3 Low-Level System Information

3.1 Introduction

This chapter defines the rules to which language processors must adhere. These rules will allow language processors to do the following:

- Generate conforming function-code calling sequences (including argument-passing conventions, register allocation conventions, and support for dynamic binding)
- Allow access of program global data between code modules that have been written in different source languages (rules for common data types only are defined)
- Mix object modules generated by language processors from different vendors

This chapter defines low-level system information such as data formats and linkage conventions. The following topics are covered:

3.2 Instruction Set

Conforming new applications must be compiled using only the COMMON mode PowerPC instruction set. The COMMON mode instruction set is the intersection set of instructions from the POWER (RS1) and PowerPC User Instruction Sets.

For information on the PowerPC architecture and instruction set, refer to *PowerPC Architecture, First Edition*; IBM; May, 1993; customer reorder number 52G7487.

For information on the POWER instruction set, refer to *AIX Version 4.1 for RISC System/6000 Assembler Language Reference*; customer reorder number SC23−2642.

3.2.1 Required Instructions for Application Compatibility

ABI-conformant system must support all COMMON MODE instructions for the PowerPC or POWER implementation. They are specified in *PowerPC Architecture, First Edition* and *AIX Version 4.1 RISC System/6000 Assembler Language Reference.* The ABI does not dictate whether an instruction must be implemented in hardware or emulated in software, nor does it place constraints on performance.

3.2.1.1 Required POWER Instructions

The following is a list of POWER required instructions that are not supported by the hardware on PowerPC architecture systems. Therefore the required instruction must be supported with software, and emulated in the PowerPC systems. The MQ register exists on some PowerOpen platforms but not on all and therefore must be supported in the PowerPC architecture systems.

Table 3−1 Required POWER Instructions

Mnemonic	Instruction
abs	Absolute
clcs	Cache line compute size
clf	Cache line flush
dclst	Date cache line store
div	Divide
divs	Divide short
doz	Difference or zero
dozi	Difference or zero immediate
lscbx	Load string and compare byte extended
maskg	Mask generate
mfsr	Move from segment register
mfsri	Move from segment register indirect
maskir	Mask insert from register
mul	Multiply
nabs	Negative absolute
rlmi	Rotate left then mask insert
rrib	Rotate right and insert bit
sle	Shift left extended
slq	Shift left with MQ

Mnemonic	Instruction
sraiq	Shift right algebraic with immediate MQ
sraq	Shift right algebraic with MQ
sre	Shift right extended
srlq	Shift right long with MQ
srq	Shift right with MQ
srea	Shift right extended algebraic
sreq	Shift right extended with MQ
sriq	Shift right immediate with MQ
srliq	Shift right long immediate with MQ
sleq	Shift left extended with MQ
sliq	Shift left immediate with MQ
slliq	Shift left long immediate with MQ
sllq	Shift left long with MQ

Note: When all PowerOpen Applications have been compiled to use the common mode instruction set, the requirement for PowerOpen ABI conforming systems to support this set of POWER instructions will no longer exist and may at that time be removed from the ABI document.

3.2.1.2 Common Mode Instruction Set

Conforming new applications must be compiled using only the COMMON mode PowerPC instruction set. The COMMON mode instruction set is the intersection set of instructions from the POWER (RS1) and PowerPC User Instruction Sets.

The following table lists the POWER and PowerPC instruction set mnemonics for common mode instructions.

Table 3−2 COMMON MODE Instruction Set

Primary OP	Ext. OP	POWER	PowerPC
03		ti	twi
07		muli	mulli
08		sfi	subfic
10		cmpli	cmpli
11		cmpi	cmpi
12		ai	addic
13		ai.	addic.
14		cal	addi
15		cau	addis
16		bc	bc
18		b	b
19	0	mcrf	mcrf
19	16	bcr	bclr
19	33	crnor	crnor

Primary OP	Ext. OP	POWER	PowerPC
19	50	rfi	rfi
19	129	crandc	crandc
19	150	ics	isync
19	193	crxor	crxor
19	225	crnand	crnand
19	257	crand	crand
19	289	creqv	creqv
19	417	crorc	crorc
19	449	cror	cror
19	528	bcc	bcctr
20		rlimi	rlwimi
21		rlinm	rlwinm
23		rlnm	rlwnm
24		oril	ori
25		oriu	oris
26		xoril	xori
27		xoriu	xoris
28		andil.	andi.
29		andiu.	andis.
31	0	cmp	cmp
31	4	t	tw
31	8	sf	subfc
31	10	a	addc
31	19	mfcr	mfcr
31	23	lx	lwzx
31	24	sl	slw
31	26	cntlz	cntlzw
31	28	and	and
31	32	cmpl	cmpl
31	55	lux	lwzux
31	60	andc	andc
31	83	mfmsr	mfmsr
31	87	lbzx	lbzx
31	104	neg	neg
31	119	lbzux	lbzux
31	124	nor	nor
31	136	sfe	subfe
31	138	ae	adde
31	144	mtcrf	mtcrf

Primary OP	Ext. OP	POWER	PowerPC
31	146	mtmsr	mtmsr
31	151	stx	stwx
31	183	stux	stwux
31	200	sfze	subfze
31	202	aze	addze
31	215	stbx	stbx
31	232	sfme	subfme
31	234	ame	addme
31	235	muls	mullw
31	247	stbux	stbux
31	266	cax	add
31	279	lhzx	lhzx
31	284	eqv	eqv
31	311	lhzux	lhzux
31	316	xor	xor
31	343	lhax	lhax
31	375	lhaux	lhaux
31	407	sthx	sthx
31	412	orc	orc
31	439	sthux	sthux
31	444	or	or
31	476	nand	nand
31	512	mcrxr	mcrxr
31	534	lbrx	lwbrx
31	535	lfsx	lfsx
31	536	sr	srw
31	567	lfsux	lfsux
31	598	dcs	sync
31	599	lfdx	lfdx
31	631	lfdux	lfdux
31	661	stsx	stswx
31	662	stbrx	stwbrx
31	663	stfsx	stfsx
31	695	stfsux	stfsux
31	725	stsi	stswi
31	727	stfdx	stfdx
31	759	stfdux	stfdux
31	790	lhbrx	lhbrx
31	792	sra	sraw

Primary OP	Ext. OP	POWER	PowerPC
31	824	srai	srawi
31	918	sthbrx	sthbrx
31	922	exts	extsh
31	1014	dclz	dcbz
32		l	lwz
33		lu	lwzu
34		lbz	lbz
35		lbzu	lbzu
36		st	stw
37		stu	stwu
38		stb	stb
39		stbu	stbu
40		lhz	lhz
41		lhzu	lhzu
42		lha	lha
43		lhau	lhau
44		sth	sth
45		sthu	sthu
47		stm	stmw
48		lfs	lfs
49		lfsu	lfsu
50		lfd	lfd
51		lfdu	lfdu
52		stfs	stfs
53		stfsu	stfsu
54		stfd	stfd
55		stfdu	stfdu
63	0	fcmpu	fcmpu
63	12	frsp	frsp
63	18	fd	fdiv
63	20	fs	fsub
63	21	fa	fadd
63	25	fm	fmul
63	28	fms	fmsub
63	29	fma	fmadd
63	30	fnms	fnmsub
63	31	fnma	fnmadd
63	32	fcmpo	fcmpo
63	38	mtfsb1	mtfsb1

Primary OP	Ext. OP	POWER	PowerPC
63	40	fneg	fneg
63	64	mcrfs	mcrfs
63	70	mtfsb0	mtfsb0
63	72	fmr	fmr
63	134	mtfsfi	mtfsfi
63	136	fnabs	fnabs
63	264	fabs	fabs
63	583	mffs	mffs
63	711	mtfsf	mtfsf

3.3 POWER Compatibility and Low Memory Definition

The PowerPC architecture is similar to the POWER (RISC System/6000) architecture. Several POWER instructions were not included in the PowerPC instruction set, and new ones were added. For binary compatibility, a special mode of compilation and assembling exists that uses only those instructions that are common to both architectures. Operations that are not part of the intersection of the two instruction sets have been identified and are uniquely coded for each architecture. They are included as part of the low memory image. The proper emulation functions will be loaded based on the machine type as part of the operating system installation process. Language processors that support this mode of code generation must provide a facility to do so. They must also generate calls to these special functions instead of using the architecture-specific instructions.

