
```

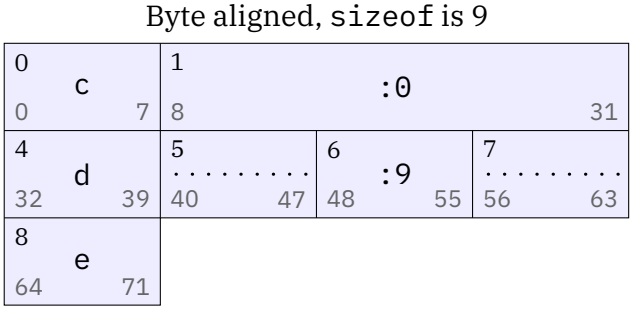

Figure 1.15.: Unnamed bit-fields

1.1.2.5. Vector Types

Vector types are used for SIMD (single-instruction, multiple-data) programming. They are not part of the C language, but defined by a language extension, such as the "vector extensions" described in the respective section in the GCC manual [4].

A vector holds multiple values ("elements") of a given base type ("element type"). Valid element types include the scalar types shown in table 1.1, except for pointer types and the Boolean type `_Bool`. The number of elements in a vector must be a power of two. Each allowed combination of base type and number of elements forms a distinct vector type. A single-element vector type is not compatible with its base type.

The size of a vector type is the size of the base type multiplied by the number of elements. Vectors with a size of 1, 2, 4, or ≥ 8 bytes are aligned to a 1-, 2-, 4-, or 8-byte boundary, respectively.

1.2. Function-Calling Sequence

This section discusses the standard function-calling sequence, including stack frame layout, register usage, and parameter passing.

1.2.1. Registers

The ABI makes the assumption that the processor has 16 general registers, `r0` through `r15`, and 16 floating-point registers, `f0` through `f15`. z/Architecture processors have these registers; each register is 64 bits wide.

Optionally, z/Architecture processors may have a vector facility installed, which extends the 64-bit floating-point registers to 128-bit vector registers, `v0` through `v15`, and provides 16 additional 128-bit vector registers, `v16` through `v31`.

In addition, the processor state includes 32-bit access registers `a0` through `a15`, a 2-bit condition code `cc`, a 4-bit program mask `pm`, and a 32-bit floating-point control register `fpc`.

1.2.1.1. Register Preservation Rules

Table 1.3 summarizes the roles of registers and their persistence across function calls. Registers marked as "saved" are also referred to as "nonvolatile"; they "belong" to the calling function and must retain their values over the function call. A called function modifying these registers must restore their original values before returning. By contrast, "volatile" registers need not be restored. To preserve such a register's value across the function call, the caller must take care of saving and restoring the value by itself. "Reserved" registers are reserved for system use and must not be modified at all.

Using these definitions, the registers are categorized as follows:

- Registers `r6` through `r13`, `r15`, and `f8` through `f15` are nonvolatile.
- Access registers `a0` and `a1` are reserved.

Register name	Role(s)	Call effect[†]
`r0`, `r1`	–	Volatile
`r2`	Argument / return value	Volatile
`r3`, `r4`, `r5`	Arguments	Volatile
`r6`	Argument	Saved
`r7`…`r11`	–	Saved
`r12`	(Commonly used as GOT pointer)	Saved
`r13`	(Commonly used as literal pool pointer)	Saved
`r14`	Return address	Volatile
`r15`	Stack pointer	Saved
`f0`	Argument / return value	Volatile
`f2`, `f4`, `f6`	Arguments	Volatile
`f1`, `f3`, `f5`, `f7`	–	Volatile
`f8`…`f15`	–	Saved
`v0`…`v7`	(Extend `f0`…`f7`)	Volatile
`v8`…`v15`	(Extend `f8`…`f15`)	Volatile[††]
`v16`…`v23`	–	Volatile
`v24`	Argument / return value	Volatile
`v25`…`v31`	Arguments	Volatile
`cc`	Condition code	Volatile
`pm`	Program mask	Cleared
`a0`, `a1`	Reserved for system use	Reserved
`a2`…`a15`	–	Volatile

[†] Volatile: These registers' values are not preserved across function calls.
Saved: These registers' values are preserved across function calls.
Cleared: These registers must be 0 before entering/leaving a function.
Reserved: These registers must not be modified by ABI-compliant functions.
[††] Except for bytes 0–7, which are aliased to `f8`…`f15`.

Table 1.3.: Register usage across function calls

- The left halves of vector registers `v8` through `v15` are nonvolatile, since they are aliased to `f8` through `f15`. The right halves are volatile.
- The program mask `pm` must be zero before entering and before leaving a function.
- All other registers are volatile.
- Furthermore the values in registers `r0` and `r1` may be altered by the interface code in cross-module calls, so a function cannot depend on the values in these registers having the same values that were placed in them by the caller.

1.2.1.2. Register Roles

The roles mentioned in table 1.3 have the following meaning:

Argument: When calling a function, such a register may hold an argument to that function, according to the parameter-passing rules defined in section 1.2.3.

Return value: When a called function returns, such a register may hold the return value of that function, according to the rules defined in section 1.2.5.

GOT pointer: Global Offset Table pointer. In a position-independent module, such a register may point to the start of that module's GOT, described in section 3.2.2. If instructions like "Load Relative" can be used, no GOT pointer may be needed.

Literal pool pointer: Some constant local data objects ("literals") can be encoded in the instructions themselves, using immediate values. Others are typically grouped into pools of such literals, in which case a register may be set up as a base pointer to such a pool.

Return address: When entering a function, this register, `r14`, contains the address of the instruction that the function must return to. Except at function entry, no special role is assigned to `r14`.

Stack pointer: This register, `r15`, always points to the lowest allocated valid stack frame. It shall maintain an 8-byte alignment. A function may decrement `r15` to allocate a new stack frame or to enlarge the current one. Before returning, `r15` must be restored to its original value. For more information about stack frames, see section 1.2.2.

1.2.1.3. Registers And Signal Handling

Signals can interrupt processes. Functions called during signal handling have no unusual restrictions on their use of registers. Moreover, if a signal-handling function returns, the process will resume its original execution path with all registers restored to their original values. Thus programs and compilers may freely use all registers listed above, except those reserved for system use, without the danger of signal handlers inadvertently changing their values.

1.2.1.4. Register Usage in Inline Assemblies

With these calling conventions, the following usage of the registers for inline assemblies is recommended:

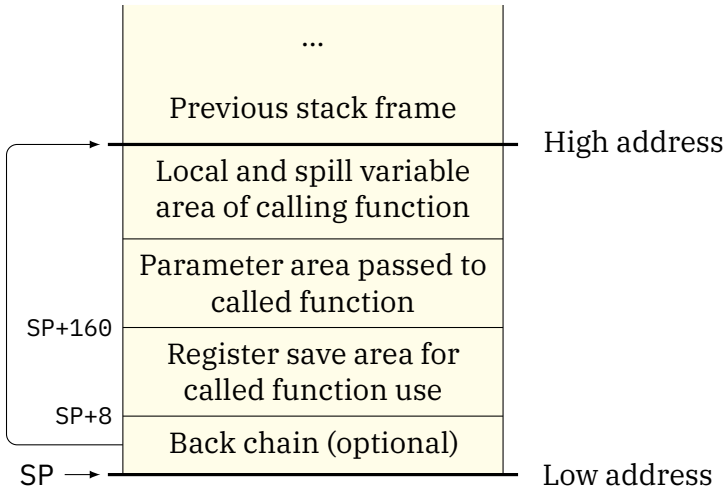

Figure 1.16.: Standard stack frame. SP denotes the value of r15 upon entering the called function.

- General registers r0 and r1 should be used internally whenever possible.
- General registers r2 to r5 should be second choice.
- General registers r12 to r15 should only be used for their standard function.

1.2.2. The Stack Frame

A function will be passed a frame on the runtime stack by the function which called it, and may allocate a new stack frame. A new stack frame is required if the called function will in turn call further functions (which must be passed the address of the new frame). The stack grows downward from high addresses. Each stack frame is aligned on an 8-byte boundary. General register r15 holds the stack pointer and always points to the first byte of the lowest allocated stack frame. Figure 1.16 shows the stack frame organization.

1.2.2.1. Back-Chain Slot

The first doubleword of a calling function's stack frame is preserved across function calls. It may be used for maintaining a back chain for stack unwinding, in which case it must hold the address of the previously allocated stack frame (toward higher addresses), or zero (NULL) if there is none.

Maintenance of the stack back chain is optional. If a function chooses to maintain the back chain, it should also store the values of r14 and r15 at function entry into the register save area, using their standard save slots as shown in figure 1.17.

1.2.2.2. Register Save Area

The first 160 bytes of a calling function's stack frame, excluding the initial doubleword, are referred to as the register save area. This area must be allocated by the caller and

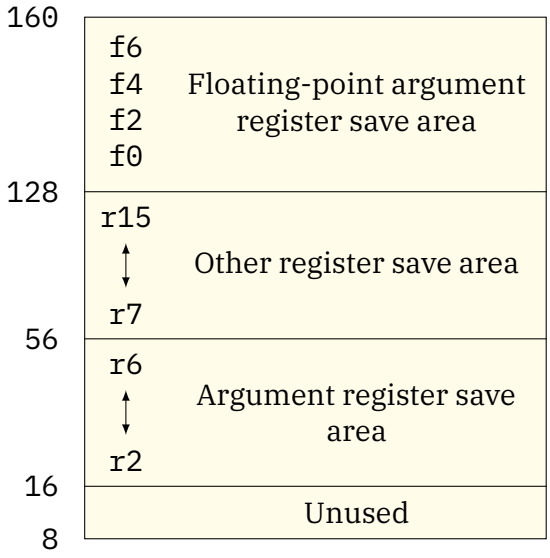

Figure 1.17.: Register save area usage example. The slots for `r2` through `r5` and for the floating-point argument registers are used when the called function receives varying arguments.

may be used by the called function in any way. For example, if the called function is going to modify any nonvolatile registers, it may use the register save area for saving these registers' original values first. It is customary to assign a standard save slot to each register, as shown in figure 1.17.

1.2.2.3. Parameter Area

The parameter area shall be allocated by a calling function if some parameters cannot be passed in registers, but must be passed on the stack instead (see section 1.2.3). This area starts at byte offset 160 of the calling function's stack frame and consists of as many 8-byte parameter slots as needed. The calling function cannot rely on the contents of these slots to be preserved across the function call.

1.2.2.4. Stack Frame Allocation

A function may allocate a new stack frame by decrementing the stack pointer by the size of the new frame. The stack pointer must be restored prior to return. By restoring the stack pointer, the allocated stack frame is deallocated and may not be accessed after that.

A new stack frame is required if the function calls further functions. Then the stack frame must at least contain the back chain slot, the register save area, and the parameter area (if needed). The remaining space in the stack frame is called the "local-variable area." It immediately follows the parameter area and can have arbitrary size, provided that it contains any padding necessary to make the entire frame a multiple of 8 bytes in length.

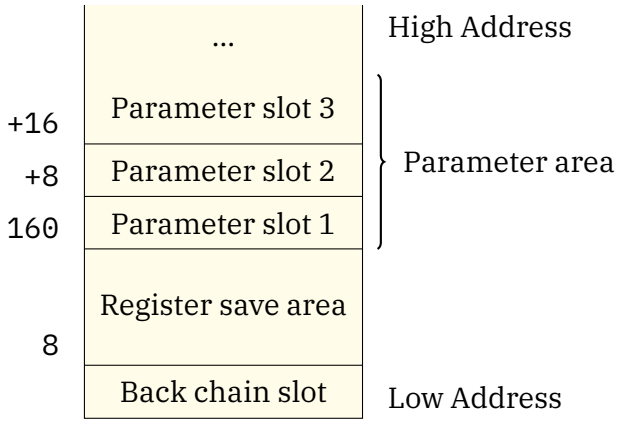

Figure 1.18.: Parameter area

If a function does not call any other functions and does not require more stack space than available in the register save area, it need not establish a stack frame.

1.2.3. Parameter Passing

Arguments to called functions are passed in registers. Since all computations must be performed in registers, memory traffic can be eliminated if the caller can compute arguments into registers and pass them in the same registers to the called function, where the called function can then use these arguments for further computation in the same registers. The number of registers implemented in a processor architecture naturally limits the number of arguments that can be passed in this manner.

This ABI defines that the following registers shall be used for parameter passing:

- General registers `r2` to `r5` (volatile)
- General register `r6` (nonvolatile)
- Floating-point registers `f0`, `f2`, `f4` and `f6` (volatile)
- Vector registers `v24` to `v31` (volatile)

If needed, more arguments are passed in the parameter area, which starts 160 bytes above the stack pointer (see figure 1.18).

The following algorithm specifies where argument data is passed for the C language. For this purpose, consider the arguments as ordered from left (first argument) to right, although the order of evaluation of the arguments is unspecified. In this algorithm `fr` contains the number of the next available floating-point register, `gr` contains the number of the next available general register, and `starg` is the address of the next available stack argument word.

‹initialize›: Allocate a sufficiently large parameter area for the arguments that will be passed according to the ‹more› and ‹more_vec› descriptions that follow. Set `fr` = 0, `gr` = 2, `vr` = 24, and `starg` to the address of the parameter area.

‹return_parameter›: If the called function's return value is not passed in a register (according to section 1.2.5), then allocate a return value buffer, store its address in `r2`, and set `gr` = 3.

‹scan›: If there are no more arguments, terminate. Otherwise, select one of the following depending on the type of the next argument:

‹double_or_float›: A double_or_float is one of the following:

- A `float` or `_Decimal32`.
- A `double` or `_Decimal64`.
- A structure equivalent to one of the above. A structure is equivalent to a type T if and only if it has exactly one member, which is either of type T itself or a structure equivalent to type T.

If `fr` > 6, that is, if there are no more floating-point registers available for parameter passing, go to ‹more›. Otherwise, load the argument value into floating-point register `fr`, set `fr` to `fr` + 2, and go to ‹scan›.

‹vector_arg›: A vector_arg has one of the following types:

- Any vector type whose size is 16 bytes or less.
- A structure equivalent to such a vector type, where "equivalent" has the same meaning as for double_or_float.

If the argument is part of the varying arguments (see section 1.2.4), or if `vr` = *nil*, go to ‹more_vec›. Otherwise, load the value left-justified into vector register `vr`, set `vr` to the next entry in the list

$$24, 26, 28, 30, 25, 27, 29, 31, \mathit{nil}$$

and go to ‹scan›.

‹simple_arg›: A simple_arg is one of the following:

- One of the simple integer types no more than 64 bits wide. This includes `signed char`, `short`, `int`, `long`, `long long`, their unsigned counterparts, `_Bool`, and any `enum` type. If such an argument is shorter than 64 bits, replace it by a full 64-bit integer representing the same number, using sign or zero extension, as appropriate.
- Any pointer type.
- A struct or a union of 1, 2, 4, or 8 bytes that is not a double_or_float (see above). If such a struct or union is strictly smaller than 8 bytes, extend it to 8 bytes by adding padding bytes with unspecified contents on the left.
- A struct or union of any other size, a complex type, an `__int128`, a `long double`, a `_Decimal128`, or a vector whose size exceeds 16 bytes. Replace such an argument by a pointer to the object, or to a copy where necessary to enforce call-by-value semantics. Only if the caller can ascertain that the object is "constant" can it pass a pointer to the object itself.

If `gr` > 6, go to ‹more›. Otherwise load the argument value (now 64 bits wide) into general register `gr`, set `gr` to `gr` + 1, and go to ‹scan›.

