

zSeries ELF Application Binary Interface Supplement

zSeries ELF Application Binary Interface Supplement

v1.02 Edition

Published 18 November 2002

Copyright © 2001, 2002 IBM Corporation

Copyright © 2002 Linux Foundation

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Linux is a trademark of Linus Torvalds.

LSB is a trademark of the Free Standards Group in the USA and other countries.

Table of Contents

Preface	v
1. Low-level system information	1
Machine interface	1
Processor architecture	1
Data representation	1
Function calling sequence	7
Registers	7
The stack frame	9
Parameter passing	11
Variable argument lists	13
Return values	14
Operating system interface	14
Virtual address space	14
Page size	14
Virtual address assignments	14
Managing the process stack	15
Coding guidelines	15
Processor execution modes	16
Exception interface	16
Process initialization	17
Registers	17
Process stack	17
Coding examples	20
Code model overview	21
Function prolog and epilog	22
Profiling	24
Data objects	24
Function calls	26
Branching	27
Dynamic stack space allocation	28
DWARF definition	31
2. Object files	33
ELF Header	33
Machine Information	33
Sections	33
Special Sections	33
Symbol Table	34
Relocation	34
3. Program loading and dynamic linking	39
Program Loading	39
Dynamic Linking	42
Dynamic Section	42
Global Offset Table	42
Function Addresses	43
Procedure Linkage Table	43
A.	47
GNU Free Documentation License	47
PREAMBLE	47
APPLICABILITY AND DEFINITIONS	47
VERBATIM COPYING	48
COPYING IN QUANTITY	48
MODIFICATIONS	49
COMBINING DOCUMENTS	50
COLLECTIONS OF DOCUMENTS	50
AGGREGATION WITH INDEPENDENT WORKS	51
TRANSLATION	51

TERMINATION	51
FUTURE REVISIONS OF THIS LICENSE	51
How to use this License for your documents	51
4. Notices.....	53
Programming interface information.....	54
Trademarks.....	54
B.	55
Index.....	57

Preface

This v1.02 edition, published on 18 November 2002, applies to version 2, release 2, modification 16 of the Linux kernel and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces LNUX-1007-02 published in July 2001.

Chapter 1. Low-level system information

Machine interface

This section describes the processor-specific information for the zSeries processors.

Processor architecture

[z/Architecture Principles of Operation] (SA22–7832) defines the zSeries architecture.

Programs intended to execute directly on the processor use the zSeries instruction set, and the instruction encoding and semantics of the architecture.

An application program can assume that all instructions defined by the architecture 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 could conform to the ABI.

In z/Architecture a processor runs in big-endian mode. (See the Section called *Byte ordering*.)

Data representation

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, or "Big-Endian" as it is sometimes called, means that the most significant byte of a structure is located in the lowest addressed byte position in a storage unit (byte 0).

Figure 1-1 to Figure 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.

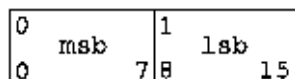


Figure 1-1. Bit and byte numbering in halfwords

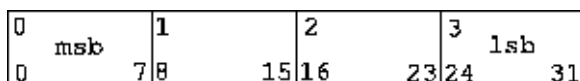


Figure 1-2. Bit and byte numbering in words

0	1	2	3
0 msb 7	8 15	16 23	24 31
4	5	6	7 lsb
32 39	40 47	48 55	56 63

Figure 1-3. Bit and byte numbering in doublewords

0	1	2	3
0 msb 7	8 15	16 23	24 31
4	5	6	7
32 39	40 47	48 55	56 63
8	9	10	11
64 71	72 79	80 87	88 95
12	13	14	15 lsb
96 103	104 111	112 119	120 127

Figure 1-4. Bit and byte numbering in quadwords

Fundamental types

Table 1-1 shows how ANSI C scalar types correspond to those of the zSeries processor. For all types a NULL pointer has the value zero (binary).

Table 1-1. Scalar types

Type	ANSI C	sizeof (bytes)	Alignment	type (zSeries)
Character	signed char char unsigned char	1	1	byte
Short	signed short short unsigned short	2	2	halfword
Integer	signed int int un- signed int enum	4	4	word
Long Long long	signed long long unsigned long long signed long long long long un- signed long long	8	8	doubleword
Pointer	any-type * any- type (*) ()	8	8	unsigned doubleword

Type	ANSI C	sizeof (bytes)	Alignment	type (zSeries)
Floating point	float	4	4	single precision (IEEE)
	double	8	8	double precision (IEEE)
	long double ¹	16	16	extended precision (IEEE)
¹ Compilers and systems may implement the long double data type in some other way, for performance reasons, using a compiler option. Examples of such formats could be two successive doubles or even a single double. Such usage does not conform to this ABI however, and runs the risk of passing a wrongly formatted floating-point number to another function as an argument. Programs using other formats should transform long double floating-point numbers to a conforming format before passing them.				

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 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 following examples (Figure 1-5 to Figure 1-9), member byte offsets (for the big-endian implementation) appear in the upper left corners.

Table 1-2.

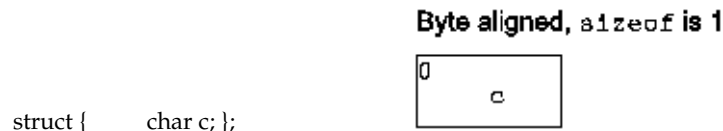


Figure 1-5. Structure smaller than a word

Table 1-3.

Word aligned, sizeof is 8		
0	c	1
		d
		2
		s
4	n	

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

Figure 1-6. No padding

Table 1-4.

Halfword aligned, sizeof is 4		
0	c	1
		pad
		2
		s

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

Figure 1-7. Internal padding

Table 1-5.

Doubleword aligned, sizeof is 24		
0	c	1
		pad
4	pad	
8	d	
12	d	
16	s	18
		pad
20	pad	

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

Figure 1-8. Internal and tail padding

Table 1-6.

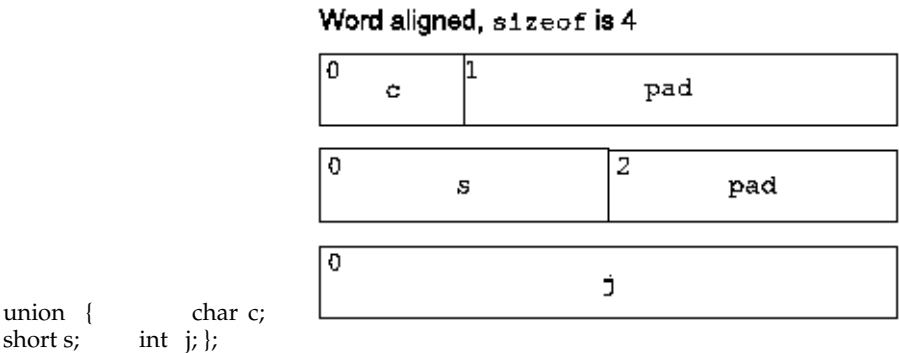


Figure 1-9. Union padding

Bit-fields

C struct and union definitions may have "bit-fields," defining integral objects with a specified number of bits (see Table 1-7).