Note that binary compatibility of existing applications (at the instruction-set level) that run on POWER-based products will be maintained for all PowerOpen ABI-conformant systems. However, binary compatibility of applications that are specifically compiled for PowerOpen ABI-conformance are not backward-compatible with POWER unless intersection mode compiling and assembling has been specified.

The following are general rules that apply to all special low-memory functions:

- Reachable by Branch and Link Absolute (**bla**) Instruction
- Do not use the TOC
- Do not alter any FPRs
- Do alter the LR

For performance reasons, compilers should generate **bla** instructions but with relocation information for the particular symbol being called — resolved relocation entries and symbol table entries. This isolates the compiler from the need to know the actual locations of these functions.

Name resolution of the absolute address will be done by the binder.

The following tables include the names and interfaces to the special functions. The application must have read (execute) permissions on these effective addresses.

Note: No special purpose registers are used by PowerPC, but POWER uses register MQ in each of the instances listed.

Table 3−3. The **_mulh** Function.

Address	0x00003100
Purpose	Calculate high-order 32 bits of integer product *arg1**arg2*
Input	R3=*arg1* (signed int) R4=*arg2* (signed int)
Output	R3=high-order 32 bits of *arg1**arg2*
GPRs altered	R3, R4
CR Fields altered	None

Table 3–4. The **_mull** Function.

Address	0x0x00003180
Purpose	Calculate 64 bits of integer product *arg1**arg2*
Input	R3=*arg1* (signed int) R4=*arg2* (signed int)
Output	R3=high-order 32 bits of *arg1**arg2* R4=low-order 32 bits of *arg1**arg2*
GPRs altered	R0, R3, R4
CR Fields altered	None

Table 3–5. The **_divss** Function.

Address	0x00003200
Purpose	Calculate 32-bit integer quotient and 32-bit integer remainder of *arg1*/*arg2*, where *arg1* and *arg2* are signed ints — for division by zero and overflow, the quotient and remainder are undefined and may vary by architecture
Input	R3=*arg1*–dividend (signed int) R4=*arg2*–divisor (signed int)
Output	R3=quotient of *arg1*/*arg2* (signed int) R4=remainder of *arg1*/*arg2* (signed int)
GPRs altered	R0, R3, R4
CR Fields altered	CR0, CR1

Table 3–6. The **_divus** Function.

Address	0x00003280
Purpose	Calculate 32-bit integer quotient and 32-bit integer remainder of *arg1*/*arg2* where *arg1* and *arg2* are unsigned ints — for division by zero, the quotient and remainder are undefined and may vary by architecture
Input	R3=*arg1*–dividend (unsigned int) R4=*arg2*–divisor (unsigned int)
Output	R3=quotient of *arg1*/*arg2* (unsigned int) R4=remainder of *arg1*/*arg2* (unsigned int)
GPRs altered	R0, R3, R4
CR Fields altered	CR0, CR1

Table 3−7. The **_quoss** Function.

Address	0x00003300
Purpose	Calculate the 32-bit integer quotient of *arg1*/*arg2* where *arg1* and *arg2* are signed ints — for division by zero, the quotient is undefined and may vary by architecture
Input	R3=*arg1*−dividend (signed int) R4=*arg2*−divisor (signed int)
Output	R3=quotient of *arg1*/*arg2* (signed int)
GPRs altered	R3, R4
CR Fields altered	None

Table 3−8. The **_quous** Function.

Address	0x00003380
Purpose	Calculate the 32-bit integer quotient of *arg1*/*arg2* where *arg1* and *arg2* are unsigned ints — for division by zero, the quotient is undefined and may vary by architecture
Input	R3=*arg1*−dividend (unsigned int) R4=*arg2*−divisor (unsigned int)
Output	R3=quotient of *arg1*/*arg2* (unsigned int)
GPRs altered	R0, R3, R4
CR Fields altered	CR0, CR1

3.4 Character Representations

Character representations are as follows:

- Any character value between 0 and 127 (inclusive) must correspond to the seven-bit ASCII code (*ISO/IEC 646*).
- Support for multibyte characters is as defined in the ISO standard *ISO/IEC 9899:1990 (section 5.2.1.2)*.

3.5 Storage Operands and Alignment

The PowerOpen Environment (POE) uses the Big-Endian byte ordering; that is, the lowest addressed byte of a multi-byte data item is the most significant. Bytes in storage are numbered consecutively, starting with zero. (In this document, left and right refer to most- and least-significant bit, respectively.) Each number is the address in storage of the corresponding byte. An address is representable as a 32-bit unsigned integer.

Bits are ordered from left to right, with bit 0 being the left-most and most-significant bit.

An area in storage comprises zero or more contiguous bytes. The address of an area is the address of its first byte (lowest address). A storage operand is a byte, halfword, word, doubleword or quadword. A storage operand is an area, but an area might not correspond to a storage operand.

An area whose address is an integral multiple of one is said to be byte-aligned, so all areas and storage operands are byte-aligned. An area whose address is a multiple of two is halfword-aligned; an area whose address is a multiple of four is word-aligned; an area whose address is a multiple of eight is doubleword-aligned; an area whose address is a multiple of 16 is quadword-aligned.

An area starts at its lowest numbered byte and ends immediately after its highest numbered byte.

3.5.1 Storage Operands

Storage operands for single-register Storage Access instructions must have the characteristics shown in Table 3−9. (Although not required as storage operands, quadwords are shown because quadword alignment is desirable for certain storage operands.)

Table 3−9. Storage Operand Characteristics.

Operand	Length	Address bits 28:31, if aligned
Byte	8 bits	*bbbb*
Halfword	2 bytes	*bbb*0
Word	4 bytes	*bb*00
Doubleword	8 bytes	*b*000
Quadword	16 bytes	0000

Note: A *b* in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.

3.5.2 Scalar Types

Even though the PowerPC architecture supports the accessing of data from any alignment, for performance reasons, it is recommended that all scalar data types be aligned at a boundary which is a multiple of their length.

The optimal performance mapping for scalars is shown in Table 3−10. Specific handling of actual alignment is done by the language conventions, but for inter-language and intra-language processing to work with different language processors, follow the rules in this table.

The format of and values represented by float must be those of the PowerPC architecture's 32-bit floating-point single format in IEEE-conformant mode. The format and values represented by double and eight-byte long double must be those of PowerPC architecture's 64-bit floating-point double format in IEEE-conformant mode.

Table 3−10. Alignment for Scalar Types.

Type	C Data Type	Size in Bytes	Recommended Byte Alignment	Storage Operand Type
Characters	unsigned char char	1	Byte	Unsigned byte
	signed char	1	Byte	Signed byte
Integrals	signed short, short, short int	2	Halfword	Signed halfword
	unsigned short, unsigned short int	2	Halfword	Unsigned halfword
	int, long, long int signed int, signed long, enum	4	Word	Signed word
	unsigned int, unsigned long int	4	Word	Unsigned word
	long long, long long int signed long long int	8	Doubleword	Signed double word
	unsigned long long unsigned long long int	8	Doubleword	Unsigned double word
Pointers	any type *, any type (*)()	4	Word	Unsigned word
Floating-point	float	4	Word	Single-precision
	double eight-byte long double	8	Doubleword	Double-precision
	16-byte long double	16	Quadword	Extended-precision

Note: For further information on 16-byte long double and support for 128-bit floating-point, refer to the page 3−18. See page 3−19 for 64−bit integer support.

3.5.3 Aggregates and Unions

The following definitions apply to aggregates:

array Contains an ordered group of adjacent data objects. Each object is called an element. All elements within an array have the same size and data type.

structure Contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have different data types and size. Each data object in a structure is called a member.

union An object that holds any one of a set of named members. The members of the named set can be of any type. The storage allocated for a union is equal to the storage required for the largest member of that union, plus any padding required for alignment.

Table 3−11 shows the mandatory alignment for the scalar members of unions and structures. These rules must be followed in order to guarantee consistent structure mapping among all ABI-conformant language processors.