```
typedef float __attribute__((vector_size(8)) v2f_t;

int i, j, k, l;
long long ll;
double f, g, h;
v2f_t v1, v2;
int m;
x = func(i, j, g, k, l, ll, f, h, m, v1, v2);
```

Listing 1.1: Parameter-passing example

General registers	Floating-point registers	Vector registers	Stack frame offset
r2: i	f0: g	v24: v1	160: m
r3: j	f2: f	v26: v2	
r4: k	f4: h		
r5: l			
r6: ll			

Table 1.4.: Parameter-passing example: register allocation

‹more›: The argument cannot be passed in registers; it will be passed in the parameter area of the caller's stack frame instead. After having applied the replacement rules previously explained as appropriate, the argument now has a size of 8 bytes, except when its type is equivalent to `float` or `_Decimal32`, in which case it has 4 bytes. Copy the argument value right-aligned into the 8-byte parameter slot at the current stack position `starg`, leaving the skipped bytes (if any) at unspecified values. Increment `starg` by 8, then go to ‹scan›.

‹more_vec›: The argument cannot be passed in vector registers, but will be passed in the parameter area. Copy its value to the current stack position `starg`, increment `starg` by the argument size, align `starg` to the next 8-byte boundary, and go to ‹scan›.

As an example, assume the declarations and the function call shown in listing 1.1. The corresponding register allocation and storage would be as shown in table 1.4.

1.2.4. Variable Argument Lists

If a C function declaration has a parameter type list that terminates with an ellipsis "`...`," a call to that function can have varying numbers and types of arguments corresponding to the ellipsis. Except for vector arguments of 16 bytes or less, these varying arguments are passed to the called function as if the ellipsis were replaced with a parameter type list of the actual arguments. Varying vector arguments are always passed in the parameter area.

The called function can store the varying arguments in a variable of type `va_list`, defined in `<stdarg.h>`. Such a variable represents the list of remaining arguments to be

```
typedef struct __va_list_tag {
    long __gpr;
    long __fpr;
    void *__overflow_arg_area;
    void *__reg_save_area;
} va_list[1];
```

Listing 1.2: `va_list` declaration example

processed and can be passed down to further functions. The s390x ABI defines `va_list` to be equivalent to a structure with four doubleword members, or to an array whose single element is such a structure, like the declaration shown in listing 1.2. The declaration as an array reduces copying of the structure when used as an argument. The structure members have the following meaning:

`__gpr` holds the number (0 to 5) of general argument registers that have already been processed.

`__fpr` holds the number (0 to 4) of floating-point argument registers that have already been processed.

`__overflow_arg_area` points to the first "overflow argument" (passed via the parameter area) that has not been processed yet.

`__reg_save_area` points to the start of a 160-byte memory region that contains the saved values of all argument registers, with the general registers (`r2` to `r6`) starting at offset 16 and the floating-point registers (`f0`, `f2`, `f4`, and `f6`) starting at offset 128. These offsets correspond to the layout shown in figure 1.17. The argument registers that have already been processed do not actually need to be saved in their slots.

Note: Since `va_list` may be defined as an array, a variable of this type cannot be copied by a simple C assignment. The standard C header `<stdarg.h>` defines the macro `va_copy` for this purpose instead. Any C code that intends to be portable across platforms should use this macro for copying a `va_list` variable.

1.2.5. Return Values

A function must pass its return value either in general register `r2`, in floating-point register `f0`, in vector register `v24`, or in a return value buffer allocated by the caller, depending on the return value type:

- A value of type `double` or `_Decimal64` is returned in `f0`.
- A value of type `float` or `_Decimal32` is returned in the left half of `f0` and encoded in short BFP format or short DFP format, respectively. The right half of `f0` is unspecified.
- Any integer type with 64 or fewer bits, including `_Bool`, as well as any `enum` type, is returned in `r2`. The return value is zero- or sign-extended to 64 bits, as appropriate.
- A pointer to any type is returned in `r2`.

- A vector of 16 or fewer bytes is returned left-aligned in `v24`. The padding bits' values are unspecified.
- Any other type, such as `long double`, `_Decimal128`, `__int128`, a complex type, a structure, a union, or a vector larger than 16 bytes, is returned in a return value buffer allocated by the caller. This buffer's address is treated like a "hidden argument" and passed by the caller in `r2`.

1.3. Operating System Interface

This section describes various interfaces with the operating system that are specific to the s390x ABI.

1.3.1. Signal Context

A signal handler that was installed with `sigaction` using the `SA_SIGINFO` flag receives three arguments, as follows:

```
void handler(int sig, siginfo_t *info, void *ucontext);
```

The second argument `info` is a pointer to a structure containing additional signal information, including the number `si_code` that indicates why the signal `sig` was sent.

The third argument `ucontext` points to a `ucontext_t` structure on the stack where signal-related context information has been saved by the operating system. It contains the processing context to be restored when resuming the interrupted program, including the architecture-dependent register state. Although most signal handlers will ignore this information, some may access it for debugging purposes such as printing the registers, or when their logic depends on that state.

Listing 1.3 shows the declaration of `ucontext_t` on systems implementing the s390x ABI.

1.3.2. Exception Interface

When the CPU detects an exceptional condition while a process is executing instructions, an interruption may occur, transferring control to the operating system. The operating system then handles the interruption either in a manner transparent to the application, or by delivering a signal.

If such an exception and its corresponding interruption are immediately caused by the execution of an instruction, the exception is called "synchronous". Program interruptions generally fall into this category. They may give rise to `SIGILL`, `SIGSEGV`, `SIGBUS`, `SIGTRAP`, or `SIGFPE`. If one of these signals is generated due to an exception when the signal is blocked, the behavior is undefined.

When a signal handler other than for `SIGSEGV` or `SIGBUS` gets control after a synchronous exception, the `si_addr` field in the signal handler's `siginfo_t` argument points to the