Table 1-7. Bit fields

Bit-field type	Width n	Range
signed char char unsigned char	1 to 8	-2^1 to $2^1 - 1$ 0 to $2 - 1$ 0 to $2 - 1$
signed short short unsigned short	1 to 16	-2^1 to $2^1 - 1$ 0 to $2 - 1$ 0 to $2 - 1$
signed int int unsigned int enum	1 to 32	-2^1 to $2^1 - 1$ 0 to $2 - 1$ 0 to $2 - 1$
signed long long unsigned long long signed long long unsigned long long	1 to 64	-2^1 to $2^1 - 1$ 0 to $2 - 1$ 0 to $2 - 1$

"Plain" bit-fields (that is, those neither signed nor unsigned) always have

non-negative values. Although they may have type short, int or long (which can have negative values), bit-fields of these types have the same range as bit-fields of the same size with the corresponding unsigned type. 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 following examples (Figure 1-10 through Figure 1-15) show structure and union member byte offsets in the upper left corners. Bit numbers appear in the lower corners.

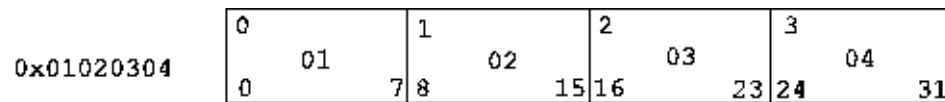


Figure 1-10. Bit numbering

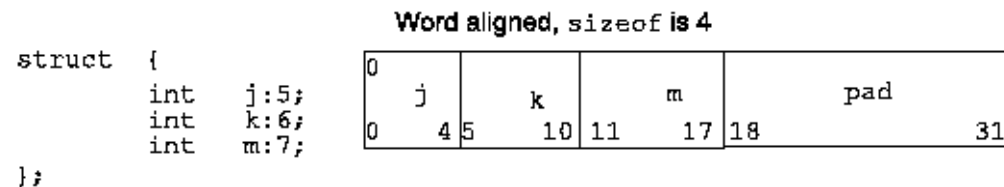


Figure 1-11. Left-to-right allocation

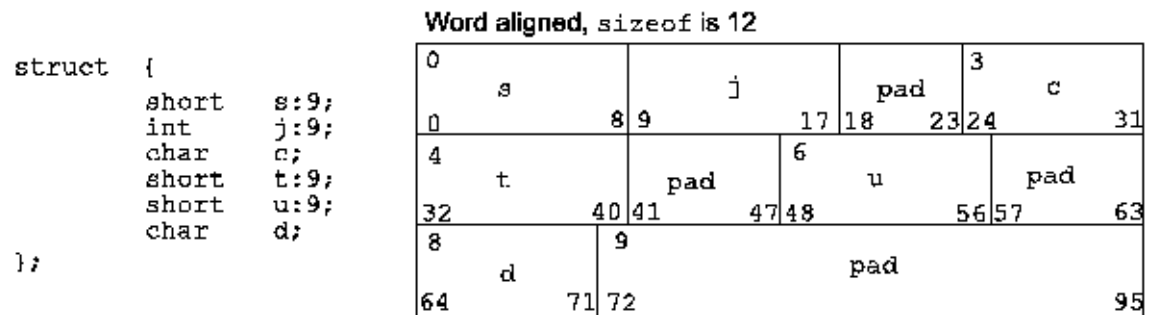


Figure 1-12. Boundary alignment

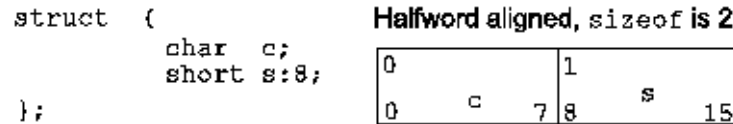


Figure 1-13. Storage unit sharing

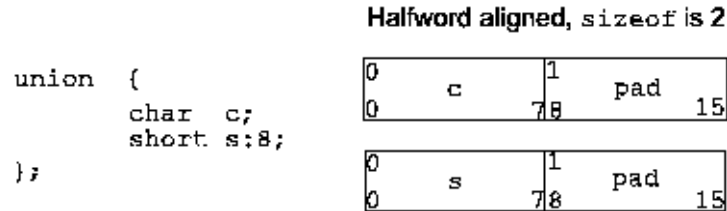


Figure 1-14. Union allocation

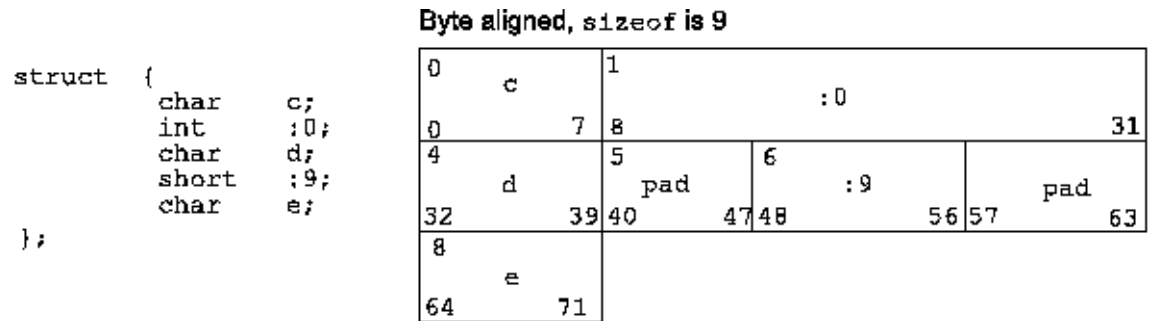


Figure 1-15. Unnamed bit fields

Function calling sequence

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

Registers

The ABI makes the assumption that the processor has 16 general purpose registers and 16 IEEE floating point registers. zSeries processors have these registers; each register is 64 bits wide. The use of the registers is described in the table below.

Table 1-8.

Register name	Usage	Call effect
r0, r1	General purpose	Volatile ¹

Register name	Usage	Call effect
r2	Parameter passing and return values	Volatile
r3, r4, r5	Parameter passing	Volatile
r6	Parameter passing	Saved ²
r7 - r11	Local variables	Saved
r12	Local variable, commonly used as GOT pointer	Saved
r13	Local variable, commonly used as Literal Pool pointer	Saved
r14	Return address	Volatile
r15	Stack pointer	Saved
f0, f2, f4, f6	Parameter passing and return values	Volatile
f1, f3, f5, f7	General purpose	Volatile
f8 – f15	General purpose	Saved
Access registers 0, 1	Reserved for system use	Volatile
Access registers 2-15	General purpose	Volatile

¹Volatile: These registers are not preserved across function calls.

²Saved: These registers belong to the calling function. A called function shall save these registers' values before it changes them, restoring their values before it returns.

- Registers r6 through r13, r15, and f8 through f15 are nonvolatile; that is, they "belong" to the calling function. A called function shall save these registers' values before it changes them, restoring their values before it returns.
- Registers r0 through r5, r14 and f0 through f7 are volatile; that is, they are not

preserved across function calls.

- 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.

The following registers have assigned roles in the standard calling sequence:

Table 1-9.

r12	Global Offset Table pointer. If a position-independent module uses cross-linking the compiler must point r12 to the GOT as described in the Section called <i>Dynamic Linking</i> in Chapter 3. If not this register may be used locally.
r13	Commonly used as the Literal Pool pointer. If the Literal Pool is not required this register may be used locally.
r14	This register will contain the address to which a called function will normally return. r14 is volatile across function calls.
r15	The stack pointer (stored in r15) will maintain an 8-byte alignment. It will always point to the lowest allocated valid stack frame, and will grow towards low addresses. The contents of the word addressed by this register may point to the previously allocated stack frame. If required it can be decremented by the called function – see the Section called <i>Dynamic stack space allocation</i> .

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.

Register usage

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

- 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.