Table 3−11. Data Alignment Required for Scalar Members of Unions and Structures.

Type	Data Type	Required Byte Alignment
Characters	unsigned char char	Byte
	signed char	Byte
Integrals	signed short, short, short int	Halfword
	unsigned short, unsigned short int	Halfword
	int, long, long int signed int, signed long, enum	Word
	unsigned int, unsigned long int	Word
	long long, long long int signed long long	Word
	unsigned long long unsigned long long int	Word
Pointers	any type *, any type (*)()	Word
Floating-point	float	Word
	double eight-byte long double	Word
	16-byte float	Word

The following aggregate alignment rules apply:

- The alignment of an array is the same as the alignment of its elements' type. For example, an array of elements of type int requires word alignment, and an array of type char requires byte alignment.
- The alignment of the beginning of a structure or a union is the maximum alignment of any of its members. Each member (field) within the structure or union must be placed at its proper alignment as defined in the above table, which may require implicit internal padding, depending on the previous member.
- Structure size must be an integral multiple of its alignment, which may require tail-padding depending on the last member. Language processors must pad up to but not more than the alignment of the strictest member. Structures and unions can be grouped in arrays. Each array element (structure or union) must begin and end at the proper alignment.
- It is acceptable for language processors to start a non-nested aggregate on an alignment greater than what the above table shows; for example, a structure containing a double may start at a doubleword boundary. This may lead to better performance when accessing such data items. However, the internal mapping rules stated above must be preserved.

Figure 3−1 illustrates data definitions, each followed by its layout in memory.

The examples that follow illustrate the internal mapping rules and the required alignment for the beginning of the structure.

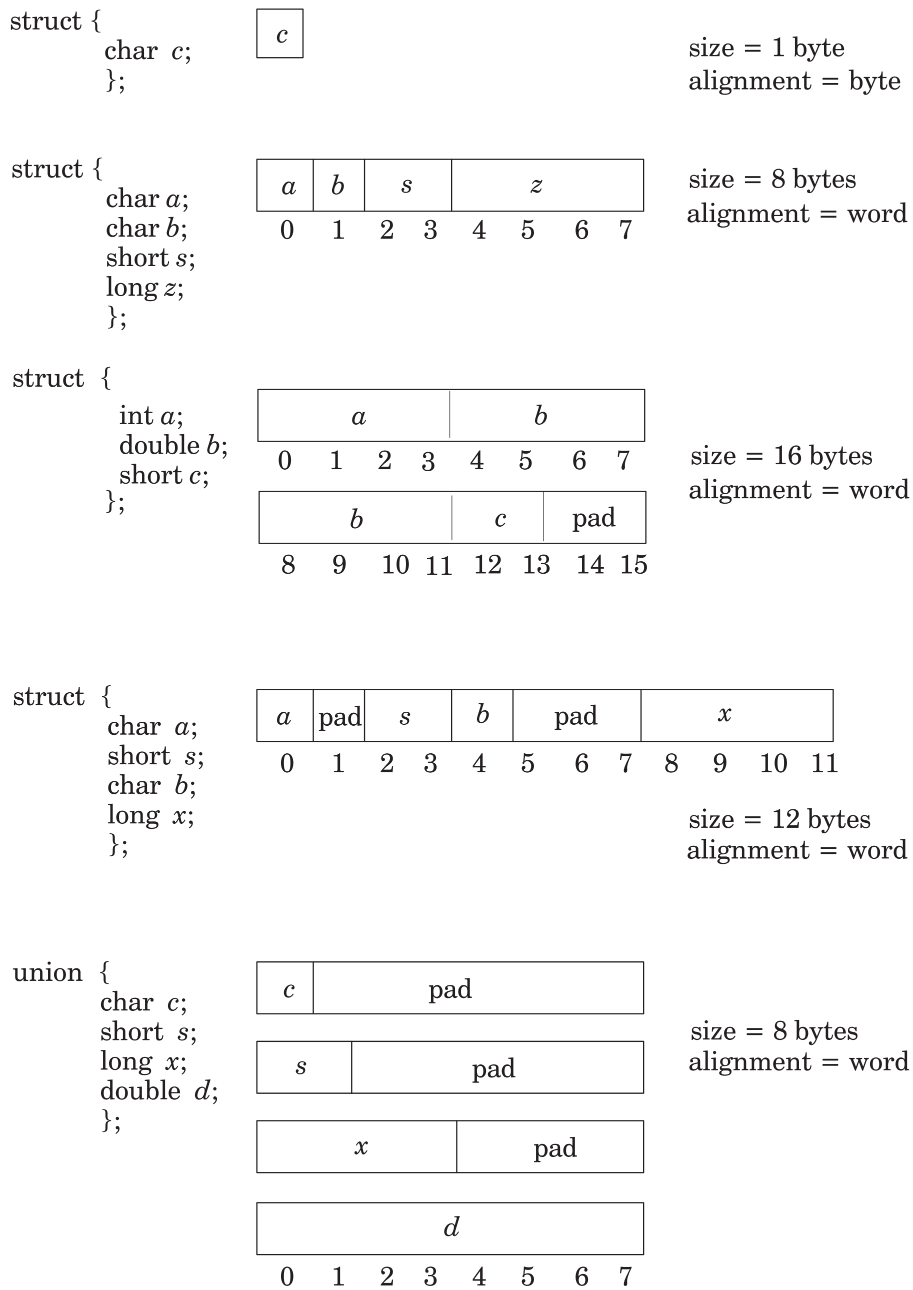

Figure 3–1. Aggregate Data Definitions.

Note: The variables *c*, *s*, *x* and *d* in the last example in the figure share the same address in memory and overlay each other. The alignment of the union is that of the variable with the maximum alignment (which is *d* or *x* in this case).

3.5.4 Bit-Fields

C *struct* and *union* definitions may have data elements known as *bit-fields*. A bit-field definition specifies an integral object with a given number of bits. An unnamed bit-field that is defined as having a length of zero causes the next bit-field, if any, to start on the next word boundary. Fields of other types will start at the required aggregate byte alignment. The maximum length of a bit-field is 32 bits.

Bit-fields must be defined as having type int, signed int or unsigned int. Plain bit-fields having type int must be equivalent to those of type unsigned int.

Bit-fields are allocated sequentially with each bit-field in a word having a higher starting bit number than the previous item.

If a series of bit-fields does not add up to the size of an int, padding will take place up to the next byte. A structure that contains only bit-fields is word-aligned. The amount of padding will be determined by the alignment of the next member in the structure or to the extent that tail padding is required in the structure. Bit-fields cannot cross word boundaries. Figure 3−2 is an example of the internal mapping of a structure containing bit-fields.

```
struct bit_example {
        unsigned a : 1;
        unsigned b : 1;
        int x;
        unsigned c : 4;
        unsigned d : 4;
        unsigned e : 1;
        unsigned : 0;
        unsigned f : 1;
        char y;
        };
```

size = 16 bytes
alignment = word

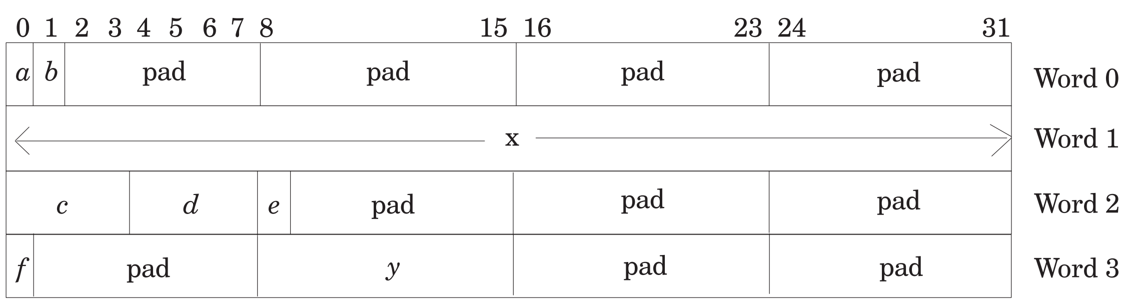

Figure 3−2. Sample Bit-Field Layout.