```
typedef struct {
    unsigned long      mask;       /* PSW mask */
    unsigned long      addr;       /* PSW address */
} __psw_t;

typedef union {
    double             d;
    float              f;
} fpreg_t;

typedef struct {
    unsigned int       fpc;        /* floating-point control register */
    fpreg_t            fprs[16];   /* floating-point registers */
} fpregset_t;

typedef struct {
    __psw_t            psw;
    unsigned long      gregs[16];  /* general registers */
    unsigned int       aregs[16];  /* access registers */
    fpregset_t         fpregs;
} mcontext_t;

typedef struct {
    void              *ss_sp;
    int                ss_flags;
    size_t             ss_size;
} stack_t;

typedef ... sigset_t;               /* opaque type */

struct ucontext_t {
    unsigned long      uc_flags;
    struct ucontext_t *uc_link;
    stack_t            uc_stack;
    mcontext_t         uc_mcontext; /* machine-specific context */
    sigset_t           uc_sigmask;  /* blocked signals */
};
```

Listing 1.3: The `ucontext_t` structure. The size of `uc_sigmask` may vary, and additional information may be stored after it.

instruction that caused the exception, while the PSW address in the signal context points to the next instruction.

In the case of `SIGSEGV` or `SIGBUS`, the PSW address points to the faulting instruction instead, whereas `si_addr` points to the address of the memory access causing the fault, possibly rounded down to a page boundary.

The correspondence between the causes of program interruptions and the resulting signals is shown in table 1.5.

1.3.3. Virtual Address Space

Processes execute in a 64-bit virtual address space. Memory management translates virtual addresses to physical addresses, hiding physical addressing and letting a process run anywhere in the system's real memory. Processes typically begin with three logical segments, commonly called "text," "data," and "stack." An object file may contain more segments (for example, for debugger use), and a process can also create additional segments for itself with system services.

Note: The term "virtual address" as used in this document refers to a 64-bit address generated by a program, as contrasted with the physical address to which it is mapped.

1.3.4. Page Size

Memory is organized into pages, which are the system's smallest units of memory allocation. The hardware page size for z/Architecture is 4096 bytes.

1.3.5. Virtual Address Assignments

Processes have a 42, 53, or 64-bit address space available to them, depending on the Linux kernel level.

Figure 1.19 shows the virtual address configuration on the z/Architecture. The segments with different properties are typically grouped in different areas of the address space. The loadable segments may begin at zero (`0`); the exact addresses depend on the executable file format (see chapters 2 and 3). The process's stack resides at the end of the virtual memory and grows downwards. Processes can control the amount of virtual memory allotted for stack space, as described below.

Note: Although application programs may begin at virtual address 0, they conventionally begin above `0x1000` (4 Kbytes), leaving the initial 4 Kbytes with an invalid address mapping. Processes that reference this invalid memory (for example by de-referencing a null pointer) generate a translation exception as described in section 1.3.2.

Although applications may control their memory assignments, the typical arrangement follows figure 1.19.

z/Architecture exception	Signal	si_code
Addressing	SIGILL	ILL_ILLADR
Data, general-operand		ILL_ILLOPN
Execute		ILL_ILLOPN
Operand		ILL_ILLOPN
Operation, no breakpoint†		ILL_ILLOPC
Privileged-operation		ILL_PRVOPC
Special-operation		ILL_ILLOPN
Space-switch		ILL_PRVOPC
Specification		ILL_ILLOPN
Transaction-constraint		ILL_ILLOPN
Operation, breakpoint†	SIGTRAP	TRAP_BRKPT
Data, (simulated) IEEE invalid operation	SIGFPE	FPE_FLTINV
Data, (simulated) IEEE division by zero		FPE_FLTDIV
Data, any (simulated) IEEE overflow		FPE_FLTOVF
Data, any (simulated) IEEE underflow		FPE_FLTUND
Data, any (simulated) IEEE inexact‡		FPE_FLTRES
Data, neither IEEE nor general-operand		SI_USER
Fixed-point/decimal divide		FPE_INTDIV
Fixed-point/decimal overflow		FPE_INTOVF
HFP divide		FPE_FLTDIV
HFP exp. overflow		FPE_FLTOVF
HFP exp. underflow		FPE_FLTUND
HFP square root		FPE_FLTINV
HFP significance		FPE_FLTRES
Vector-processing, invalid operation		FPE_FLTINV
Vector-processing, division by zero		FPE_FLTDIV
Vector-processing, overflow		FPE_FLTOVF
Vector-processing, underflow		FPE_FLTUND
Vector-processing, inexact		FPE_FLTRES
Protection	SIGSEGV	SEGV_ACCERR
Any translation*		SEGV_MAPERR
Any translation*	SIGBUS	BUS_ADDRERR

† A breakpoint is recognized when a `ptrace` target executes the special illegal instruction `0x0001`.
‡ Except if an overflow or underflow condition is indicated as well.
* For a translation exception the operating system may yield SIGSEGV or SIGBUS, or it may handle the fault without a signal.

Table 1.5.: Exceptions and signals. `si_code` refers to the respective field in `siginfo_t`.

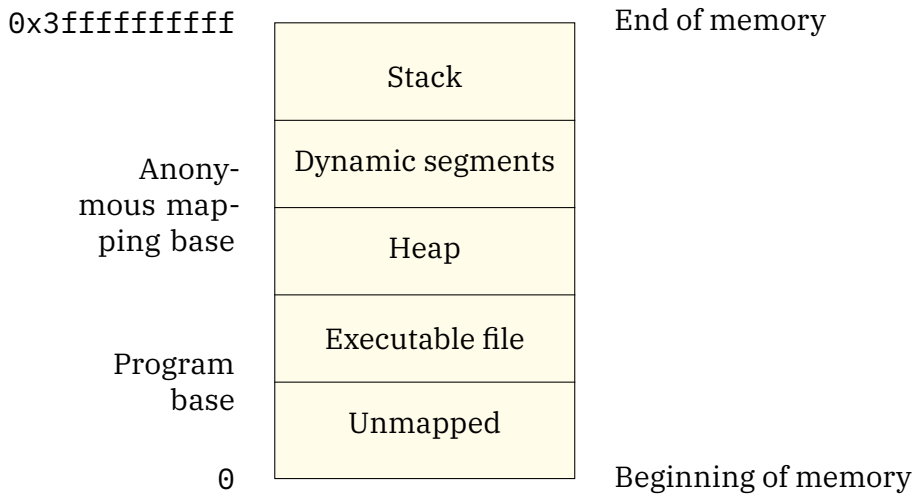

Figure 1.19.: 42-bit virtual address configuration

1.3.6. Managing the Process Stack

Section 1.4 describes the initial stack contents. Stack addresses can change from one system to the next—even from one process execution to the next on a single system. A program, therefore, should not depend on finding its stack at a particular virtual address.

A tunable configuration parameter controls the system maximum stack size. A process can also use `setrlimit` to set its own maximum stack size, up to the system limit. The stack segment is both readable and writable.

1.3.7. Coding Guidelines

Operating system facilities, such as `mmap`, allow a process to establish address mappings in two ways. Firstly, the program can let the system choose an address. Secondly, the program can request the system to use an address the program supplies. The second alternative can cause application portability problems because the requested address might not always be available. Differences in virtual address space can be particularly troublesome between different architectures, but the same problems can arise within a single architecture.

Processes' address spaces typically have three segments that can change size from one execution to the next: the stack (through `setrlimit`); the data segment (through `malloc`); and the dynamic segment area (through `mmap`). Changes in one area may affect the virtual addresses available for another. Consequently an address that is available in one process execution might not be available in the next. Thus a program that used `mmap` to request a mapping at a specific address could appear to work in some environments and fail in others. For this reason programs that want to establish a mapping in their address space should let the system choose the address.

Despite these warnings about requesting specific addresses, the facility can be used properly. For example, a multiprocess application might map several files into the address

space of each process and build relative pointers among the files' data. This could be done by having each process ask for a certain amount of memory at an address chosen by the system. After each process received its own private address from the system it would map the desired files into memory at specific addresses within the original area. This collection of mappings could be at different addresses in each process but their relative positions would be fixed. Without the ability to ask for specific addresses, the application could not build shared data structures because the relative positions for files in each process would be unpredictable.

1.3.8. Processor Execution Modes

Two execution modes exist in z/Architecture: problem (user) state and supervisor state. Processes run in problem state (the less privileged). The operating system kernel runs in supervisor state. A program executes a "Supervisor Call" (SVC) instruction to change execution modes.

Note that the ABI does not define the implementation of individual system calls. Instead programs should use the system libraries. Programs with embedded SVC instructions do not conform to the ABI.

1.4. Process Initialization

This section describes the machine state that `exec` creates for "infant" processes, including argument passing, register usage, and stack frame layout. Programming language systems use this initial program state to establish a standard environment for their application programs. For example, a C program begins executing at a function named `main`, conventionally declared as follows:

```
extern int main (int argc, char *argv[ ], char *envp[ ]);
```

Its parameters are passed from the C programming language system when invoking `main`. They are:

argc a non-negative argument count

argv an array of argument strings, with

```
argv[argc] == NULL
```

envp an array of environment strings, also terminated by a null pointer

Although this section does not describe C program initialization, it gives the information necessary to implement the call to `main` or to the entry point for a program in any other language.

1.4.1. Registers

When a process is first entered (from an `exec` system call), the contents of registers other than those listed below are unspecified. Consequently, a program that requires registers