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). This stack grows downwards from high addresses. Figure 1-16 shows the stack frame organization. SP in the figure denotes the stack pointer (general purpose register r15) passed to the called function on entry. Maintenance of the back chain

pointers is not a requirement of the ABI, but the storage area for these pointers must be allocated whether used or not.

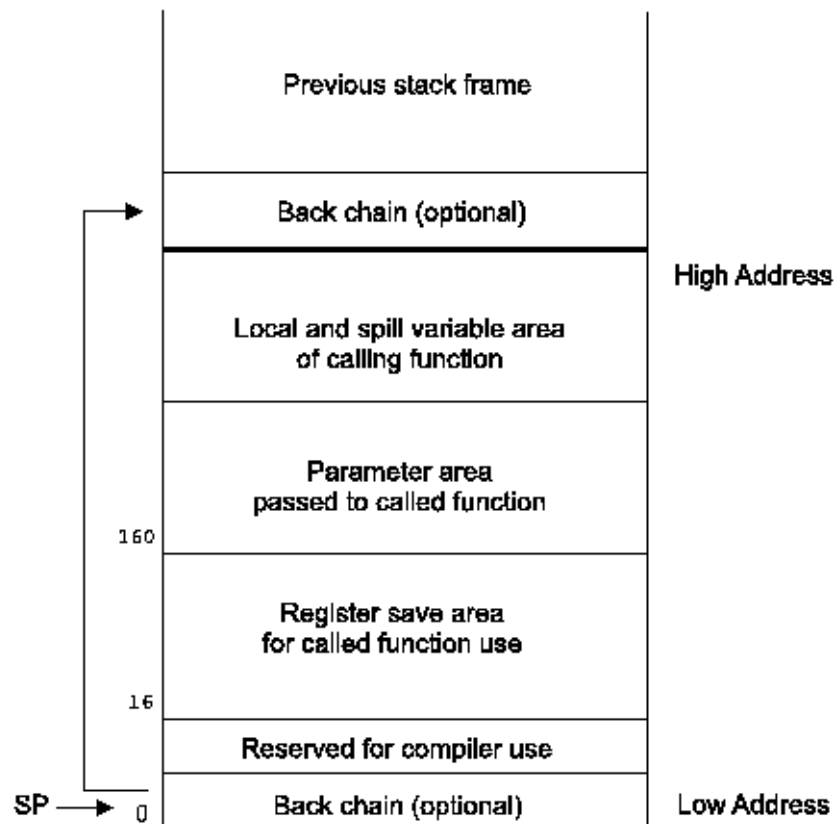


Figure 1-16. Standard stack frame

The format of the register save area created by the gcc compiler is:

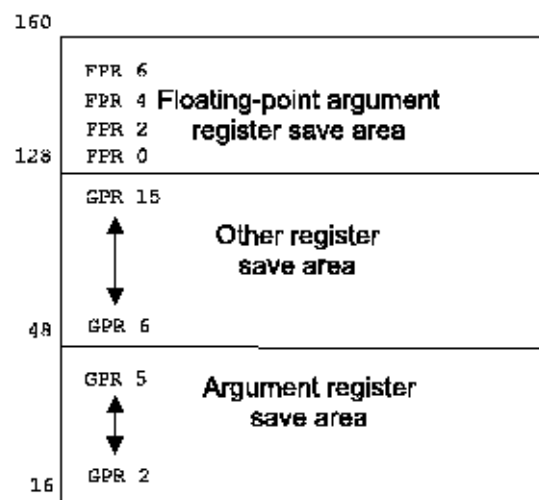


Figure 1-17. Register save area

The following requirements apply to the stack frame:

- The stack pointer shall maintain 8-byte alignment.
- The stack pointer points to the first word of the lowest allocated stack frame. If the "back chain" is implemented this word will point to the previously allocated stack frame (towards higher addresses), except for the first stack frame, which shall have a back chain of zero (NULL). The stack shall grow downwards, in other words towards lower addresses.
- The called function may create a new stack frame by decrementing the stack pointer by the size of the new frame. This is required if this function calls further functions. The stack pointer must be restored prior to return.
- The parameter list area shall be allocated by the caller and shall be large enough to contain the arguments that the caller stores in it. Its contents are not preserved across calls.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size.

The stack space for the register save area and back chain must be allocated by the caller. The size of these is 160 bytes.

Except for the stack frame header and any padding necessary to make the entire frame a multiple of 8 bytes in length, a function need not allocate space for the areas that it does not use. If a function does not call any other functions and does not require any of the other parts of the stack frame, it need not establish a stack frame. Any padding of the frame as a whole shall be within the local variable area; the parameter list area shall immediately follow the stack frame header, and the register save areas shall contain no padding.

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.

For Linux for zSeries, the following applies:

- General registers r2 to r6 are used for integer values.
- Floating point registers f0, f2, f4 and f6 are used for floating point values.

If there are more than five integral values or four floating point values, the rest of the arguments are passed on the stack 160 bytes above the initial stack pointer.

Beside these general rules the following rules apply:

- char, short, int, long and long long are passed in general registers.
- Structures equivalent to a floating point type are passed in floating point registers. A structure is equivalent to a floating point type if and only if it has exactly one member, which is either of floating point type or itself a structure equivalent to a floating point type.
- Structures with a size of 1, 2, 4, or 8 bytes which are not equivalent to a floating point type are passed as integral values.
- All other structures are passed by reference. If needed, the called function makes a copy of the value.
- Complex numbers are passed as structures.

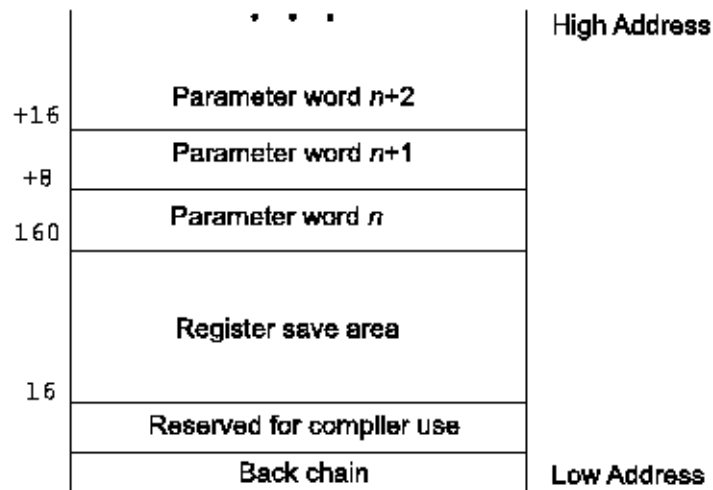


Figure 1-18. Parameter list area

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 purpose register, and *starg* is the address of the next available stack argument word.

INITIALIZE

Set *fr*=0, *gr*=2, and *starg* to the address of parameter word 1.

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 single length floating point type,
- A double length floating point type.
- A structure equivalent to a floating point type.

If *fr*>6, that is, if there are no more available floating-point registers, go to OTHER. Otherwise, load the argument value into floating-point register *fr*, set *fr* to *fr*+2, 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 (char, short, int, long, long long, enum).
- A pointer to an object of any type.
- A struct or a union of 1, 2, 4 or 8 bytes which is not a structure equivalent to a floating point type.
- A struct or union of another size, or a long double, any of which shall be passed as a pointer to the object, or to a copy of the object 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 OTHER. Otherwise load the argument value into general register gr , set gr to $gr+1$, and go to SCAN. Values shorter than 64 bits are sign- or zero-extended (as appropriate) to 64 bits.