3.5.5 Support for 128-Bit Floating-Point

The PowerOpen ABI requires the default to be the required C data type long double to be the same as double, using the 64-bit floating point type native to the PowerPC hardware in IEEE-conforming mode. C language processors can provide a compilation mode that treats long double as a 16-byte (128-bit) extended-precision float number. Systems must also provide the optional 128−bit floating−point for application support.

The 128-bit format to be used by long double is actually implemented as a pair of 64-bit native floating-point numbers. The value of the 128-bit quantity is defined as the sum of the two

64-bit components. Thus, each 64-bit half of the 128-bit quantity is itself a double-precision floating-point number with a sign, exponent and significand. Normally, the two numbers do not overlap and all of the bits of the significand add precision beyond the first number. It is possible to arithmetically generate numbers that do overlap, and they may also overlap when they are so small that they denormalize. (The value of the whole number is typically less than the value of the low-order fraction bit in the high-order number.) Please note the following:

- The exponent range is no greater than PowerPC native double-precision.
- As the absolute value of the quantity becomes very small, the additional precision available in the second number decreases, until for denormalized numbers, the precision is the same as for a native 64-bit long double.
- The high-order double word is assumed to have the larger magnitude number.

The 128-bit implementation need not be conformant with IEEE 754. Deviations permitted are as follows:

- The only rounding mode that must be supported is round-to-nearest.
- IEEE special numbers NaN and INF may not be fully supported. Even basic operations such as addition and subtraction may not propagate NaN and INF correctly.
- IEEE status flags in the fpscr for overflow, underflow, *et cetera* may not be correctly set, even by basic operations such as addition and subtraction.

3.5.5.1 C Language Support for 128-Bit Format

If both modes are supported, there must be a mode switch on certain tools to determine which mode is desired. This switch is necessary on the compiler. The default mode in the compiler and libraries must use be the 64-bit mode. A new library, libc128.a, must be provided which contains a single member, shr.o. This member, shr.o, contains only routines from libc.a that are sensitive to the size of long double, programmed for 128 bits.

Compiler suppliers may provide the following:

- A compiler facility (e.g., flag) to cause the compiler to generate code for 128-bit long double format. This facility will cause the compiler to define _LONGDOUBLE128, and header files which depend on the 128-bit long double mode can use it.
- Cause libc128.a to be bound ahead of libc.a at bind time.

libccnv Provides conversion routines to convert to and from 128-bit format and all other supported formats.

libm Provides all math routines required, with appropriate names and error semantics provided by ISO C.

3.5.6 Double Word Integer Support

A long long integer is a 64−bit integer. On 32−bit hardware it is implemented as an ordered pair of 32−bit native integers. The first integer is high order; i.e. (0x1, 0x0) is 1 unit larger than (0x0, 0xFFFFFFFF). A 64−bit signed integer will be represented as a 2’s complement quantity. As seen as a pair of 32−bit integers, the first word (lower memory address or register (for parameters or arguments and results)) will contain the sign bit and 31−bits of numeric information, and the second word (higher address or register) will contain 32−bits of numeric information.

There is also an unsigned long long integer.

3.6 Register Usage and Conventions

The PowerPC 32-bit architecture has 32 *general-purpose registers* (GPRs) and 32 *floating-point registers* (FPRs). Each GPR is 32 bits wide and each FPR is 64 bits wide. In addition, special registers are used for branching, exception handling and other purposes. A full description can be found in *PowerPC Architecture, First Edition*; IBM; May, 1993; customer reorder number 52G7487. Table 3−12 shows how the GPRs are used.

Table 3−12. General-Purpose Register Conventions.

Register	Status	Use
GPR0	Volatile	In function prologs
GPR1	Dedicated	Stack pointer
GPR2	Dedicated	Table of Contents (TOC) pointer
GPR3	Volatile	First word of a function's argument list; first word of a scalar function return
GPR4	Volatile	Second word of a function's argument list; second word of a scalar function return
GPR5	Volatile	Third word of a function's argument list
GPR6	Volatile	Fourth word of a function's argument list
GPR7	Volatile	Fifth word of a function's argument list
GPR8	Volatile	Sixth word of a function's argument list
GPR9	Volatile	Seventh word of a function's argument list
GPR10	Volatile	Eighth word of a function's argument list
GPR11	Volatile	In calls by pointer and as an environment pointer for languages that require it (for example, Pascal)
GPR12	Volatile	For special exception handling required by certain languages and in **glink** code
GPR13:GPR31	Non-volatile	These registers must be preserved across a function call

Note: Volatile registers are scratch registers presumed to be destroyed across a call and are, therefore, not saved by the callee. Volatile registers are also used for specific purposes as shown above. Non-volatile and dedicated registers are required to retain their values across a function call.

Note: The POE stack frame structure assumes that language processors will allocate a contiguous set of GPRs from GPR31 down towards GPR13.

Table 3−13 shows how the floating-point registers are used.

Table 3−13. Floating-Point Register Conventions.

Register	Status	Use
FPR0	Volatile	As a scratch register
FPR1	Volatile	First floating-point parameter; first floating-point scalar return value
FPR2	Volatile	Second floating-point parameter; second floating-point scalar return value
FPR3	Volatile	Third floating-point parameter; third floating-point scalar return value
FPR4	Volatile	Fourth floating-point parameter; fourth floating-point scalar return value
FPR5	Volatile	Fifth floating-point parameter
FPR6	Volatile	Sixth floating-point parameter
FPR7	Volatile	Seventh floating-point parameter
FPR8	Volatile	Eighth floating-point parameter
FPR9	Volatile	Ninth floating-point parameter
FPR10	Volatile	Tenth floating-point parameter
FPR11	Volatile	Eleventh floating-point parameter
FPR12	Volatile	Twelfth floating-point parameter
FPR13	Volatile	Thirteenth floating-point parameter
FPR14:FPR31	Non-volatile	If modified, must be preserved across a call

Note: The preferred method of using FPRs is to use the volatile registers first. Next, the non-volatile registers are used in descending order, starting with FPR31 and going down to FPR14.

Note: Only scalars are returned in multiple registers. The number of registers needed depends on the size of the scalar and its type. For floating-point values, the following must be true:

Note: The POE stack frame structure assumes that language processors will allocate a contiguous set of FPRs from FPR31 down towards FPR14.

- A 128-bit floating point value is returned in FPR1 and FPR2 with the high order 64 bits being returned in FPR1 and the low order 64 bits in FPR2.
- In an eight or 16-byte complex value, the real part is returned in FPR1, and the imaginary part is returned in FPR2.
- In a 32-byte complex value, the real part is returned as a 128-bit floating point value in FPR1 and FPR2, with the high-order 64 bits being returned in FPR1 and the low-order 64 bits being returned in FPR2. ,The imaginary part is returned in FPR3 and FPR4 with the high-order 64 bits being returned in FPR3 and the low-order 64 bits being returned in FPR4.

3.6.1 Special Purpose Registers

Table 3−14 shows the special purpose registers (SPRs) of the *PowerPC Architecture, First Edition*; IBM; May, 1993, and only the SPRs for which there is an ABI register convention.

Table 3−14. Special Purpose Register Conventions.

Register	Status	Use
LR	Volatile	Used as a branch target address or to hold a return address
CTR	Volatile	Used for loop count decrement and branching
XER	Volatile	Fixed point exception register
FPSCR	Volatile	Floating-point status and control register

Register Field	Status
CR0, CR1	Volatile
CR2, CR3, CR4	Non-volatile
CR5, CR6, CR7	Volatile

Note: Functions that alter CR2, CR3 and CR4 must save and restore at least those fields of the CR. The use of the other CR fields does not require saving or restoring.

3.7 Run-Time Process Stack

The PowerOpen ABI stack format convention is designed to enhance the efficiency of the following:

- Prolog and epilog function usage
- Parameter passing
- Shared library support

Figure 3−3 illustrates the PowerOpen ABI run-time stack. It shows the stack after function *sender*() called function *catcher*() but before *catcher*() calls another function. This assumes that *catcher*() will call other functions, so *catcher*() requires another link area (as described in the stack layout). PW*n* refers to the *n*th word of parameters passed.