```
typedef struct {
    long a_type;
    union {
        long a_val;
        void *a_ptr;
        void (*a_fcn)();
    } a_un;
} auxv_t;
```

Listing 1.4: Auxiliary vector structure

to have specific values must set them explicitly during process initialization. It should not rely on the operating system to set all registers to 0. Following are the registers whose contents are specified:

r15 The initial stack pointer, aligned to an 8-byte boundary and pointing to a stack location that contains the argument count (see section 1.4.2 for further information about the initial stack layout).

fpc The floating-point control register contains 0, specifying "round to nearest" mode and the disabling of floating-point exceptions.

1.4.2. Process Stack

Every process has a stack, but the system defines no fixed stack address. Furthermore, a program's stack address can change from one system to another—even from one process invocation to another. Thus the process initialization code must use the stack address in general register `r15`. Data in the stack segment at addresses below the stack pointer contain undefined values.

When a process receives control, its stack holds the arguments, environment, and auxiliary vector (see section 1.4.3) from `exec`. Argument strings, environment strings, and the auxiliary information appear in no specific order within the information block; the system makes no guarantees about their relative arrangement. The system may also leave an unspecified amount of memory between the `NULL` auxiliary vector entry and the beginning of the information block. A sample initial stack is shown in figure 1.20.

1.4.3. Auxiliary Vector

Whereas the argument and environment vectors transmit information from one application program to another, the auxiliary vector conveys information from the operating system to the program. This vector is an array of structures, which are defined in listing 1.4.

The structures are interpreted according to the `a_type` member, as shown in table 1.6.

`a_type` auxiliary vector types are described in the following:

AT_NULL The auxiliary vector has no fixed length, so an entry of this type is used to denote the end of the vector. The corresponding value of `a_un` is undefined.

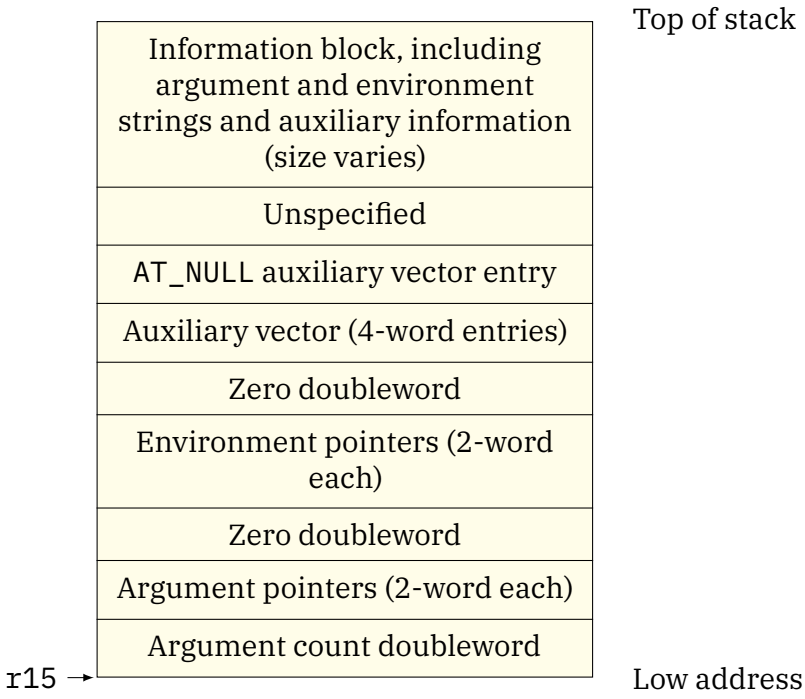

Figure 1.20.: Initial process stack

Name	Value	a_un	Name	Value	a_un
AT_NULL	0	ignored	AT_UID	11	a_val
AT_IGNORE	1	ignored	AT_EUID	12	a_val
AT_EXECFD	2	a_val	AT_GID	13	a_val
AT_PHDR	3	a_ptr	AT_EGID	14	a_val
AT_PHENT	4	a_val	AT_PLATFORM	15	a_ptr
AT_PHNUM	5	a_val	AT_HWCAP	16	a_val
AT_PAGESZ	6	a_val	AT_CLKTCK	17	a_val
AT_BASE	7	a_ptr	AT_SECURE	23	a_val
AT_FLAGS	8	a_val	AT_RANDOM	25	a_ptr
AT_ENTRY	9	a_ptr	AT_EXECFN	31	a_ptr
AT_NOTELF	10	a_val	AT_SYSINFO_EHDR	33	a_ptr

Table 1.6.: Auxiliary vector types, a_type

Name	Value	Description
HWCAP_S390_ZARCH	0x2	Running in z/Architecture mode
HWCAP_S390_STFLE	0x4	Store-facility-list-extended facility installed
HWCAP_S390_MSA	0x8	Message-security assist available
HWCAP_S390_LDISP	0x10	Long-displacement facility installed
HWCAP_S390_EIMM	0x20	Extended-immediate facility installed
HWCAP_S390_DFP	0x40	Decimal floating-point facility and perform floating-point facility (PFPO) installed
HWCAP_S390_HPAGE	0x80	Huge page support available
HWCAP_S390_ETF3EH	0x100	Extended-translation facility 3 and ETF3-enhancement facility installed
HWCAP_S390_TE	0x400	Transactional-execution facility installed
HWCAP_S390_VXRS	0x0800	Vector facility installed
HWCAP_S390_VXRS_BCD	0x1000	Vector packed-decimal facility installed
HWCAP_S390_VXRS_EXT	0x2000	Vector-enhancements facility 1 installed
HWCAP_S390_GS	0x4000	Guarded-storage facility installed
HWCAP_S390_VXRS_EXT2	0x8000	Vector-enhancements facility 2 installed
HWCAP_S390_VXRS_PDE	0x10000	Vector-packed-decimal enhancement facility installed
HWCAP_S390_DFLT	0x40000	Deflate-conversion facility installed

Table 1.7.: Hardware capabilities

AT_IGNORE This type indicates the entry has no meaning. The corresponding value of a_un is undefined.

AT_EXECFD exec may pass control to an interpreter program. When this happens, the system places either an entry of type AT_EXECFD or one of type AT_PHDR in the auxiliary vector. The a_val field in the AT_EXECFD entry contains a file descriptor for the application program's object file.

AT_PHDR Under some conditions, the system creates the memory image of the application program before passing control to an interpreter program. When this happens, the a_ptr field of the AT_PHDR entry tells the interpreter where to find the program header table in the memory image. If the AT_PHDR entry is present, entries of types AT_PHENT, AT_PHNUM, and AT_ENTRY must also be present. See chapter 3 for more information about the program header table.

AT_PHENT The a_val field of this entry holds the size, in bytes, of one entry in the program header table at which the AT_PHDR entry points.

AT_PHNUM The a_val field of this entry holds the number of entries in the program header table at which the AT_PHDR entry points.

AT_PAGESZ If present, this entry's a_val field gives the system page size in bytes. The same information is also available through sysconf.

AT_BASE The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory.

AT_FLAGS If present, the a_val field of this entry holds 1-bit flags. Undefined bits are set to zero.

AT_ENTRY The a_ptr field of this entry holds the entry point of the application program to which the interpreter program should transfer control.

AT_NOTELF The a_val field of this entry is non-zero if the program is in another format than ELF, for example in the old COFF format.

AT_UID The a_ptr field of this entry holds the real user id of the process.

AT_EUID The a_ptr field of this entry holds the effective user id of the process.

AT_GID The a_ptr field of this entry holds the real group id of the process.

AT_EGID The a_ptr field of this entry holds the effective group id of the process.

AT_PLATFORM The a_ptr field of this entry holds the address of a string that identifies the platform the program runs on.

AT_HWCAP The a_val field of this entry holds a bit map of hardware capabilities hints. Table 1.7 lists some of the assigned bits and their meaning.

AT_CLKTCK The a_val field of this entry holds the number of clock ticks per second. The function times(), which measures execution time, reports all times in clock ticks. The number of clock ticks per second is also available through sysconf.

AT_SECURE The a_val field of this entry holds a Boolean that indicates whether the program shall be locked into a secure environment, such as when access rights have been upgraded by executing a setuid/setgid executable.

AT_RANDOM The a_ptr field of this entry holds the address of 16 random bytes.

AT_EXECFN The a_ptr field of this entry holds the address of a string that contains the executable's file name.

AT_SYSINFO_EHDR The a_ptr field of this entry holds the address at which the system-supplied dynamic shared object (DSO), specifically its ELF header, is mapped in the program's virtual address space.

Other auxiliary vector types are reserved. No flags are currently defined for AT_FLAGS on s390x.

1.5. Coding Examples

This section describes example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections discussed how a program may use the machine or the operating system, and they specified what a program may and may not assume about the execution environment. Unlike previous material, the information in this section illustrates how operations *may* be done, not how they *must* be done.

As before, examples use the ISO 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.

The following sections describe the differences between these models. When different, code sequences for the models appear together for easier comparison.

Note: The examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences or to reproduce compiler output.

1.5.1. Code Model Overview

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 if 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 Current Instruction Address (CIA), or use registers that hold the transfer address. A CIA-relative branch computes its destination address in terms of the CIA, not relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in instructions (in the text segment), the compiler generates code to calculate an absolute address (in a register or in the stack or data segment) during execution.

Because z/Architecture provides CIA-relative branch instructions and also branch instructions using registers that hold the transfer address, compilers can satisfy the first condition easily.

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

Two position-independent models give programs a choice between more efficient code with some size restrictions and less efficient code without those restrictions. Because of the processor architecture, a GOT with no more than 512 entries (4096 bytes) is more efficient than a larger one. Programs that need more entries must use the larger, more general code. In the following sections, the term “small model position-independent code” is used to refer to code that assumes the smaller GOT, and “large model position-independent code” is used to refer to the general code.

1.5.2. Function Prologue and Epilogue

This section describes the prologue and epilogue code of functions. A function’s prologue establishes a stack frame, if necessary, and may save any nonvolatile registers it uses. A function’s epilogue generally restores registers that were saved in the prologue code, restores the previous stack frame, and returns to the caller.

1.5.2.1. Prologue

The prologue of a function has to save the state of the calling function and set up the base register for the code of the function body. The following is in general done by the function prologue:

- Save all registers used within the function which the calling function assumes to be nonvolatile.
- Set up the base register for the literal pool, if needed.
- Allocate stack space by decrementing the stack pointer.
- Set up the dynamic chain by storing the old stack pointer value at stack location zero if the “back chain” is implemented.

```
          .section .rodata
          .align  2
.LC0:     .string "hello, world!"

          .text
          .align  8
          .globl  main
          .type   main, @function
main:
                                        # Prologue
          stmg    %r14,%r15,112(%r15)   # Save caller's registers
          lgr     %r1,%r15              # Load stack pointer into r1
          aghi    %r15,-160             # Allocate new stack frame
          stg     %r1,0(%r15)           # Store back chain
                                        # Prologue end
          larl    %r2,.LC0
          brasl   %r14,puts
          lghi    %r2,0
                                        # Epilogue
          lmg     %r14,%r15,272(%r15)   # Restore registers
          br      %r14                  # Branch back to caller
                                        # Epilogue end
```

Listing 1.5: Prologue and epilogue example. This example stores the optional backchain.

- Set up the GOT pointer if the compiler is generating position-independent code.
  (Usually the GOT pointer is loaded into a nonvolatile register. This may be omitted if the function makes no external data references. If external data references are only made within conditional code, loading the GOT pointer may be deferred until it is known to be needed.)
- Set up the frame pointer if the function allocates stack space dynamically (with `alloca`).

The compiler tries to do as little as possible of the above; the ideal case is to do nothing at all (for a leaf function without symbolic references).

1.5.2.2. Epilogue

The epilogue of a function restores the registers saved in the prologue (which include the stack pointer) and branches to the return address.

The small program in listing 1.5 shows a simple example of a function prologue and epilogue.

1.5.3. Profiling

This section shows a way of providing profiling (entry counting) for s390x applications. An ABI-conforming system is not required to provide profiling; however, if it does, this is