OTHER

Arguments not otherwise handled above are passed in the parameter words of the caller's stack frame. SIMPLE_ARGS, as defined above, are considered to have size of 8 bytes, where simple integer types shorter than 8 bytes are signed or zero-extended (as appropriate) to 8 bytes, and other arguments of size less than 8 bytes will be placed right-justified into a 8 byte slot. float and double arguments are considered to have a size of 8 bytes, where float arguments will be placed right-justified into an 8 byte slot.

The contents of registers and words which are skipped by the above algorithm for alignment purposes (padding) are undefined.

As an example, assume the declarations and the function call shown in Figure 1-19. The corresponding register allocation and storage would be as shown in Table 1-10.

```
int i, j, k, l;

long long ll;

double f, g, h;

int m;

x = func(i, j, g, k, l, ll, f, h,
m);
```

Figure 1-19. Parameter passing example

Table 1-10. Parameter passing example: Register allocation

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

Variable argument lists

Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments are passed on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions have never been portable, but they have worked on many implementations. However, they do not work on z/Architecture because some arguments are passed in regis-

ters. Portable C programs use the header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists on zSeries and other machines as well.

Return values

In general, arguments are returned in registers, as described in Table 1-11.

Table 1-11. Registers for return values

Type	Returned in register:
char, short, int, long and long long	general register 2 (r2)
double and float	floating point register 0 (f0)

Functions shall return float or double values in f0, with float values rounded to single precision. Functions shall return values of type int, long, long long, enum, short and char, or a pointer to any type as unsigned or signed integers as appropriate, zero- or sign-extended to 64 bits if necessary, in r2.

Values of type long double and structures or unions are returned in a storage buffer allocated by the caller.

Operating system interface

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.

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.

Virtual address assignments

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

Figure 1-20 shows the virtual address configuration on the zSeries 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 Chapter 2 and Chapter 3). The process' 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.

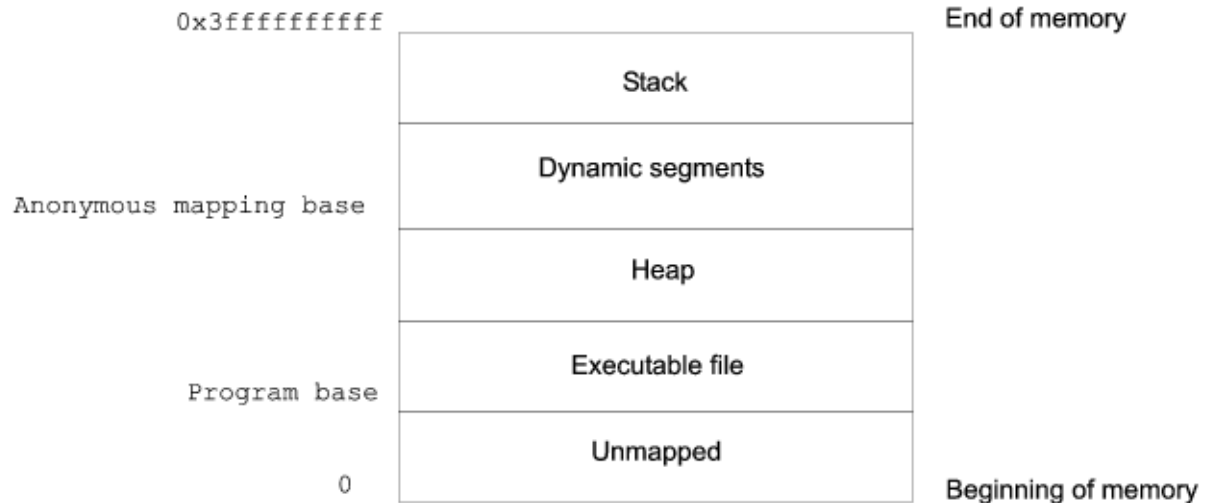


Figure 1-20. 42-bit virtual address configuration

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 the Section called *Exception interface*.

Although applications may control their memory assignments, the typical arrangement follows the diagram above. When applications let the system choose addresses for dynamic segments (including shared object segments), the system will prefer addresses in the upper half of the address space (for a 42-bit address space this means addresses above 1 TByte).

Managing the process stack

The section the Section called *Process initialization* 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.

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 receives 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.

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 an supervisor call (svc) instruction to change execution modes.

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

Exception interface

The z/Architecture exception mechanism allows the processor to change to supervisor state as a result of six different causes: system calls, I/O interrupts, external interrupts, machine checks, restart interruptions or program checks (unusual conditions arising in the execution of instructions).

When exceptions occur:

1. information (such as the address of the next instruction to be executed after control is returned to the original program) is saved,
2. program control passes from user to supervisor level, and
3. software continues execution at an address (the exception vector) predetermined for each exception.

Exceptions may be synchronous or asynchronous. Synchronous exceptions, being caused by instruction execution, can be explicitly generated by a process. The operating system handles an exception either by completing the faulting operation in a manner transparent to the application or by delivering a signal to the application. The correspondence between exceptions and signals is shown in Table 1-12.

Table 1-12. Exceptions and Signals

Exception Name	Signal	Examples
Illegal instruction	SIGILL	Illegal or privileged instruction, Invalid instruction form, Optional, unimplemented instruction

Storage access	SIGSEGV	Unmapped instruction or data location access, Storage protection violation
Alignment	SIGBUS	Invalid data item alignment, Invalid memory access
Breakpoint	SIGTRAP	Breakpoint program check
Floating exception	SIGFPE	Floating point overflow or underflow, Floating point divide by zero, Floating point conversion overflow, Other enabled floating point exceptions

The signals that an exception may give rise to are SIGILL, SIGSEGV, SIGBUS, SIGTRAP, and SIGFPE. If one of these signals is generated due to an exception when the signal is blocked, the behavior is undefined.

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 in the way described in Figure 1-21:

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

Figure 1-21. Declaration for `main`

Briefly, `argc` is a non-negative argument count; `argv` is an array of argument strings, with `argv[argc] == 0`, and `envp` is 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.

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 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:

Table 1-13.

r15	The initial stack pointer, aligned to a 8-byte boundary and pointing to a stack location that contains the argument count (see the Section called <i>Process stack</i> 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

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 purpose register r15. Data in the stack segment at addresses below the stack pointer contain undefined values.

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 Figure 1-22.

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

Figure 1-22. Auxiliary vector structure

The structures are interpreted according to the `a_type` member, as shown in Table 1-14.

Table 1-14. Auxiliary Vector Types, `a_type`

Name	Value	a_un
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr
AT_NOTELF	10	a_val
AT_UID	11	a_val
AT_EUID	12	a_val

AT_GID	13	a_val
AT_EGID	14	a_val

a_type auxiliary vector types are described in 'Auxiliary Vector Types Description' below.

Auxiliary Vector Types Description

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.

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_PHEMT, AT_PHNUM and AT_ENTRY must also be present. See the section Chapter 3 for more information about the program header table.

AT_PHEMT

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.

Other auxiliary vector types are reserved. No flags are currently defined for `AT_FLAGS` on the zSeries architecture.

When a process receives control, its stack holds the arguments, environment, and auxiliary vector 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 1-23.