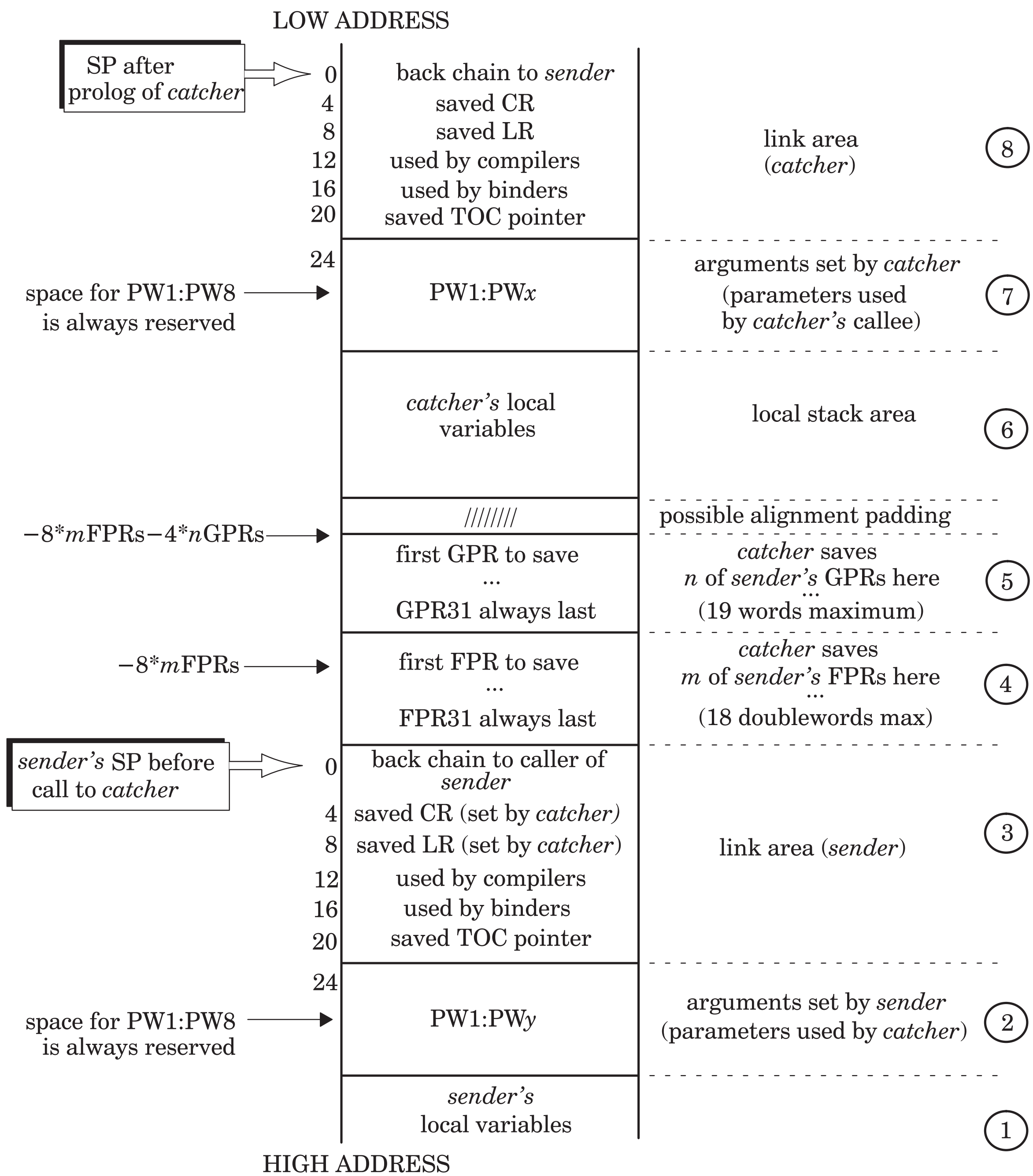

Figure 3−3. Run-Time Stack.

3.7.1 Stack Layout

Only one register is used for addressing the stack, referred to as the *stack pointer* (SP), and GPR1 is the dedicated stack pointer register. The stack grows from numerically higher storage addresses to numerically lower addresses.

Figure 3−3 illustrates what happens when a function shown as *sender*() calls another function shown as *catcher*(), and *catcher*() requires a stack frame of its own. When a function makes no calls and requires no local storage of its own, no stack frame is required, and the SP need not be altered.

The address value in the stack pointer must be quadword-aligned. (It must be a multiple of 16.)

Note: To reduce confusion, data being passed from *sender* (the caller) is referred to as arguments, and the same data being received by *catcher* (the callee) is referred to as parameters. *Sender's* output argument area is the same as *catcher's* input parameter area.

3.7.1.1 Stack Areas

For convenience, the stack layout has been divided into eight areas, numbered 1 to 8, starting from the bottom of the diagram (high address) and extending to the top of the diagram (low address). *Sender's* stack pointer is pointing to the top of area 3 when the call to *catcher* is made, which is also the same SP value that is used by *catcher* on entry to its prolog. Following is a description of the stack areas starting from the bottom of the diagram (area 1) and moving up to the top (area 8).

3.7.1.2 Area 1: Sender's Local Variable Area

This area contains all local variables and temporary space required by *sender*.

3.7.1.3 Area 2: Sender's Output Argument Area

The output argument area is at least eight words in size and must be doubleword-aligned. The first eight words are not used by the caller (*sender*) since their corresponding values are placed directly in the argument registers (GPR3:GPR10). The storage is reserved so that if the callee (*catcher*) takes the address of any of its parameters, then the values passed in GPR3:GPR10 can be stored in their address locations (PW1:PW8, respectively). If *sender* is passing more than eight words of arguments to *catcher*, then it must reserve space for the excess parameters and store them as register images beyond the eight reserved words, starting at offset 56 from *sender's* SP value.

Note: This area may also be used by language processors and is volatile across calls to other functions.

3.7.1.4 Area 3: Sender's Link Area

The link area for *sender* consists of six words and is at offset 0 from *sender's* SP. Certain fields in this area are used by *catcher* as part of its prolog code; those fields are marked in the diagram and explained below.

The first word is the *back chain*, where *sender* saved its caller's SP value prior to modifying the SP. The second word (at offset 4) is where *catcher* can save the CR if it modifies any of the non-volatile CR fields. The third word (offset 8) is where *catcher* can save the LR if *catcher* makes any calls.

The fourth word is reserved for compilers, and the fifth word is used by binder-generated instructions. The last word in the link area (offset 20) is where the TOC area register (see TOC section for description) is saved by the **global linkage** (**glink**) interface routine. This occurs when an out-of-module call is performed; e.g., when a shared library function is called.

3.7.1.5 Area 4: Catcher's Floating-Point Registers Save Area

The callee's (*catcher*) floating-point registers save area is doubleword-aligned. It represents the space needed to save all the non-volatile FPRs used by the callee. The FPRs are saved immediately above the link area (at a lower address) at a negative displacement from *sender's* SP. The size of this area varies from zero to a maximum of 144 bytes depending on the number of FPRs being saved (maximum number is 18 FPRs * 8 bytes each).

3.7.1.6 Area 5: Catcher's General Purpose Registers Save Area

The callee's (*catcher*) general purpose registers save area is at least word-aligned. It represents the space needed by the callee to save all the non-volatile GPRs it uses. The GPRs are saved immediately above the FPR save area (at a lower address) at a negative displacement from *sender's* SP. The size of this area varies from zero to a maximum of 76 bytes depending on the number of GPRs being saved (maximum number is 19 GPRs * 4 bytes each).

Note: A *stackless leaf* function makes no calls and requires no local variable area, but it may use non-volatile GPRs and FPRs.

Note: The save area consists of the FPRs' save area (4) and the GPRs' save area (5), which have a combined maximum size of 220 bytes. The stack floor of the currently executing function is located at 220 bytes less than the value in the SP. The area between the value in the SP and the stack floor is the maximum save area that a stackless leaf function may use without acquiring its own stack. Functions may use it as temporary space, volatile across calls to other functions. Such execution elements as interrupt handlers and binder-inserted code which cannot be seen by compiled code as calls must not use this area.

3.7.1.7 Area 6: Catcher's Local Variable Area

This area contains local variables and temporary space required by *catcher. Catcher* addresses this area using its own SP, which points to the top of area 8, as a base register.