```
        stg     %r14,8(%r15)          # Pass r14 in first regsave slot
        brasl   %r14,_mcount          # Branch to _mcount
        lg      %r14,8(%r15)          # Restore r14
        stmg    %r7,%r15,56(%r15)     # Save caller's registers
        aghi    %r15,-160             # Allocate new frame
        ...
```

Listing 1.6: Code for profiling

one possible (not required) implementation.

If a function is to be profiled, it has to call the `_mcount` routine before the function prologue. This routine has a special linkage. Its return address is passed in `r14` as usual. However, instead of register arguments it receives the caller's return address in the first slot of the register save area, which is located 8 bytes above the current stack pointer. And it preserves more registers than a normal function, treating all the usual argument registers as nonvolatile as well. Since `_mcount` gets invoked before the caller's prologue, no additional frame needs to be allocated for it. It may overwrite the caller's register save area, except for the first slot, which it will preserve.

Listing 1.6 shows an example of a function prologue preceded by a call to `_mcount`.

1.5.4. Data Objects

This section describes only objects with static storage duration. It excludes stack-resident objects because programs always compute their virtual addresses relative to the stack or frame pointers.

Because z/Architecture instructions cannot hold 64-bit addresses directly, a program has to build an address in a register and access memory through that register. In order to do so, a function may contain a literal pool that holds the addresses of data objects used by the function. Then `r13` is typically set up in the function prologue to point to the start of this literal pool.

Position-independent code cannot contain absolute addresses. In order to access a local symbol, the literal pool contains the (signed) offset of the symbol relative to the start of the pool. Combining the offset loaded from the literal pool with the address in `r13` gives the absolute address of the local symbol. In the case of a global symbol the address of the symbol has to be loaded from the Global Offset Table. The offset in the GOT can either be contained in the instruction itself or in the literal pool.

Tables 1.8 to 1.10 show sample assembly language equivalents to C language code for absolute and position-independent compilations. It is assumed that all shared objects are compiled as position-independent and only executable modules may have absolute addresses. The function prologue is not shown, and it is assumed that it has loaded the address of the literal pool in `r13`.

C	z/Architecture machine instructions (Assembler)

extern int src;
extern int dst;
extern int *ptr;
dst = src;
ptr = &dst;
	larl %r1,src
	larl %r2,dst
	larl %r3,ptr
	mvc 0(4,%r2),0(%r1) *# dst = src*
	stg %r2,0(%r3) *# ptr = &dst*

Table 1.8.: Absolute addressing

C	z/Architecture machine instructions (Assembler)

extern int src;
extern int dst;
extern int *ptr;
dst = src;
ptr = &dst;
*ptr = src;
	larl %r12,_GLOBAL_OFFSET_TABLE_
	lg %r1,dst@GOT12(%r12)
	lg %r2,src@GOT12(%r12)
	lgf %r3,0(%r2)
	st %r3,0(%r1)
	larl %r12,_GLOBAL_OFFSET_TABLE_
	lg %r1,ptr@GOT12(%r12)
	lg %r2,dst@GOT12(%r12)
	stg %r2,0(%r1)
	larl %r12,_GLOBAL_OFFSET_TABLE_
	lg %r2,ptr@GOT12(%r12)
	lg %r1,0(%r2)
	lg %r2,src@GOT12(%r12)
	lgf %r3,0(%r2)
	st %r3,0(%r1)

Table 1.9.: Small model position-independent addressing

C	z/Architecture Assembler

extern int src;
extern int dst;
extern int *ptr;
dst = src;
ptr = &dst;
*ptr = src;
	larl %r2,dst@GOT
	lg %r2,0(%r2)
	larl %r3,src@GOT
	lg %r3,0(%r3)
	mvc 0(4,%r2),0(%r3)
	larl %r2,ptr@GOT
	lg %r2,0(%r2)
	larl %r3,dst@GOT
	lg %r3,0(%r3)
	stg %r3,0(%r2)
	larl %r2,ptr@GOT
	lg %r2,0(%r2)
	larl %r3,src@GOT
	lg %r3,0(%r3)
	mvc 0(4,%r3),0(%r2)

Table 1.10.: Large model position-independent addressing

C	z/Architecture machine instructions (Assembler)

`extern void func();`
`extern void (*ptr)();`
`ptr = func;`
`func();`
`(*ptr) ();`
	`larl  %r1,ptr`
	`larl  %r2,func`
	`stg   %r2,0(%r1)`
	`brasl %r14,func`
	`larl  %r1,ptr`
	`lg    %r1,0(%r1)`
	`basr  %r14,%r1`

Table 1.11.: Absolute function call

C	z/Architecture machine instructions (Assembler)

`extern void func();`
`extern void (*ptr)();`
`ptr = func;`
`func();`
`(*ptr) ();`
	`larl  %r12,_GLOBAL_OFFSET_TABLE_`
	`lg    %r1,ptr@GOT12(%r12)`
	`lg    %r2,func@GOT12(%r12)`
	`stg   %r2,0(%r1)`
	`brasl %r14,func@PLT`
	`larl  %r12,_GLOBAL_OFFSET_TABLE_`
	`lg    %r1,ptr@GOT12(%r12)`
	`lg    %r1,0(%r1)`
	`basr  %r14,%r1`

Table 1.12.: Small model position-independent function call

1.5.5. Function Calls

Programs can use the z/Architecture BRASL instruction to make direct function calls. A BRASL instruction has a self-relative branch displacement that can reach 4 GBytes in either direction. To call functions beyond this limit (inter-module calls), load the address in a register and use the BASR instruction for the call. Register r14 is used as the first operand of BASR to hold the return address as shown in table 1.11.

The called function may be in the same module (executable or shared object) as the caller, or it may be in a different module. In the former case, if the called function is not in a shared object, the linkage editor resolves the symbol. In all other cases the linkage editor cannot directly resolve the symbol. Instead the linkage editor generates “glue” code and resolves the symbol to point to the glue code. The dynamic linker will provide the real address of the function in the Global Offset Table. The glue code loads this address and branches to the function itself. See section 3.2.4 for more details.

1.5.6. Branching

Programs use branch instructions to control their execution flow. z/Architecture has a variety of branch instructions. The most commonly used of these performs a self-relative jump with a 128-Kbyte range (up to 64 Kbytes in either direction). For large functions,

C	z/Architecture machine instructions (Assembler)

extern void func();
extern void (*ptr)();
ptr = func;
func();
(*ptr) ();
	larl %r2,ptr@GOT
	lg %r2,0(%r2)
	larl %r3,func@GOT
	lg %r3,0(%r3)
	stg %r3,0(%r2)
	brasl %r14,func@PLT
	larl %r2,ptr@GOT
	lg %r2,0(%r2)
	lg %r2,0(%r2)
	basr %r14,%r2

Table 1.13.: Large model position-independent function call

C	z/Architecture machine instructions (Assembler)
label: ... **goto** label; ... ... farlabel: ... ... **goto** farlabel;	.L01: ... j .L01 ... ... .L02: ... ... jg .L02

Table 1.14.: Branch instruction

another self-relative jump is available with a range of 8 Gbytes (up to 4 Gbytes in either direction).

C language switch statements provide multi-way selection. When the case labels of a switch statement satisfy grouping constraints, the compiler implements the selection with an address table. The examples shown in tables 1.15 and 1.16 use several simplifying conventions to hide irrelevant details:

1. The selection expression resides in `r2`.
2. The case label constants begin at zero.
3. The case labels, the default, and the address table use assembly names `.Lcasei`, `.Ldef`, and `.Ltab` respectively.

C	z/Architecture machine instructions (Assembler)	

switch(j)
  {
  case 0:
  /* ... */
  case 1:
  /* ... */
  case 3:
  /* ... */
  default:
  }
	.Ltab:	lghi %r1,%r3
		clgr %r2,%r1
		brc 2,.Ldef
		sllg %r2,%r2,3
		larl %r1,.Ltab
		lg %r3,0(%r1,%r2)
		br %r3
		.quad .Lcase0
		.quad .Lcase1
		.quad .Ldef
		.quad .Lcase3

Table 1.15.: Absolute switch code

C	z/Architecture machine instructions (Assembler)	

switch(j)
  {
  case 0:
  /* ... */
  case 1:
  /* ... */
  case 3:
  /* ... */
  default:
  }
	.LT0:
	
	.Ltab:
		# Literal pool
		
		# Code
		lghi %r1,3
		clgr %r2,%r1
		brc 2,.Ldef
		sllg %r2,%r2,3
		larl %r1,.Ltab
		lg %r3,0(%r1,%r2)
		agr %r3,%r13
		br %r3
		.quad .Lcase0-.LT0
		.quad .Lcase1-.LT0
		.quad .Ldef-.LT0
		.quad .Lcase3-.LT0

Table 1.16.: Position-independent switch code, all models

1.5.7. Dynamic Stack Space Allocation

The GNU C compiler, and most recent compilers, support dynamic stack space allocation via `alloca`.

Figure 1.21 shows the stack frame before and after dynamic stack allocation. The local variables area is used for storage of function data, such as local variables, whose sizes are known to the compiler. This area is allocated at function entry and does not change in size or position during the function's activation.

The parameter area holds "overflow" arguments passed in calls to other functions. (See the ‹more› label in section 1.2.3.) Its size is also known to the compiler and can be allocated along with the fixed frame area at function entry. However, the standard calling sequence requires that the parameter area begins at a fixed offset (160) from the stack pointer, so this area must move when dynamic stack allocation occurs.

Data in the parameter area are naturally addressed at constant offsets from the stack pointer. However, in the presence of dynamic stack allocation, the offsets from the stack pointer to the data in the local-variable area are not constant. To provide addressability, a frame pointer is established to locate the local variables area consistently throughout the function's activation.

Dynamic stack allocation is accomplished by "opening" the stack just above the parameter area. The following steps show the process in detail:

1. After a new stack frame is acquired, and before the first dynamic space allocation, a new register, the frame pointer or FP, is set to the value of the stack pointer. The frame pointer is used for references to the function's local, non-static variables. The frame pointer does not change during the execution of a function, even though the stack pointer may change as a result of dynamic allocation.
2. The amount of dynamic space to be allocated is rounded up to a multiple of 8 bytes, so that 8-byte stack alignment is maintained.
3. The stack pointer is decreased by the rounded byte count, and the address of the previous stack frame (the back chain) may be stored at the word addressed by the new stack pointer. The back chain is not necessary to restore from this allocation at the end of the function since the frame pointer can be used to restore the stack pointer.

Figure 1.21 is a snapshot of the stack layout after the prologue code has dynamically extended the stack frame.

The above process can be repeated as many times as desired within a single function activation. When it is time to return, the stack pointer is set to the value of the back chain, thereby removing all dynamically allocated stack space along with the rest of the stack frame. Naturally, a program must not reference the dynamically allocated stack area after it has been freed.

Even in the presence of signals, the above dynamic allocation scheme is "safe." If a signal interrupts allocation, one of three things can happen:

- The signal handler can return. The process then resumes the dynamic allocation from the point of interruption.

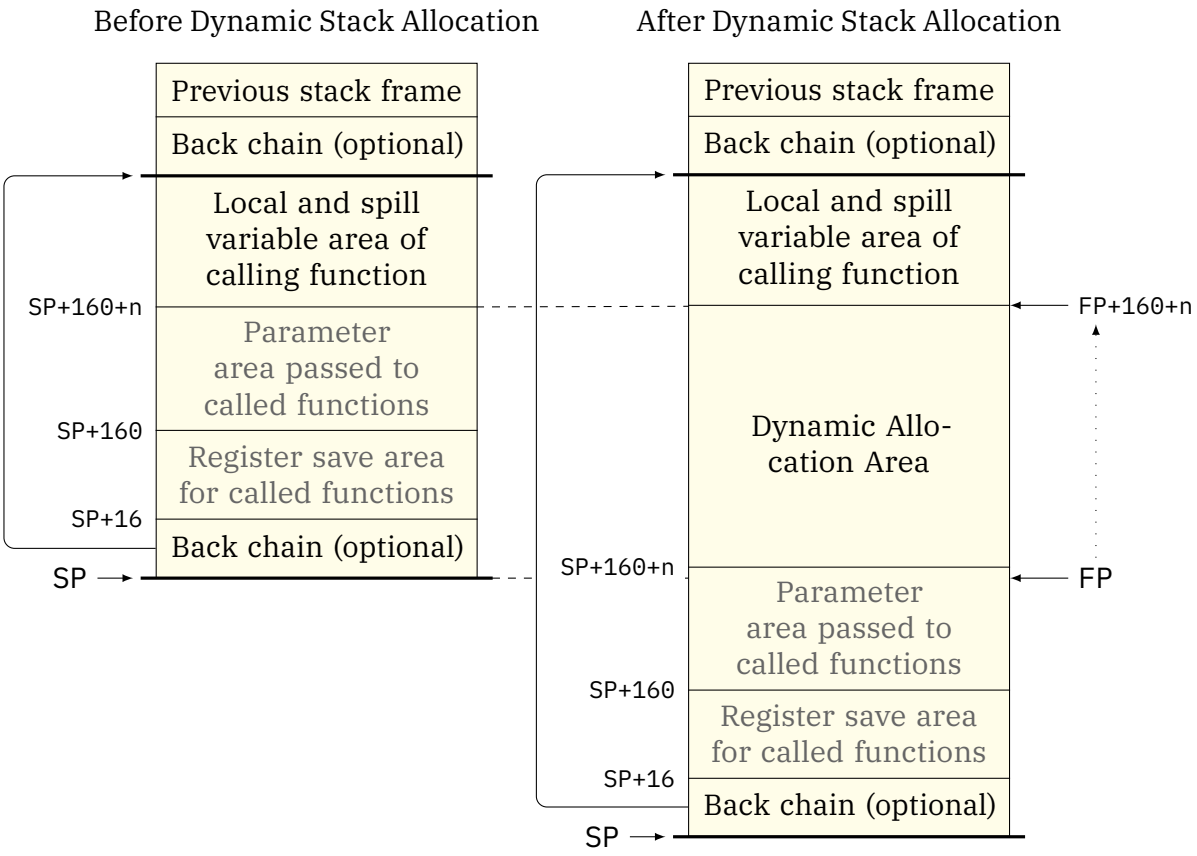

Figure 1.21.: Dynamic stack space allocation

- The signal handler can execute a non-local goto or a jump. This resets the process to a new context in a previous stack frame, automatically discarding the dynamic allocation.
- The process can terminate.

Regardless of when the signal arrives during dynamic allocation, the result is a consistent (though possibly dead) process.

1.6. DWARF Definition

This section defines the "Debug With Attributed Record Format" (DWARF) debugging format for z/Architecture processors. The s390x ABI does not define a debug format. However, all systems that do implement DWARF 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.

The DWARF definition requires some machine-specific definitions. The register number mapping is specified for the z/Architecture processors in table 1.17.

For the placement of a piece within a composite location description, as defined by the byte piece operation `DW_OP_piece` or the bit piece operation `DW_OP_bit_piece`, the following applies:

- Pieces of a floating-point or vector register are taken from the left. This means that a bit piece with offset t and size n consists of the register's bits numbered from t to $t + n - 1$, according to big-endian bit numbering. And a byte piece of a floating-point or vector register of size n consists of the register's n leftmost bytes.
- For any other register, pieces are taken from the right. This means that a bit piece with offset t and size n consists of the bits numbered from $w - t - n$ to $w - t - 1$, where w is the register's bit width. And a byte piece of size n consists of the register's n rightmost bytes.

Whenever interpreting a register as a given type, such as when using the register value operation `DW_OP_regval_type` or the register location description `DW_OP_regx`, the resulting value consists of the same bits as the bit piece starting at offset zero and having the size of the given type.

DWARF number	z/Architecture register	DWARF number	z/Architecture register
0–15	`r0`–`r15`	65	PSW address
16	`f0` / `v0`	66	*reserved* (z/OS)
17	`f2` / `v2`	67	*reserved* (z/OS)
18	`f4` / `v4`	68	`v16`
19	`f6` / `v6`	69	`v18`
20	`f1` / `v1`	70	`v20`
21	`f3` / `v3`	71	`v22`
22	`f5` / `v5`	72	`v17`
23	`f7` / `v7`	73	`v19`
24	`f8` / `v8`	74	`v21`
25	`f10` / `v10`	75	`v23`
26	`f12` / `v12`	76	`v24`
27	`f14` / `v14`	77	`v26`
28	`f9` / `v9`	78	`v28`
29	`f11` / `v11`	79	`v30`
30	`f13` / `v13`	80	`v25`
31	`f15` / `v15`	81	`v27`
32–47	`cr0`–`cr15`[†]	82	`v29`
48–63	`a0`–`a15`	83	`v31`
64	PSW mask		

[†] Control registers cannot be referenced by user-space applications. They are reserved for use by operating system code.

Table 1.17.: DWARF register number mapping

2. Object files

This section describes the Executable and Linking Format (ELF).

2.1. ELF Header

2.1.1. Machine Information

For file identification in `e_ident` the z/Architecture processor family requires the values shown in table 2.1.

The ELF header's `e_flags` field holds bit flags associated with the file. Since the z/Architecture processor family defines no flags, this member contains zero.

Processor identification resides in the ELF header's `e_machine` field and must have the value 22, defined as the name `EM_S390`.

2.2. Sections

2.2.1. Special Sections

Various sections hold program and control information. The following sections, whose types and attributes are listed in table 2.2, are used by the system:

`.got` This section holds the Global Offset Table, or GOT. See sections 1.5 and 3.2.2 for more information.

`.plt` This section holds the Procedure Linkage Table, or PLT. See section 3.2.4 for more information.

Position	Value	Comments
`e_ident[EI_CLASS]`	`ELFCLASS64`	For all 64 bit implementations
`e_ident[EI_DATA]`	`ELFDATA2MSB`	For all Big-Endian implementations

Table 2.1.: Machine-specific ELF identification fields

Name	Type	Attributes
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.plt	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

Table 2.2.: Special sections

2.3. Symbol Table

2.3.1. Symbol Values

A symbol table entry's st_value field is the symbol value. If that value represents a section offset or a virtual address, it must be halfword aligned. This enables use of CIA-relative addressing instructions such as LARL.

If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for the file will contain an entry for that symbol. The st_shndx field 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 field for that symbol table entry is nonzero, the value is the virtual address of the first instruction of that PLT entry. Otherwise the st_value field contains zero. This PLT entry address is used by the dynamic linker in resolving references to the address of the function. See section 3.2.3 for details.

2.4. Relocation

2.4.1. Relocation Types

Relocation entries describe how to alter the instruction and data relocation fields listed below. Figure 2.1 illustrates the affected bits of each field type.

quad64 This specifies a 64-bit field occupying 8 bytes, the alignment of which is 4 bytes unless otherwise specified.

word32 This specifies a 32-bit field occupying 4 bytes, the alignment of which is 4 bytes unless otherwise specified.

pc32 This specifies a 32-bit field occupying 4 bytes with 2-byte alignment. The signed value in this field is shifted to the left by 1 before it is used as a program counter relative displacement (for example, the immediate field of a "Load Address Relative Long" instruction).

pc24 This specifies a 24-bit field contained within 3 consecutive bytes with 1-byte alignment. The signed value in this field is shifted to the left by 1 before it is used as a program counter relative displacement (for example, the third immediate field of a "Branch Prediction Relative Preload" instruction).

mid20 This specifies a 20-bit field contained within 4 consecutive bytes with 2-byte alignment. The 20-bit signed value is the "long displacement" of a memory reference.

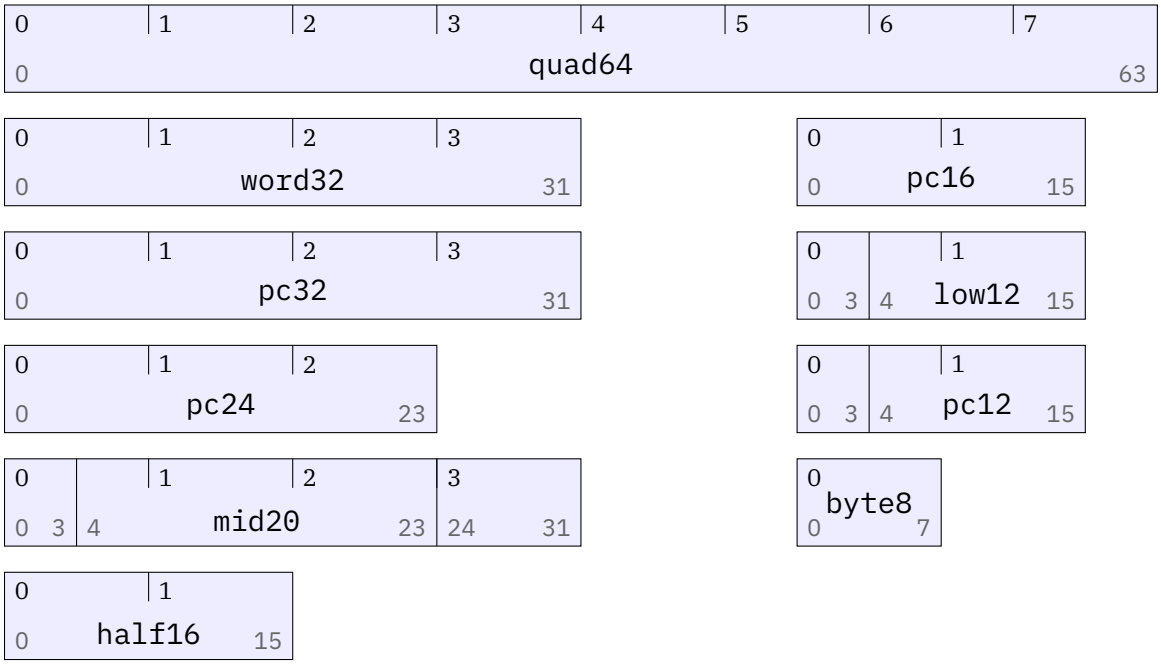

Figure 2.1.: Relocation fields. Bit numbers appear in the lower box corners; byte numbers appear in the upper left box corners.

half16 This specifies a 16-bit field occupying 2 bytes with 2-byte alignment (for example, the immediate field of an "Add Halfword Immediate" instruction).

pc16 This specifies a 16-bit field occupying 2 bytes with 2-byte alignment. The signed value in this field is shifted to the left by 1 before it is used as a program counter relative displacement (for example, the immediate field of an "Branch Relative" instruction).

low12 This specifies a 12-bit field contained within a halfword with 2-byte alignment. The 12 bit unsigned value is the displacement of a memory reference.

pc12 This specifies a 12-bit field contained within a halfword with 1-byte alignment. The signed value in this field is shifted to the left by 1 before it is used as a program counter relative displacement (for example, the second immediate field of a "Branch Prediction Relative Preload" instruction).

byte8 This specifies an 8-bit field with 1-byte alignment.

Calculations in table 2.3 assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the linkage editor merges one or more relocatable files to form the output. It first determines how to combine and locate the input files, next it updates the symbol values, and then it performs relocations.

Relocations applied to executable or shared object files are similar and accomplish the same result. The following notations are used in table 2.3:

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 file is built with a 0 base virtual address, but the execution address will be different.

G Represents the section offset or address of the Global Offset Table. See sections 1.5 and 3.2.2 for more information.

L Represents the section offset or address of the Procedure Linkage Table entry for a symbol. A PLT entry redirects a function call to the proper destination. The linkage editor builds the initial PLT. See section 3.2.4 for more information.

O Represents the offset into the GOT at which the address of the relocation entry's symbol will reside during execution. See sections 1.5 and 3.2.2 for more information.

P Represents the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).

R Represents the offset of the symbol within the section in which the symbol is defined (its section-relative address).

S Represents the value of the symbol whose index resides in the relocation entry.

T Similar to *O*, except that the address that is stored may be the address of the PLT entry for the symbol.

Relocation entries apply to bytes, halfwords, words, or doublewords. In either case, the `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. The z/Architecture family uses only the `Elf64_Rela` relocation entries with explicit addends. For the relocation entries, the `r_addend` field serves as the relocation addend. In all cases, the offset, addend, and the computed result use the byte order specified in the ELF header.

The following general rules apply to the interpretation of the relocation types in table 2.3:

- "+" and "−" denote 64-bit modulus addition and subtraction, respectively. ">>" denotes arithmetic right-shifting (shifting with sign copying) of the value of the left operand by the number of bits given by the right operand.
- Reference in a calculation to the value *G*, *O*, or *T* implicitly creates a GOT entry for the indicated symbol, and a reference to *L* implicitly creates a PLT entry.
- A computed value must be suited for the relocation field it is used for. In particular:

  `half16:` The upper 48 bits must be all ones or all zeroes.

  `pc16:` The upper 47 bits must be all ones or all zeroes and the lowest bit must be zero.

  `pc32:` The upper 31 bits must be all ones or all zeroes and the lowest bit must be zero.

  `low12:` The upper 52 bits must all be zero.

  `byte8:` The upper 56 bits must all be zero.

Table 2.3.: Relocation types

Name	Value	Field	Calculation
R_390_NONE	0	*none*	*none*
R_390_8	1	byte8	$S + A$
R_390_12	2	low12	$S + A$
R_390_16	3	half16	$S + A$
R_390_32	4	word32	$S + A$
R_390_PC32	5	word32	$S + A - P$
R_390_GOT12	6	low12	$O + A$
R_390_GOT32	7	word32	$O + A$
R_390_PLT32	8	word32	$L + A$
R_390_COPY †	9	*none*	
R_390_GLOB_DAT †	10	quad64	$S + A$
R_390_JMP_SLOT †	11	*none*	
R_390_RELATIVE †	12	quad64	$B + A$
R_390_GOTOFF32	13	word32	$S + A - G$
R_390_GOTPC	14	quad64	$G + A - P$
R_390_GOT16	15	half16	$O + A$
R_390_PC16	16	half16	$S + A - P$
R_390_PC16DBL	17	pc16	$(S + A - P) >> 1$
R_390_PLT16DBL	18	pc16	$(L + A - P) >> 1$
R_390_PC32DBL	19	pc32	$(S + A - P) >> 1$
R_390_PLT32DBL	20	pc32	$(L + A - P) >> 1$
R_390_GOTPCDBL	21	pc32	$(G + A - P) >> 1$
R_390_64	22	quad64	$S + A$
R_390_PC64	23	quad64	$S + A - P$
R_390_GOT64	24	quad64	$O + A$
R_390_PLT64	25	quad64	$L + A - P$
R_390_GOTENT	26	pc32	$(G + O + A - P) >> 1$
R_390_GOTOFF16	27	half16	$S + A - G$
R_390_GOTOFF64	28	quad64	$S + A - G$
R_390_GOTPLT12	29	low12	$T + A$
R_390_GOTPLT16	30	half16	$T + A$
R_390_GOTPLT32	31	word32	$T + A$
R_390_GOTPLT64	32	quad64	$T + A$
R_390_GOTPLTENT	33	pc32	$(G + T + A - P) >> 1$
R_390_PLTOFF16	34	half16	$L - G + A$
R_390_PLTOFF32	35	word32	$L - G + A$
R_390_PLTOFF64	36	quad64	$L - G + A$
R_390_TLS_LOAD †	37	*none*	
R_390_TLS_GDCALL †	38	*none*	
R_390_TLS_LDCALL †	39	*none*	
R_390_TLS_GD64 †	41	quad64	

Table 2.3.: Relocation types – *continued*

Name	Value	Field	Calculation
R_390_TLS_GOTIE12 †	42	low12	
R_390_TLS_GOTIE64 †	44	quad64	
R_390_TLS_LDM64 †	46	quad64	
R_390_TLS_IE64 †	48	quad64	
R_390_TLS_IEENT †	49	pc32	
R_390_TLS_LE64 †	51	quad64	
R_390_TLS_LDO64 †	53	quad64	
R_390_TLS_DTPMOD †	54	quad64	
R_390_TLS_DTPOFF †	55	quad64	
R_390_TLS_TPOFF †	56	quad64	
R_390_20	57	mid20	$S + A$
R_390_GOT20	58	mid20	$O + A$
R_390_GOTPLT20	59	mid20	$T + A$
R_390_TLS_GOTIE20 †	60	mid20	
R_390_IRELATIVE †	61	quad64	$*(B + A)()$
R_390_PC12DBL	62	pc12	$(S + A - P) >> 1$
R_390_PLT12DBL	63	pc12	$(L + A - P) >> 1$
R_390_PC24DBL	64	pc24	$(S + A - P) >> 1$
R_390_PLT24DBL	65	pc24	$(L + A - P) >> 1$

The relocation types marked with "†" in table 2.3 are handled specially:

R_390_COPY The linkage 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.

R_390_GLOB_DAT This relocation type resembles R_390_64, except that it sets a Global Offset Table entry to the address of the specified symbol. This special relocation type allows one to determine the correspondence between symbols and GOT entries.

R_390_JMP_SLOT The linkage editor creates this relocation type for dynamic linking. Its offset member gives the location of a Global Offset Table entry. The dynamic linker modifies the GOT entry to transfer control to the designated symbol's address (see section 3.2.4).

R_390_RELATIVE The linkage editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a virtual address. The dynamic linker computes the virtual address by adding the shared object's base address to the addend. Relocation entries for this type must specify 0 for the symbol table index.

R_390_IRELATIVE The linkage editor creates this relocation type for dynamic linking.

The dynamic linker computes an address as for the R_390_RELATIVE relocation and then invokes the function residing at that address, passing the value of AT_HWCAP from the auxiliary vector as its single argument (see section 1.4.3). The return value resulting from that invocation is written into the location described by the offset. Such a function is also known as an “IFUNC resolver” and has the following signature:

```
void *f (unsigned long hwcap);
```

R_390_TLS_* These relocation types are used for thread-local storage handling. They are described in [1].

3. Program Loading and Dynamic Linking

This section describes how the Executable and Linking Format (ELF) is used in the construction and execution of programs.

3.1. Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When—and if—the system physically reads the file depends on the program's execution behavior, on the system load, and so on. A process does not require a physical page until it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore, if physical reads can be delayed they can frequently be dispensed with, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images of which the offsets and virtual addresses are congruent modulo the page size.

Virtual addresses and file offsets for the z/Architecture processor family segments are congruent modulo the system page size. The value of the `p_align` field of each program header in a shared object file must be a multiple of the system page size. Figure 3.1 is an example of an executable file assuming an executable program linked with a base address of `0x80000000` (2 Gbytes).

Although the file offsets and virtual addresses are congruent modulo 4 Kbytes for both text and data, up to four file pages can hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.

Member	Text	Data
`p_type`	`PT_LOAD`	`PT_LOAD`
`p_offset`	`0x0`	`0x1bf58`
`p_vaddr`	`0x80000000`	`0x8001cf58`
`p_paddr`	unspecified	unspecified
`p_filesz`	`0x1bf58`	`0x17c4`
`p_memsz`	`0x1bf58`	`0x2578`
`p_flags`	`PF_R+PF_X`	`PF_R+PF_W`
`p_align`	`0x1000`	`0x1000`

Table 3.1.: Program header segments

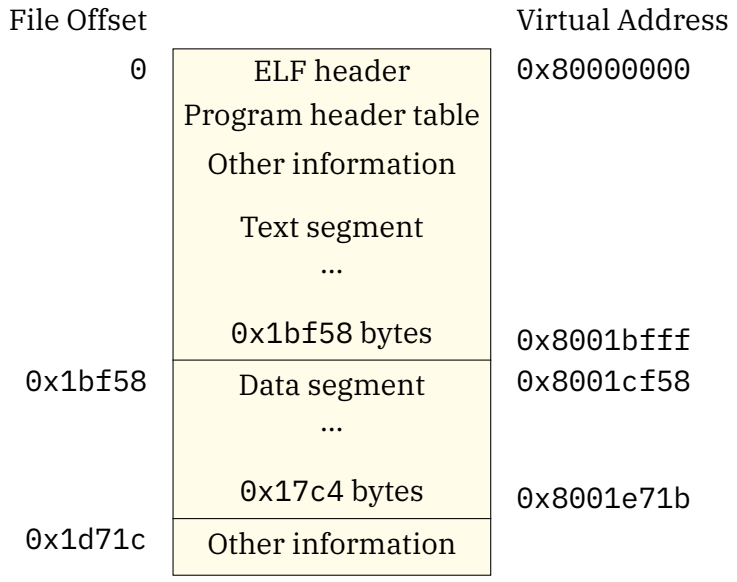

Figure 3.1.: Executable file example

- The last text page may hold a copy of the beginning of data.
- The first data page may have a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces memory permissions as if each segment were complete and separate; segment addresses are adjusted to ensure that each logical page in the address space has a single set of permissions. In the example in table 3.1 the file region holding the end of text and the beginning of data is mapped twice; at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if the last data page of a file includes information beyond the logical memory page, the extraneous data must be set to zero by the loader, rather than to the unknown contents of the executable file. “Impurities” in the other three segments are not logically part of the process image, and whether the system clears them is unspecified. The memory image for the program in table 3.1 is presented in figure 3.2.

One aspect of segment loading differs between executable files and shared objects. Executable file segments may contain absolute code. For the process to execute correctly, the segments must reside at the virtual addresses assigned when building the executable file, with the system using the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This allows a segment’s virtual address to change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the “relative positions” of the segments. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file.

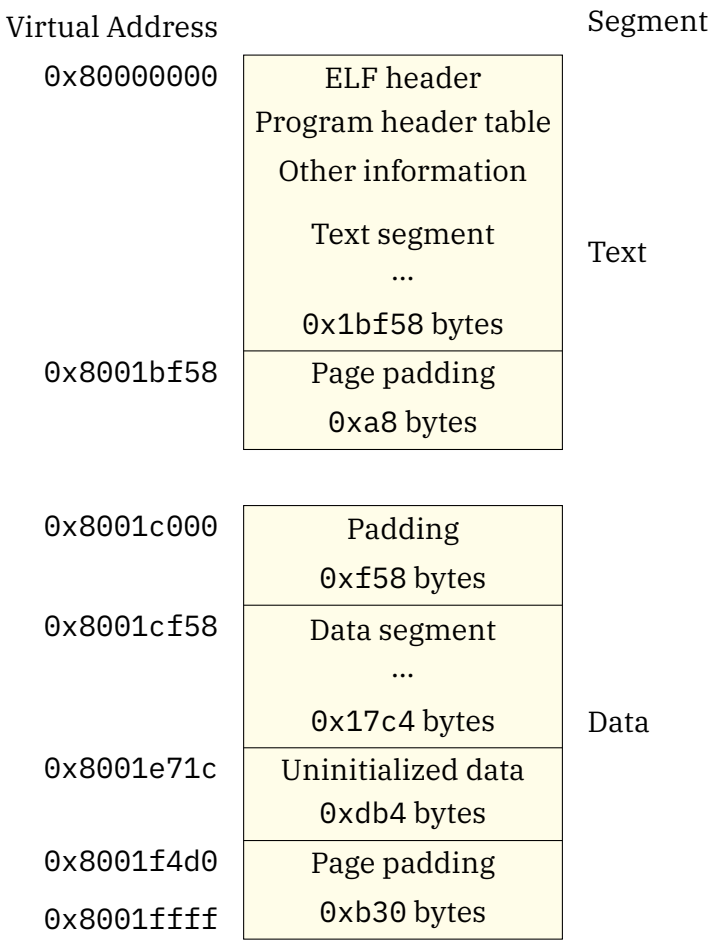

Figure 3.2.: Process image segments

Source	Text	Data	Base Address
File	0x00000000000	0x0000002a400	
Process 1	0x20000000000	0x2000002a400	0x20000000000
Process 2	0x20000010000	0x2000003a400	0x20000010000
Process 3	0x20000020000	0x2000004a400	0x20000020000
Process 4	0x20000030000	0x2000005a400	0x20000030000

Table 3.2.: Shared object segment example for 42-bit address space

Table 3.2 shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

3.2. Dynamic Linking

3.2.1. Dynamic Section

Dynamic section entries give information to the dynamic linker. Some of this information is processor-specific, including the interpretation of some entries in the dynamic structure.

DT_PLTGOT The `d_ptr` field of this entry gives the address of the first byte in the Global Offset Table. See section 3.2.2 for more information.

DT_JMPREL This entry is associated with a table of relocation entries for the PLT. For s390x this entry is mandatory both for executable and shared object files. Moreover, the relocation table's entries must have a one-to-one correspondence with the PLT. The table of `DT_JMPREL` relocation entries is wholly contained within the `DT_RELA` referenced table. See section 3.2.4 for more information.

3.2.2. 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 GOT using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

When the dynamic linker creates memory segments for a loadable object file, it processes the relocation entries, some of which will be of type `R_390_GLOB_DAT`, referring to the GOT. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the GOT entries to the proper values. Although the absolute addresses are unknown when the linkage 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.

A GOT entry provides direct access to the absolute address of a symbol without compromising position-independence and sharability. Because the executable file and shared objects have separate GOTs, a symbol may appear in several tables. The dynamic linker processes all the GOT relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The dynamic linker may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nevertheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

The format and interpretation of the Global Offset Table is processor specific. For s390x the symbol `_GLOBAL_OFFSET_TABLE_` may be used to access the table. The symbol refers to the start of the `.got` section. The first three doublewords in the GOT are reserved:

- The doubleword at `_GLOBAL_OFFSET_TABLE_[0]` is set by the linkage editor 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 doublewords at `_GLOBAL_OFFSET_TABLE_[1...2]` are reserved for system use.

The Global Offset Table resides in the ELF `.got` section.

3.2.3. Function Addresses

References to a function address from an executable file and from the shared objects associated with the file must 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 linkage 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 linkage editor will place the address of the PLT entry for that function in its associated symbol table entry. See section 2.3.1 for details. 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 these rules:

- If the `st_shndx` field of the symbol table entry is not `SHN_UNDEF`, the dynamic linker has found a definition for the symbol and uses its `st_value` field as the symbol's address.
- If the `st_shndx` field is `SHN_UNDEF` and the symbol is of type `STT_FUNC` and the `st_value` field is not zero, the dynamic linker recognizes this entry as special and uses the `st_value` field as the symbol's address.

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

Some relocations are associated with PLT entries. These entries are used for direct function calls rather than for references to function addresses. These relocations are not treated specially as described above because the dynamic linker must not redirect PLT entries to point to themselves.

3.2.4. 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 linkage editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another, so instead it arranges for the program to transfer control to entries in the PLT. The dynamic linker determines the absolute addresses of the destinations and stores them in the GOT, from which they are loaded by the PLT entry. The dynamic linker can thus redirect the entries without compromising the position-independence and sharability of the program text. Executable files and shared object files have separate PLTs.

As mentioned above, a relocation table is associated with the PLT. The `DT_JMPREL` entry in the `_DYNAMIC` array gives the location of the first relocation entry. The relocation table entries match the PLT entries in a one-to-one correspondence (relocation table entry 1 applies to PLT entry 1 and so on). The relocation type for each entry shall be `R_390_JMP_SLOT`. The relocation offset shall specify the address of the GOT entry containing the address of the function, and the symbol table index shall reference the appropriate symbol.

To illustrate Procedure Linkage Tables, listing 3.1 shows how the linkage editor might initialize the PLT when linking a shared executable or shared object.

As described below, the dynamic linker and the program cooperate to resolve symbolic references through the PLT. Again, the details described below are for explanation only. The precise execution-time behavior of the dynamic linker is not specified.

1. The caller of a function in a different shared object transfers control to the start of the PLT entry associated with the function.
2. The first part of the PLT entry loads the address from the GOT entry associated with the function to be called. Control is transferred to the code referenced by the address. If the function has already been called at least once, or if lazy binding is not used, then the address found in the GOT is the address of the function.
3. If a function has never been called and lazy binding is used, the address in the GOT points to the second half of the PLT. The second half loads the offset in the symbol table associated with the called function. Control is then transferred to the special first entry of the PLT.
4. This first entry of the PLT entry (see listing 3.2) calls the dynamic linker, giving it the offset into the symbol table and the address of a structure that identifies the location of the caller.