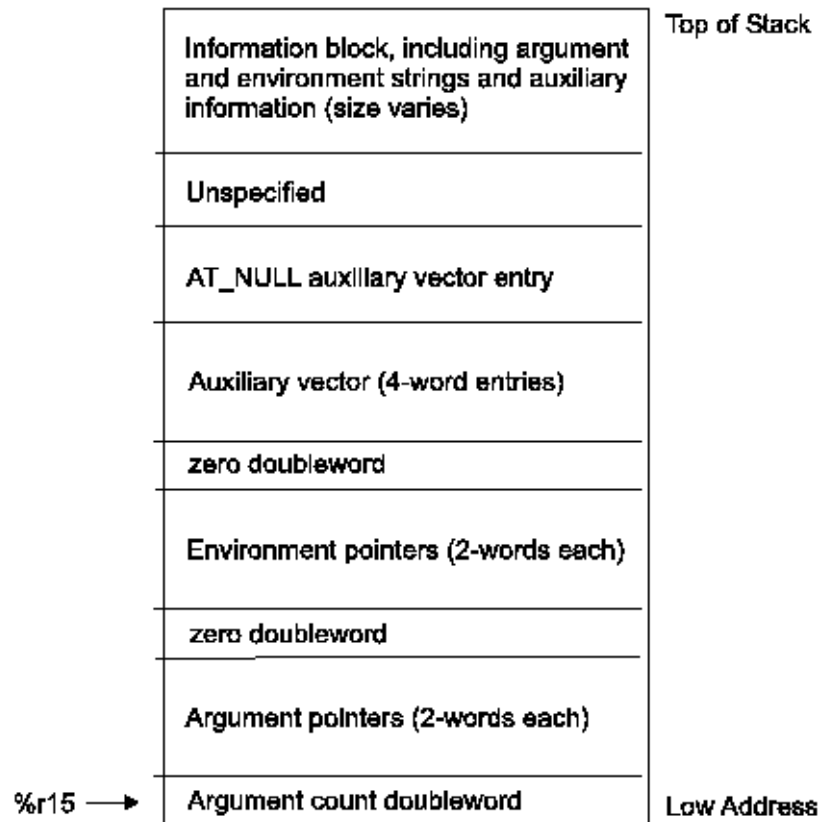


Figure 1-23. Initial Process Stack

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

Absolute code

Instructions can hold absolute addresses under this model. To execute properly, the program must be loaded at a specific virtual address, making the program's absolute addresses coincide with the process' 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.

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 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.

Function prolog and epilog

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

Prolog

The prolog 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 prolog:

- Save all registers used within the function which the calling function assumes to be non-volatile.
- Set up the base register for the literal pool.
- 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.
- Set up the GOT pointer if the compiler is generating position independent code.
(A function that is position independent will probably want to load a pointer to the GOT 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).

Epilog

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

Prolog and epilog example

```
.section .rodata
```

```

.align 2
.LC0:
.string "hello, world\n"
.text
.align 4
.globl main
.type main,@function
main:
                                # Prolog
    STMG 11,15,88(15)    # Save callers registers
    LARL 13,LT0_0        # Load literal pool pointer
.section .rodata            # Switch for literal pool
    .align 2            # to read-only data section
.LT0_0:
.LC2:
    .quad 65536
.LTN0_0:
.text                        # Back to text section
    LGR 1,15            # Load stack pointer in GPR 1
    AGHI 15,-160        # Allocate stack space
    STG 1,0(15)        # Store backchain
                                # Prolog end
    LARL 2,LC0
    LG 3,LC2-LT0_0(13)
    BRASL 14,printf
    LGHI 2,0
                                # Epilog
    LG 4,272(15)        # Load return address
    LMG 11,15,248(15)    # Restore registers
    BR 4                # Branch back to caller

```

Epilog end

Figure 1-24. Prolog and epilog example

Profiling

This section shows a way of providing profiling (entry counting) on zSeries systems. An ABI-conforming system is not required to provide profiling; however if it does this is one possible (not required) implementation.

If a function is to be profiled it has to call the `_mcount` routine after the function prolog. This routine has a special linkage. It gets an address in register 1 and returns without having changed any register. The address is a pointer to a word-aligned one-word static data area, initialized to zero, in which the `_mcount` routine is to maintain a count of the number of times the function is called.

For example Figure 1-25 shows how the code after the function prolog may look.

```

        STMG    7,15,56(15)    #
Save callers registers
        LGR     1,15           # Stack pointer
        AGHI    15,-160        # Allocate new
        STG     1,0(15)        # Save backchain
        LGR     11,15          # Local stack pointer
.data
        .align 4
.LP0:   .quad   0              # Profile counter
.text
                                # Function profiler
        STG     14,8(15)        # Preserve r14
        LARL    1,LP0          # Load address of profile counter
        BRASL   14,_mcount     # Branch to _mcount
        LG      14,8(15)        # Restore r14

```

Figure 1-25. Code for profiling

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 zSeries 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 normally has a literal pool that holds the addresses of data objects used by the function. Register 13 is set up in the function prolog 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 register 13 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. See Figure 1-26 for an example.

Figure 1-26 through Figure 1-28 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 code in the figures contains many redundant operations as it is only intended to show how each C statement could have been compiled independently of its context. The function prolog is not shown, and it is assumed that it has loaded the address of the literal pool in register 13.

Table 1-15.

C	zSeries machine instructions (Assembler)
extern int src; extern int dst; extern int *ptr; dst = src; ptr = &dst; *ptr = src;	<div>LARL 1,dst LARL 2,src</div> <div>MVC 0(4,1),0(2) LARL 1,ptr LARL 2,dst</div> <div>STG 2,0(1) LARL 2,ptr LG 1,0(2)</div> <div>LARL 2,src MVC 0(4,1),0(2)</div>

Figure 1-26. Absolute addressing

Table 1-16.

C	zSeries machine instructions (Assembler)
extern int src; extern int dst; extern int *ptr; dst = src; ptr = &dst; *ptr = src;	<div>LARL 12,_GLOBAL_OFFSET_TABLE_</div> <div>LG 1,dst@GOT12(12) LG 2,src@GOT12(12)</div> <div>LGF 3,0(2) ST 3,0(1)</div> <div>LARL 12,_GLOBAL_OFFSET_TABLE_</div> <div>LG 1,ptr@GOT12(12)</div> <div>LG 2,dst@GOT12(12) STG 2,0(1)</div> <div>LARL 12,_GLOBAL_OFFSET_TABLE_</div> <div>LG 2,ptr@GOT12(12) LG 1,0(2)</div> <div>LG 2,src@GOT12(12) LGF 3,0(2)</div> <div>ST 3,0(1)</div>

Figure 1-27. Small model position-independent addressing**Table 1-17.**

C	zSeries Assembler
extern int src; extern int dst; extern int *ptr; dst = src; ptr = &dst; *ptr = src;	LARL 2,dst@GOT LG 2,0(2) LARL 3,src@GOT LG 3,0(3) MVC 0(4,2),0(3) LARL 2,ptr@GOT LG 2,0(2) LARL 3,dst@GOT LG 3,0(3) STG 3,0(2) LARL 2,ptr@GOT LG 2,0(2) LG 1,0(2) LARL 3,src@GOT LG 3,0(3) MVC 0(4,1),0(3)

Figure 1-28. Large model position-independent addressing

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 14 is used as the first operand of BASR to hold the return address as shown in Figure 1-29.

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 the Section called *Procedure Linkage Table* in Chapter 3 for more details.

Table 1-18.

C	zSeries machine instructions (Assembler)
extern void func(); ex- tern void (*ptr)(); ptr = func; func(); (*ptr) ();	LARL 1,ptr LARL 2,func STG 2,0(1) BRASL 14,func LARL 1,ptr LG 1,0(1) BASR 14,1

Figure 1-29. Absolute function call

Table 1-19.