3.7.1.8 Area 7: Catcher's Output Argument Area

The output argument area is at least eight words in size and must be doubleword-aligned. The first eight words are not used by the caller (*catcher*) since their corresponding values are placed directly in the argument registers (GPR3:GPR10). The storage is reserved so that if the *catcher*'s callee takes the address of any of its parameters, then the values passed in GPR3:GPR10 can be stored in their address locations (PW1:PW8, respectively). If *catcher* is passing more than eight words of arguments to its callee, then it must reserve space for the excess parameters and store them as register images beyond the eight reserved words, starting at offset 56 from the *catcher's* SP value.

Note: This area may also be used by compilers and is volatile across calls to other functions.

3.7.1.9 Area 8: Catcher's Link Area

Catcher's link area contains the same fields as those in *sender's* link area (area 3).

3.7.2 Stack Extensions

All language processors must maintain the following system standards related to the stack: The SP must be atomically updated by a single instruction. This ensures that there is no timing window where an interrupt can occur, and where the stack pointer is only partially updated.

Note: The examples of program prologs and epilogs show the most efficient way to update the stack pointer.

3.8 Prologs and Epilogs

This section discusses the use of prologs and epilogs for functions, including setting the registers on function entry and restoring the registers on function exit.

3.8.1 Prologs and Epilogs for Functions

The PowerOpen ABI does not dictate any predetermined code sequences that represent function prologs and epilogs. However, certain operations must be performed under certain conditions. The following tables state the conditions and actions required for prologs and epilogs. Figure 3−4 shows the stack frame layout. A typical function's execution is as follows:

1. Prolog action
2. Body of function
3. Epilog action

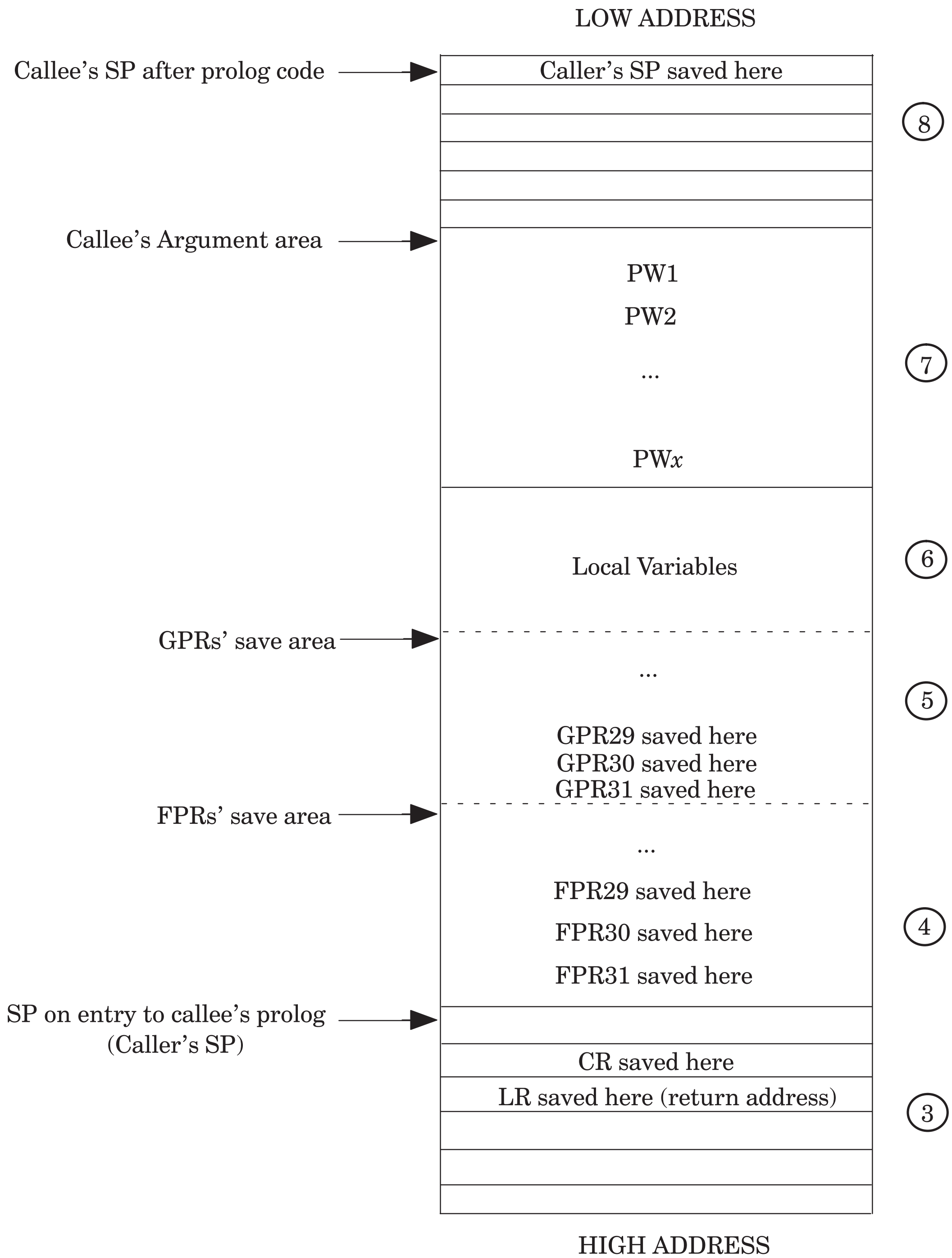

Figure 3–4. Stack Frame Layout.

Table 3–15. Prolog Actions.

If:	Then:
Any non-volatile FPRs (FPR14:FPR31) are used	Save them in the FPR save area (area 4, Figure 3–4)
Any non-volatile GPRs (GPR13:GPR31) are used	Save them in the GPR save area (area 5, Figure 3–4)
LR is used	Save the LR at offset eight from *caller's* SP
Any of the non-volatile condition register (CR) fields are used	Save the CR at offset four from *caller's* SP
A new stack frame is required	Get a stack frame and decrement the stack pointer by the size of the frame padded (if needed) to a multiple of 16 to acquire a new SP and save *caller's* SP at offset 0 from the new SP

Note: A stackless leaf function can save its caller's registers at a negative offset from caller's SP without actually acquiring a stack frame.

Table 3–16. Epilog Actions.

If:	Then:
Any non-volatile FPRs were altered	Restore the FPRs that were used
Any non-volatile GPRs were altered	Restore the GPRs that were used
The LR was altered	Restore LR
The CR was altered	Restore CR
A new stack frame was acquired	Restore the old SP
	Return to caller

As shown in the stack frame layout (Figure 3–4), the GPRs' save area is not at a fixed position from either the caller's SP or the callee's SP. The FPRs' save area starts at a fixed position, directly above the SP (lower address) on entry to callee, but the position of the GPRs' save area depends on the number of FPRs saved. Thus, it is difficult to write a general purpose GPR-saving routine which uses fixed displacements from the SP.

If the routine needs to save both GPRs and FPRs, use GPR12 as the pointer for saving/restoring GPRs. (GPR12 is a volatile register but does not contain input parameters.) This leads to the definition of multiple register save and restore routines, each of which saves or restores m FPRs and n GPRs. This is done by executing a Branch and Link Absolute instruction to specially provided system routines containing multiple entry points (one for each register number) starting from the lowest non-volatile register.

Note: When saving/restoring a small number of non-volatile registers, it may be more efficient to do so in the epilog/prolog code than to call the save/restore routines. If the LR is not saved or restored in the following code segments, the language processor must do the saving and restoring as appropriate.

Language processors must use either a proprietary method to conserve the values of non-volatile registers across a function call, or choose a method from Table 3–17 on page 3–35.

3.8.2 Saving GPRs Only

For a function that saves/restores n GPRs and no FPRs, the saving can be done using individual store/load instructions or by calling system provided routines as shown below:

Note: The number of registers being saved is n, and $<32-n>$ in the assembly examples that follow indicates the first register number to be saved/restored. All registers from $<32-n>$ up to 31, inclusive, are saved/restored.