```
*                                  # PLT for executables (not
                                   #   position-independent)
PLT1      BASR  1,0                # Establish base
BASE1     L     1,AGOTENT-BASE1(1) # Load address of the GOT entry
          L     1,0(0,1)           # Load function address from the
                                   #   GOT to r1
          BCR   15,1               # Jump to address
RET1      BASR  1,0                # Return from GOT first time
                                   #   (lazy binding)
BASE2     L     1,ASYMOFF-BASE2(1) # Load offset in symbol table to r1
          BRC   15,-x              # Jump to start of PLT
          .word 0                  # Filler
AGOTENT   .long ?                  # Address of the GOT entry
ASYMOFF   .long ?                  # Offset into the symbol table
*                                  # PLT for shared objects
                                   #   (position-independent)
PLT1      LARL  1,<fn>@GOTENT      # Load address of GOT entry in r1
          LG    1,0(1)             # Load function address from the
                                   #   GOT to r1
          BCR   15,1               # Jump to address
RET1      BASR  1,0                # Return from GOT first time
                                   #   (lazy binding)
BASE2     LGF   1,ASYMOFF-BASE2(1) # Load offset in symbol table to r1
          BRCL  15,-x              # Jump to start of PLT
ASYMOFF   .long ?                  # Offset into symbol table
```

Listing 3.1: Procedure Linkage Table example