C	zSeries machine instructions (Assembler)
extern void func(); extern void (*ptr)(); ptr = func; func(); (*ptr) ();	LARL 12,_GLOBAL_OFFSET_TABLE_ LG 1,ptr@GOT12(12) LG 2,func@GOT12(12) STG 2,0(1) BRASL 14,func@PLT LARL 12,_GLOBAL_OFFSET_TABLE_ LG 1,ptr@GOT12(12) LG 1,0(1) BASR 14,1

Figure 1-30. Small model position-independent function call

Table 1-20.

C	zSeries machine instructions (Assembler)
extern void func(); extern void (*ptr)(); ptr = func; func(); (*ptr) ();	LARL 2,ptr@GOT LG 2,0(2) LARL 3,func@GOT LG 3,0(3) STG 3,0(2) BRASL 14,func@PLT LARL 2,ptr@GOT LG 2,0(2) LG 2,0(2) BASR 14,2

Figure 1-31. Large model position-independent function call

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 another self-relative jump is available with a range of 4 Gbytes (up to 2 Gbytes in either direction).

Table 1-21.

C	zSeries machine instructions (Assembler)
label: ... goto label; farlabel: goto farlabel;	.L01: ... BRC 15,.L01L02: ... BRCL 15,.L02

Figure 1-32. Branch instruction

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 following examples use several simplifying conventions to hide irrelevant details:

1. The selection expression resides in register 2.
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.

Table 1-22.

C	zSeries machine instructions (Assembler)
switch(j) { case 0: ... case 1: ... case 3: ... default: }	<pre> LGHI 1,3 CLGR 2,1 BRC 2,.Ldef SLLG 2,2,3 LARL 1,.Ltab LG 3,0(1,2) BR 3 .Ltab: .quad .Lcase0 .quad .Lcase1 .quad .Ldef .quad .Lcase3 </pre>

Figure 1-33. Absolute switch code

Table 1-23.

C	zSeries machine instructions (Assembler)
switch(j) { case 0: ... case 1: ... case 3: ... default: }	<pre> # Code LGHI 1,3 CLGR 2,1 # Literal pool .LT0: BRC 2,.Ldef SLLG 2,2,3 LARL 1,.Ltab LG 3,0(1,2) AGR 3,13 BR 3 .Ltab: .quad .Lcase0-.LT0 .quad .Lcase1- .LT0 .quad .Ldef-.LT0 .quad .Lcase3-.LT0 </pre>

Figure 1-34. Position-independent switch code, all models

Dynamic stack space allocation

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

Figure 1-35 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 list area holds "overflow" arguments passed in calls to other functions. (See the OTHER label in the Section called *Parameter passing*.) 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 list area begin

at a fixed offset (160) from the stack pointer, so this area must move when dynamic stack allocation occurs.

Data in the parameter list 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 variables 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 list 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 1-35 is a snapshot of the stack layout after the prolog code has dynamically extended the stack frame.

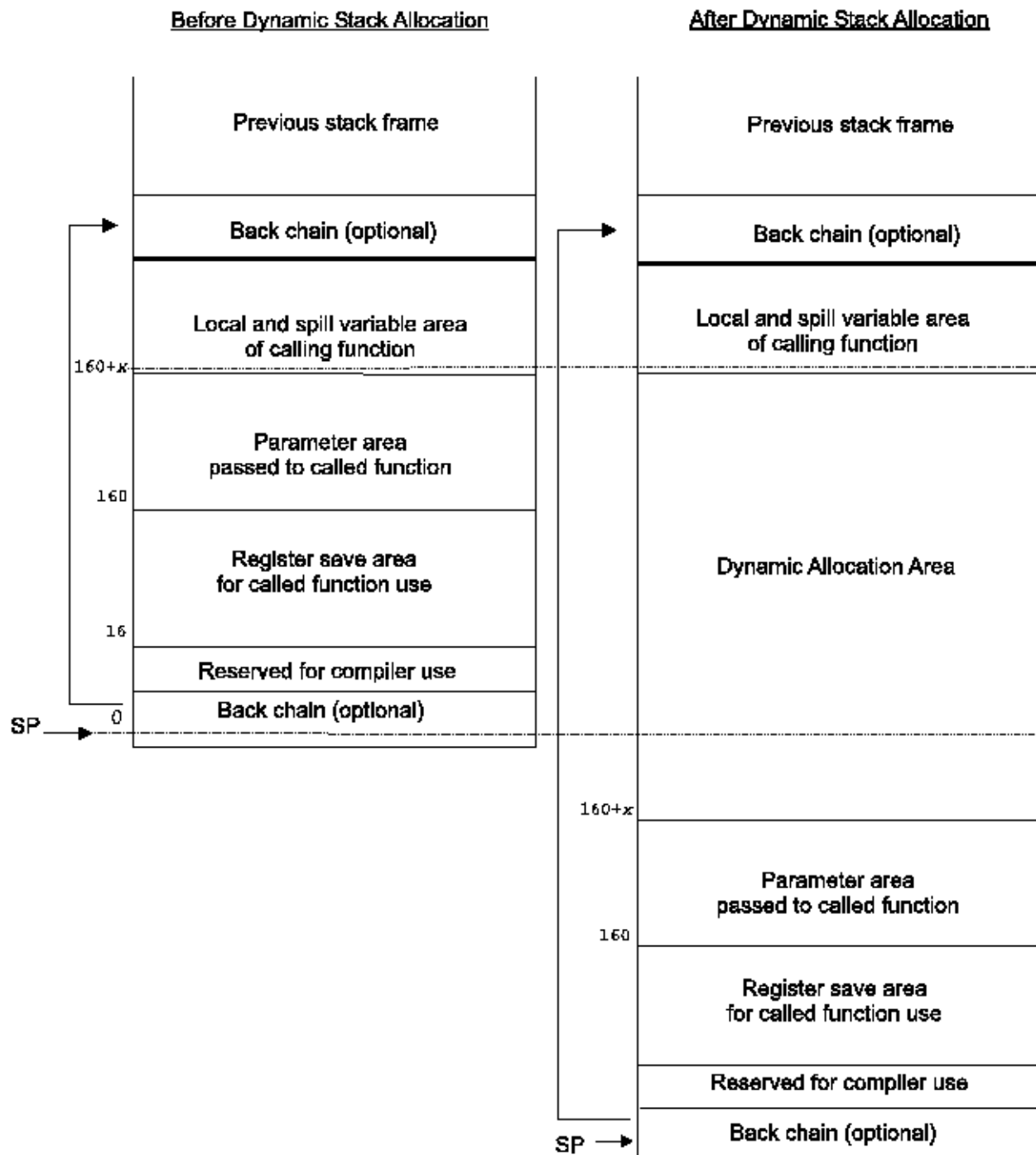


Figure 1-35. Dynamic Stack Space Allocation

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.
- 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.

DWARF definition

This section defines the "Debug with Arbitrary Record Format" (DWARF) debugging format for the zSeries processor family. The zSeries 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 zSeries processors in Table 1-24.

Table 1-24. DWARF register number mapping

DWARF number	zSeries register
0-15	gpr0-gpr15
16	fpr0
17	fpr2
18	fpr4
19	fpr6
20	fpr1
21	fpr3
22	fpr5
23	fpr7
24	fpr8
25	fpr10
26	fpr12
27	fpr14
28	fpr9
29	fpr11
30	fpr13
31	fpr15
32-47	cr0-cr15
48-63	ar0-ar15
64	PSW mask
65	PSW address

Chapter 2. Object files

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

ELF Header

Machine Information

For file identification in `e_ident` the zSeries processor family requires the values shown in Table 2-1.