Note: *frame_size* is the size of the stack frame to be acquired.

```
mflr    r0                      #move LR into GPR0
bla     _savegpr0_<32-n>        #branch & link to save GPRs
stwu    r1,<-frame_size>(r1)    #update the SP and save caller's SP
...
<save CR if necessary>
...
...                             # body of function
...
<reload saved CR if necessary>
...
<reload caller's SP into R1>    #see note below
ba      _restgpr0_<32-n>        #restore GPRs and return
```

Figure 3–5. Saving LR, n GPRs, and no FPRs.

3.8.3 Saving GPRs and FPRs

For a function that saves/restores n GPRs and m FPRs, the saving can be done using individual store/load instructions or by calling system provided routines as shown below:

```
mflr    r0                      #move LR into GPR0
subi    r12,r1,8*m              #compute GPR save pointer
bla     _savegpr1_<32-n>        #branch & link to save GPRs
bla     _savefpr_<32-m>
stwu    r1,<-frame_size>(r1)    #update SP and save caller's SP
...
<save CR if necessary>
<update the SP by frame_size and save caller's SP>
...
...                             #body of function
...
<reload saved CR if necessary>
...
<reload caller's SP into r1>
subi    r12,r1,8*m              #compute GPR restore pointer
bla     _restgpr1_<32-n>        #restore GPRs
ba      _restfpr_<32-m>         #restore FPRs and return
```

Figure 3–6. Saving LR, n GPRs, and m FPRs.

For a function that saves/restores n GPRs and m FPRs, the saving can be done using individual store/load instructions or by calling system provided routines as shown below:

```
mflr    r0                      #move LR into GPR0
subi    r12,r1,8*m              #compute GPR save pointer
bla     _savegpr1_<32-n>        #branch & link to save GPRs
bla     _savefpr_<32-m>
stwu    r1,<-frame_size>(r1)    #update SP and save caller's SP
...
<save CR if necessary>
<update the SP by frame_size and save caller's SP>
...
...                             #body of function
...
<reload saved CR if necessary>
...
<reload caller's SP into r1>
subi    r12,r1,8*m              #compute GPR restore pointer
bla     _restgpr1_<32-n>        #restore GPRs
ba      _restfpr_<32-m>         #restore FPRs and return
mtlr    r0
blr
```

Figure 3–7. Saving GPRs and FPRs When the LR is Saved/Restored by Inline Code.

3.8.4 Saving FPRs Only

For a function that saves/restores *m* FPRs, the saving can be done using individual store/load instructions or by calling system provided routines as shown below:

```
mflr    r0                      #move LR into GPR0
bla     _savefpr_<32-m>
stwu    r1,<-frame_size>(r1)    #update SP and save caller's SP
...
<save CR if necessary>
<update the SP by frame_size and save the caller's SP>
...
...                             # body of function
...
<restore saved CR if necessary>
...
<restore caller's SP into r1>
ba      _restfpr_<32-m>         #restore FPRs and return
```

Figure 3–8. Prolog/Epilog of Function Saving *m* FPRs and no GPRs.

```
_savefpr_14:    stfd    f14,-144(r1)    #save f14
_savefpr_15:    stfd    f15,-136(r1)    #save f15
_savefpr_16:    stfd    f16,-128(r1)    #save f16
_savefpr_17:    stfd    f17,-120(r1)    #save f17
_savefpr_18:    stfd    f18,-112(r1)    #save f18
_savefpr_19:    stfd    f19,-104(r1)    #save f19
_savefpr_20:    stfd    f20,-96(r1)     #save f20
_savefpr_21:    stfd    f21,-88(r1)     #save f21
_savefpr_22:    stfd    f22,-80(r1)     #save f22
_savefpr_23:    stfd    f23,-72(r1)     #save f23
_savefpr_24:    stfd    f24,-64(r1)     #save f24
_savefpr_25:    stfd    f25,-56(r1)     #save f25
_savefpr_26:    stfd    f26,-48(r1)     #save f26
_savefpr_27:    stfd    f27,-40(r1)     #save f27
_savefpr_28:    stfd    f28,-32(r1)     #save f28
_savefpr_29:    stfd    f29,-24(r1)     #save f29
_savefpr_30:    stfd    f30,-16(r1)     #save f30
_savefpr_31:    stfd    f31,-8(r1)      #save f31
                stw     r0,8(r1)        #save LR in caller's frame
                blr                     #return to prolog
```

Figure 3–9. FPR and LR Save.

```
_restfpr_14:    lfd     f14,-144(r1)    #restore f14
_restfpr_15:    lfd     f15,-136(r1)    #restore f15
_restfpr_16:    lfd     f16,-128(r1)    #restore f16
_restfpr_17:    lfd     f17,-120(r1)    #restore f17
_restfpr_18:    lfd     f18,-112(r1)    #restore f18
_restfpr_19:    lfd     f19,-104(r1)    #restore f19
_restfpr_20:    lfd     f20,-96(r1)     #restore f20
_restfpr_21:    lfd     f21,-88(r1)     #restore f21
_restfpr_22:    lfd     f22,-80(r1)     #restore f22
_restfpr_23:    lfd     f23,-72(r1)     #restore f23
_restfpr_24:    lfd     f24,-64(r1)     #restore f24
_restfpr_25:    lfd     f25,-56(r1)     #restore f25
_restfpr_26:    lfd     f26,-48(r1)     #restore f26
_restfpr_27:    lfd     f27,-40(r1)     #restore f27
_restfpr_28:    lfd     f28,-32(r1)     #restore f28
_restfpr_29:    lwz     r0,8(r1)        #get return address from frame
                lfd     r29,-24(r1)     #restore f29
                mtlr    r0              #move return address to LR
                lfd     f30,-16(r1)     #restore f30
                lfd     f31,-8(r1)      #restore f31
                blr                     #return to caller
_restfpr_30:    lfd     f30,-16(r1)
_restfpr_31:    lwz     r0,8(r1)
                lfd     f31,-8(r1)
                mtlr    r0
                blr
```

Figure 3–10. FPR and LR Restore.

```
_savefpr14:     stfd    f14,-144(r1)     #save f14
_savefpr15:     stfd    f15,-136(r1)     #save f15
_savefpr16:     stfd    f16,-128(r1)     #save f16
_savefpr17:     stfd    f17,-120(r1)     #save f17
_savefpr18:     stfd    f18,-112(r1)     #save f18
_savefpr19:     stfd    f19,-104(r1)     #save f19
_savefpr20:     stfd    f20,-96(r1)      #save f20
_savefpr21:     stfd    f21,-88(r1)      #save f21
_savefpr22:     stfd    f22,-80(r1)      #save f22
_savefpr23:     stfd    f23,-72(r1)      #save f23
_savefpr24:     stfd    f24,-64(r1)      #save f24
_savefpr25:     stfd    f25,-56(r1)      #save f25
_savefpr26:     stfd    f26,-48(r1)      #save f26
_savefpr27:     stfd    f27,-40(r1)      #save f27
_savefpr28:     stfd    f28,-32(r1)      #save f28
_savefpr29:     stfd    f29,-24(r1)      #save f29
_savefpr30:     stfd    f30,-16(r1)      #save f30
_savefpr31:     stfd    f31,-8(r1)       #save f31
                blr                      #return to prolog
```

Figure 3−11. FPR Save.

```
_restfpr14:     lfd     f14,-144(r1)     #restore f14
_restfpr15:     lfd     f15,-136(r1)     #restore f15
_restfpr16:     lfd     f16,-128(r1)     #restore f16
_restfpr17:     lfd     f17,-120(r1)     #restore f17
_restfpr18:     lfd     f18,-112(r1)     #restore f18
_restfpr19:     lfd     f19,-104(r1)     #restore f19
_restfpr20:     lfd     f20,-96(r1)      #restore f20
_restfpr21:     lfd     f21,-88(r1)      #restore f21
_restfpr22:     lfd     f22,-80(r1)      #restore f22
_restfpr23:     lfd     f23,-72(r1)      #restore f23
_restfpr24:     lfd     f24,-64(r1)      #restore f24
_restfpr25:     lfd     f25,-56(r1)      #restore f25
_restfpr26:     lfd     f26,-48(r1)      #restore f26
_restfpr27:     lfd     f27,-40(r1)      #restore f27
_restfpr28:     lfd     f28,-32(r1)      #restore f28
_restfpr29:     lfd     r29,-24(r1)      #restore f29
_restfpr30:     lfd     f30,-16(r1)      #restore f30
_restfpr31:     lfd     f31,-8(r1)       #restore f31
                blr                      #return to caller
```

Figure 3−12. FPR Restore.