```
*                               # PLT0 for static object (not
                                #   position-independent)
PLT0      ST    1,28(15)        # R1 has offset into symbol table
          BASR  1,0             # Establish base
BASE1     L     1,AGOT-BASE1(1) # Get address of GOT
          MVC   24(4,15),4(1)   # Move loader info to stack
          L     1,8(1)          # Get address of loader
          BR    1               # Jump to loader
          .word 0               # Filler
AGOT      .long got             # Address of GOT

                                # PLT0 for shared object
                                #   (position-independent)
PLT0      STG   1,56(15)        # R1 has offset into symbol table
          LARL  1,_GLOBAL_OFFSET_TABLE_
          MVC   48(8,15),8(1)   # move loader info (object struct
                                #   address) to stack
          LG    1,16(12)        # Entry address of loader in R1
          BCR   15,1            # Jump to loader
```

Listing 3.2: Special first entry in Procedure Linkage Table

5. The dynamic linker finds the real address of the symbol. It will store this address in the GOT entry of the function in the object code of the caller and it will then transfer control to the function.
6. Subsequent calls to the function from this object will find the resolved address in the first half of the PLT entry and will transfer control directly without invoking the dynamic linker.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If set to a nonempty string, the dynamic linker resolves the function call binding at load time, before transferring control to the program. In other words, the dynamic linker processes relocation entries of type `R_390_JMP_SLOT` during process initialization. If `LD_BIND_NOW` is not set, the dynamic linker evaluates PLT entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

Note: Lazy binding generally improves overall application performance because unused symbols do not incur the overhead of dynamic linking. Nevertheless, two situations make lazy binding undesirable for some applications:

1. The initial reference to a shared object function takes longer than subsequent calls because the dynamic linker intercepts the call to resolve the symbol, and some applications cannot tolerate this unpredictability.
2. If an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process

initialization, before the application receives control.

A. GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1. Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document

to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A.2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3. Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

A.6. Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7. Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.10. Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ”or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software

Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

B. Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only

and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data discussed herein is presented as derived under specific operating conditions. Actual results may vary.

The client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illus-

trate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided ”AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

> © (your company name) (year).
> Portions of this code are derived from IBM Corp. Sample Programs.
> © Copyright IBM Corp. 2001, 2021.

B.1. Trademarks

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at ”Copyright and trademark information” at `www.ibm.com/legal/copytrade.shtml`.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Bibliography

[1] Ulrich Drepper, “ELF Handling For Thread-Local Storage,” https://akkadia.org/drepper/tls.pdf; 2013

[2] “System V Application Binary Interface,” edition 4.1, http://www.sco.com/developers/devspecs/gabi41.pdf; 1997

[3] “System V Application Binary Interface,” chapters 4 and 5, latest snapshot, http://www.sco.com/developers/gabi/latest/contents.html; 2013

[4] “Using the GNU Compiler Collection,” vector extensions, https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

[5] “z/Architecture Principles of Operation,” IBM Publication No. SA22-7832-12; 2019

Index