Table 2-1. Auxiliary Vector Types Description

Position	Value	Comments
<code>e_ident[EI_CLASS]</code>	ELFCLASS64	For all 64bit implementations
<code>e_ident[EI_DATA]</code>	ELFDATA64MSB	For all Big-Endian implementations

The ELF header's `e_flags` field holds bit flags associated with the file. Since the zSeries 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`.

Sections

Special Sections

Various sections hold program and control information. The sections listed in Table 2-2 are used by the system and have the types and attributes shown.

Table 2-2. Special Sections

Name	Type	Attributes
<code>.got</code>	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
<code>.plt</code>	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_EXECINSTR

Special sections are described in Table 2-3.

Table 2-3. Special Sections Description

Name	Description
<code>.got</code>	This section holds the Global Offset Table, or GOT. See the Section called <i>Coding examples</i> in Chapter 1 and the Section called <i>Global Offset Table</i> in Chapter 3 for more information.

.plt	This section holds the Procedure Linkage Table, or PLT. See the Section called <i>Procedure Linkage Table</i> in Chapter 3 for more information.
------	--

Symbol Table

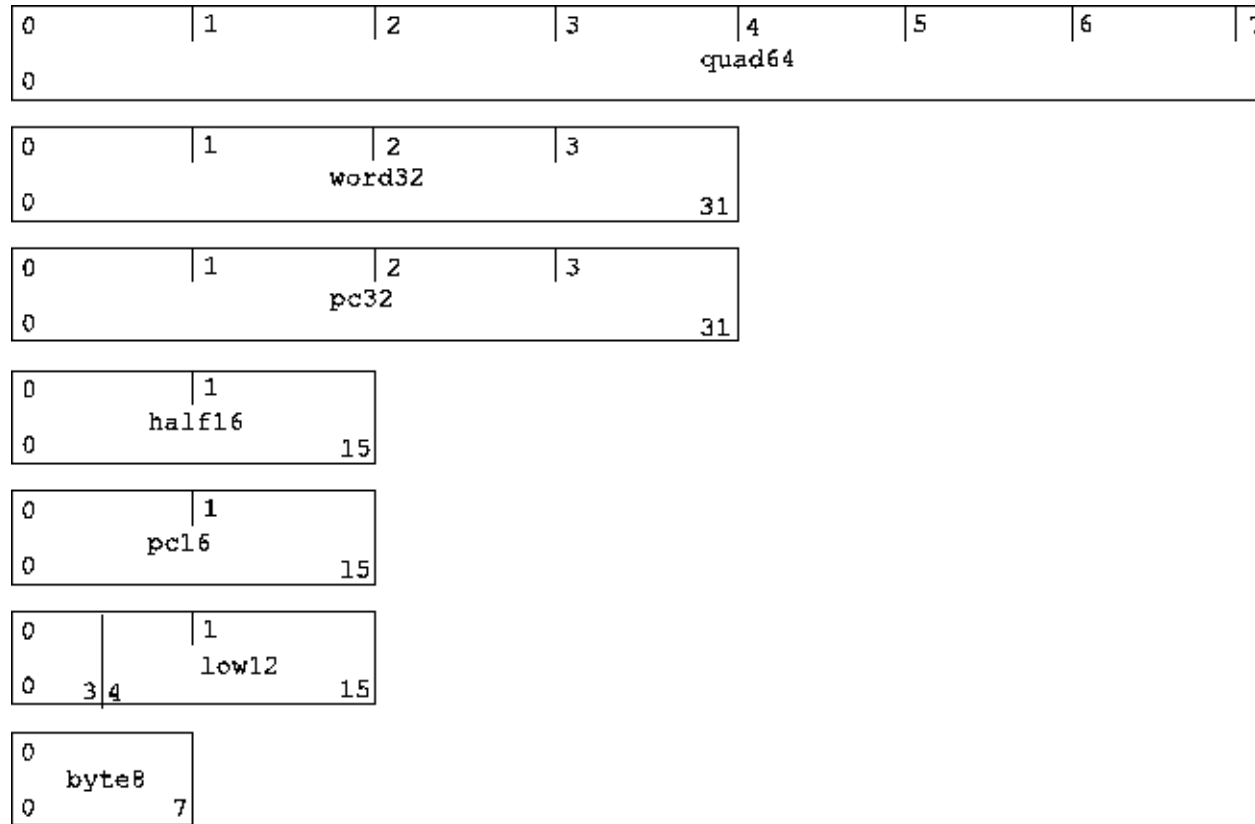
Symbol Values

If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for 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 the Section called *Function Addresses* in Chapter 3 for details.

Relocation

Relocation Types

Relocation entries describe how to alter the instruction and data relocation fields shown in Figure 2-1 (bit numbers appear in the lower box corners; byte numbers appear in the upper left box corners).

**Figure 2-1. Relocation Fields****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).

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 a 2-byte alignment. The 12 bit unsigned value is the displacement of a memory reference.

byte8

This specifies a 8-bit field with a 1-byte alignment.

Calculations in Table 2-4 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 2-4:

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 the Section called *Coding examples* in Chapter 1 and the Section called *Global Offset Table* in Chapter 3 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 the Section called *Procedure Linkage Table* in Chapter 3 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 the Section called *Coding examples* in Chapter 1 and the Section called *Global Offset Table* in Chapter 3 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.

Relocation entries apply to bytes, halfwords or words. 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 zSeries 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 2-4:

- "+" 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.
- For relocation type `half16`, the upper 48 bits of the value computed must be all ones or all zeroes. For relocation type `pc16`, the upper 47 bits of the value computed must be all ones or all zeroes and the lowest bit must be zero. For relocation type `pc32`, the upper 31 bits of the value computed must be all ones or all zeroes and the lowest bit must be zero. For relocation type `low12`, the upper 52 bits of the value computed must all be zero and for relocation type `byte8`, the upper 56 bits of the value computed must all be zero.
- Reference in a calculation to the value `G` or `O` implicitly creates a GOT entry for the indicated symbol and a reference to `L` implicitly creates a PLT entry.

Table 2-4. 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	(see below)
R_390_GLOB_DAT	10	quad64	$S + A$ (see below)
R_390_JMP_SLOT	11	none	(see below)
R_390_RELATIVE	12	quad64	$B + A$ (see below)
R_390_GOTOFF	13	quad64	$S + A - G$
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	$(S + A - P) \gg 1$
R_390_PLT16DBL	18	pc16	$(L + A - P) \gg 1$
R_390_PC32DBL	19	pc32	$(S + A - P) \gg 1$
R_390_PLT32DBL	20	pc32	$(L + A - P) \gg 1$
R_390_GOTPCDBL	21	pc32	$(G + A - P) \gg 1$
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
R_390_GOTENT	26	pc32	(G + O + A - P) >> 1

Table 2-5. Relocation type descriptions

Name	Description
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 Procedure Linkage Table entry. The dynamic linker modifies the PLT entry to transfer control to the designated symbol's address (see the Section called <i>Procedure Linkage Table</i> in Chapter 3).
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 relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

Chapter 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.

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 zSeries 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).

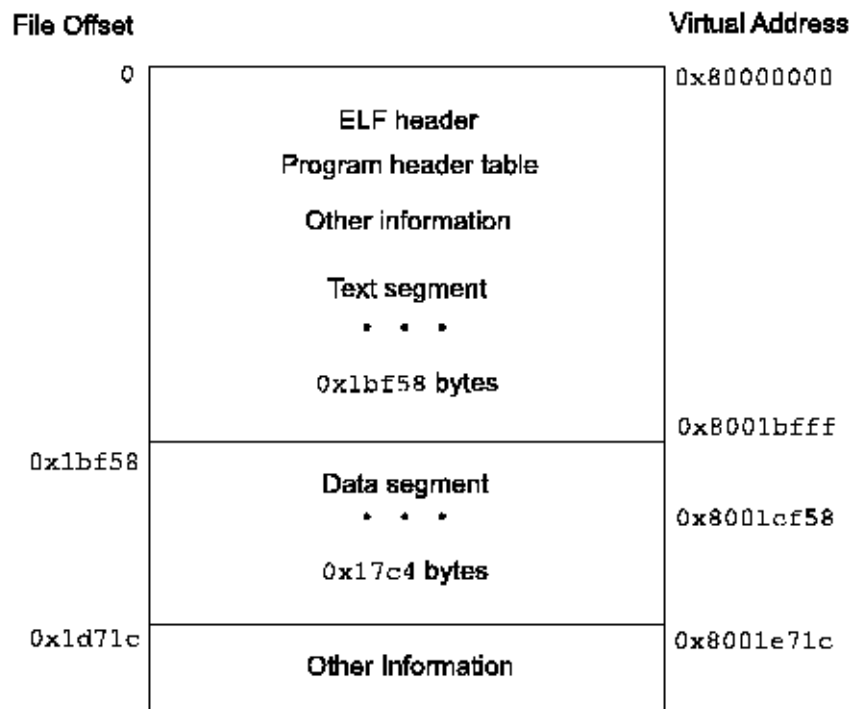


Figure 3-1. Executable File Example

Table 3-1. Program Header Segments

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

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.
- 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.

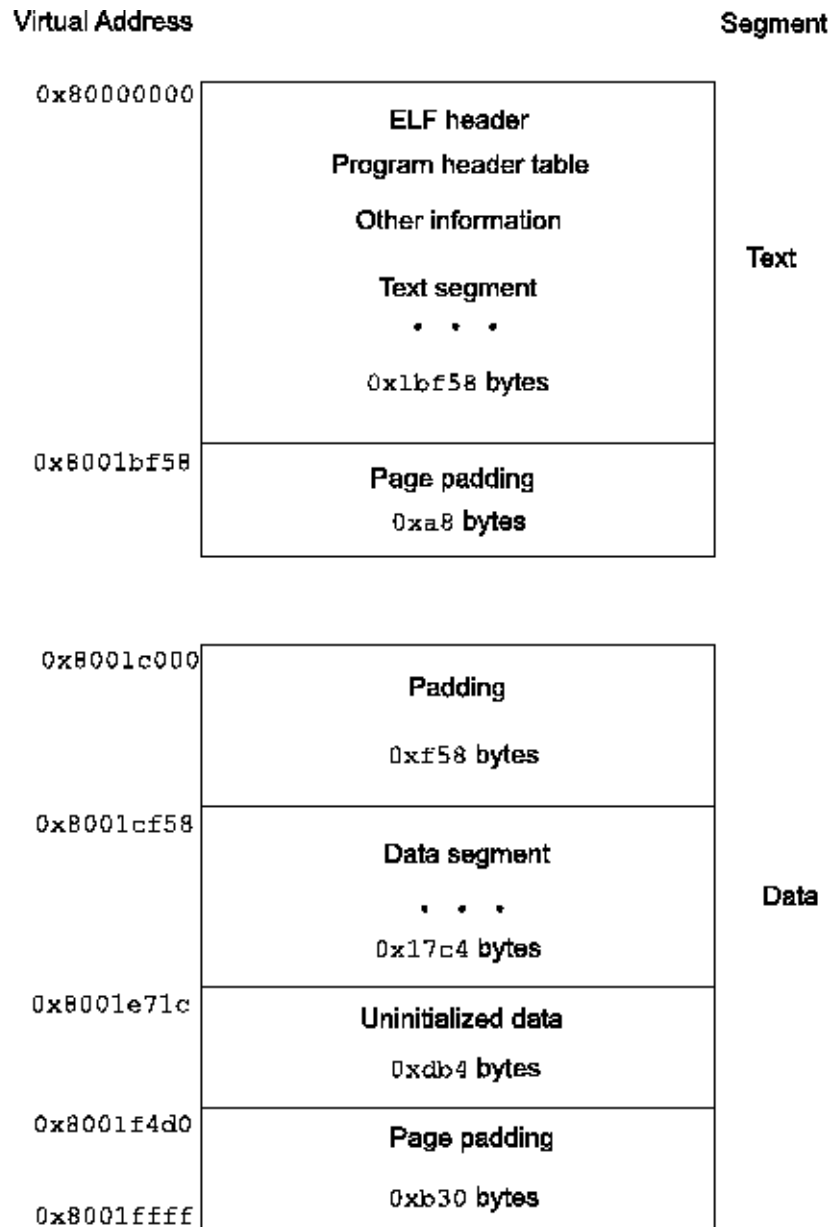


Figure 3-2. Process Image Segments

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. 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.

Table 3-2. Shared Object Segment Example for 42-bit address space

Source	Text	Data	Base Address
File	0x00000000200	0x0000002a400	
Process 1	0x20000000000	0x2000002a400	0x20000000000
Process 2	0x20000010000	0x2000003a400	0x20000010000
Process 3	0x20000020000	0x2000004a400	0x20000020000
Process 4	0x20000030000	0x2000005a400	0x20000030000

Dynamic Linking

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 Procedure Linkage Table (.PLT in the Section called *Procedure Linkage Table*).

DT_JMPREL

This entry is associated with a table of relocation entries for the PLT. For zSeries 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 the Section called *Procedure Linkage Table* for more information.

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 zSeries the symbol `_GLOBAL_OFFSET_TABLE_` may be used to access the table. The symbol refers to the start of the `.got` section. Two words in the GOT are reserved:

- The word 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 word at `_GLOBAL_OFFSET_TABLE_[1]` is reserved for future use.

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

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 the Section called *Symbol Values* in Chapter 2 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.

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, Figure 3-3 shows how the linkage editor might initialize the PLT when linking a shared executable or shared object.

```
*                               # 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

```

Figure 3-3. Procedure Linkage Table Example

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. The control is transferred to the code referenced by the address. If the function has already been called at least once, or 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 then 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 (Figure 3-4) 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.
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.

```

*                # 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
```

Figure 3-4. Special first entry in Procedure Linkage Table

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is not null 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 null 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.

Appendix A.

GNU Free Documentation License

A current copy of the license can be found at:

http://www.linuxbase.org/spec/refspecs/LSB_1.2.0/gLSB/gfdl.html

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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. Secondly, 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.

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.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, 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.

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.

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.

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."

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.

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.

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.

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.

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.

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 (c) 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.

Chapter 4. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information about 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 give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

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

IBM World Trade Asia Corporation

Licensing

2-31 Roppongi 3-chome, Minato-ku

Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: 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 states 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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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

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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate 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. 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 IBM's application programming interfaces.

Programming interface information

This book contains information and examples which are not intended to be used as a programming interface of Linux for zSeries.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

Table 4-1.

IBM	ESA/390
S/390	System/390
zSeries	z/Architecture

Linux is a registered trademark of Linus Torvalds and others.

Other company, product, and service names may be trademarks or service marks of others.

Appendix B.

Related publications:

- [*z/Architecture Principles of Operation*]: SA22–7832
- [*System V Application Binary Interface*]

Appendix B.

Index