Table 3−17 is presented to clarify the uses of the preceding tables.

Table 3−17. Use of Save/Restore Routines.

FPR Save	GPR Save	LR Save	Save Routines	Restore Routines
No	Yes	No	Figure 3−15	Figure 3−16
No	Yes	Yes	Figure 3−13	Figure 3−14
Yes	No	No	Figure 3−11	Figure 3−12
Yes	No	Yes	Figure 3−9	Figure 3−10
Yes	Yes	No	Figure 3−15, Figure 3−11	Figure 3−16, Figure 3−11
Yes	Yes	Yes	Figure 3−15, Figure 3−9	Figure 3−16, Figure 3−10

3.8.5 Multiple Entry Point Services

Systems must provide three sets of system services, which may be implemented as multiple entry-point routines or as individual routines. Independent of their implementation, they must adhere to the rules in the following tables.

Table 3−18. GPR and LR Save and Restore Entry Points.

Save		Restore	
Save Entry Point	**Registers Saved**	**Restore Entry Point**	**Registers Restored**
_savegpr0_13	GPR13:GPR31	_restgpr0_13	GPR13:GPR31
_savegpr0_14	GPR14:GPR31	_restgpr0_14	GPR14:GPR31
_savegpr0_15	GPR15:GPR31	_restgpr0_15	GPR15:GPR31
_savegpr0_16	GPR16:GPR31	_restgpr0_16	GPR16:GPR31
_savegpr0_17	GPR17:GPR31	_restgpr0_17	GPR17:GPR31
_savegpr0_18	GPR18:GPR31	_restgpr0_18	GPR18:GPR31
_savegpr0_19	GPR19:GPR31	_restgpr0_19	GPR19:GPR31
_savegpr0_20	GPR20:GPR31	_restgpr0_20	GPR20:GPR31
_savegpr0_21	GPR21:GPR31	_restgpr0_21	GPR21:GPR31
_savegpr0_22	GPR22:GPR31	_restgpr0_22	GPR22:GPR31
_savegpr0_23	GPR23:GPR31	_restgpr0_23	GPR23:GPR31
_savegpr0_24	GPR24:GPR31	_restgpr0_24	GPR24:GPR31
_savegpr0_25	GPR25:GPR31	_restgpr0_25	GPR25:GPR31
_savegpr0_26	GPR26:GPR31	_restgpr0_26	GPR26:GPR31
_savegpr0_27	GPR27:GPR31	_restgpr0_27	GPR27:GPR31
_savegpr0_28	GPR28:GPR31	_restgpr0_28	GPR28:GPR31
_savegpr0_29	GPR29:GPR31	_restgpr0_29	GPR29:GPR31
_savegpr0_30	GPR30:GPR31	_restgpr0_30	GPR30:GPR31
_savegpr0_31	GPR31	_restgpr0_31	GPR31

For Table 3−13, the calling conventions for the _savegpr0_ family of entry points are as follows:

- R1 must point to the starting address of the GPR save area.
- LR must contain the return address.
- R0 must contain the value of LR on function entry.

When calling the _restgpr0 family of entry points, R1 must point to the starting address of the GPR save area.

Actions for the _save_gpr0_ family of entry points are as follows:

- Save the registers as indicated in Table 3−13.
- Save the value in register R0 at offset 8, relative to R1.

In addition, the _restgpr0 family of entry points may use R0 without saving the original value. The routine will end by loading the stack register with the saved caller's LR address, and it will perform a blr instruction using this value.

```
_savegpr0_13:   stw     r13,-76(r1)      #save r13
_savegpr0_14:   stw     r14,-72(r1)      #save r14
_savegpr0_15:   stw     r15,-68(r1)      #save r15
_savegpr0_16:   stw     r16,-64(r1)      #save r16
_savegpr0_17:   stw     r17,-60(r1)      #save r17
_savegpr0_18:   stw     r18,-56(r1)      #save r18
_savegpr0_19:   stw     r19,-52(r1)      #save r19
_savegpr0_20:   stw     r20,-48(r1)      #save r20
_savegpr0_21:   stw     r21,-44(r1)      #save r21
_savegpr0_22:   stw     r22,-40(r1)      #save r22
_savegpr0_23:   stw     r23,-36(r1)      #save r23
_savegpr0_24:   stw     r24,-32(r1)      #save r24
_savegpr0_25:   stw     r25,-28(r1)      #save r25
_savegpr0_26:   stw     r26,-24(r1)      #save r26
_savegpr0_27:   stw     r27,-20(r1)      #save r27
_savegpr0_28:   stw     r28,-16(r1)      #save r28
_savegpr0_29:   stw     r29,-12(r1)      #save r29
_savegpr0_30    stw     r30,-8(r1)       #save r30
_savegpr0_31    stw     r31,-4(r1)       #save r31
                stw     r0 , 8(r1)       #save LR in caller's frame
                blr                      #return to prolog
```

Figure 3−13. GPR and LR Save.

```
_restgpr0_13:   lwz     r13,-76(r1)     #restore r13
_restgpr0_14:   lwz     r14,-72(r1)     #restore r14
_restgpr0_15:   lwz     r15,-68(r1)     #restore r15
_restgpr0_16:   lwz     r16,-64(r1)     #restore r16
_restgpr0_17:   lwz     r17,-60(r1)     #restore r17
_restgpr0_18:   lwz     r18,-56(r1)     #restore r18
_restgpr0_19:   lwz     r19,-52(r1)     #restore r19
_restgpr0_20:   lwz     r20,-48(r1)     #restore r20
_restgpr0_21:   lwz     r21,-44(r1)     #restore r21
_restgpr0_22:   lwz     r22,-40(r1)     #restore r22
_restgpr0_23:   lwz     r23,-36(r1)     #restore r23
_restgpr0_24:   lwz     r24,-32(r1)     #restore r24
_restgpr0_25:   lwz     r25,-28(r1)     #restore r25
_restgpr0_26:   lwz     r26,-24(r1)     #restore r26
_restgpr0_27:   lwz     r27,-20(r1)     #restore r27
_restgpr0_28:   lwz     r28,-16(r1)     #restore r28
_restgpr0_29:   lwz     r0,8(r1)        #get return address from frame
                lwz     r29,-12(r1)     #restore r29
                mtlr    r0              #move return address to LR
                lwz     r30,-8(r1)      #restore r30
                lwz     r31,-4(r1)      #restore r31
                blr                     #return to caller
_restgpr0_30:   lwz     r30,-8(r1)
_restgpr0_31    lwz     r0,8(r1)
                lwz     r31,-4(r1)
                mtlr    r0
                blr     r0
```

Figure 3–14. GPR and LR Restore When FPRs Are Not Saved.

Table 3−19. GPR Save and Restore Entry Points.

Save		Restore	
Save Entry Point	**Registers Saved**	**Restore Entry Point**	**Registers Restored**
_savegpr1_13	GPR13:GPR31	_restgpr1_13	GPR13:GPR31
_savegpr1_14	GPR14:GPR31	_restgpr1_14	GPR14:GPR31
_savegpr1_15	GPR15:GPR31	_restgpr1_15	GPR15:GPR31
_savegpr1_16	GPR16:GPR31	_restgpr1_16	GPR16:GPR31
_savegpr1_17	GPR17:GPR31	_restgpr1_17	GPR17:GPR31
_savegpr1_18	GPR18:GPR31	_restgpr1_18	GPR18:GPR31
_savegpr1_19	GPR19:GPR31	_restgpr1_19	GPR19:GPR31
_savegpr1_20	GPR20:GPR31	_restgpr1_20	GPR20:GPR31
_savegpr1_21	GPR21:GPR31	_restgpr1_21	GPR21:GPR31
_savegpr1_22	GPR22:GPR31	_restgpr1_22	GPR22:GPR31
_savegpr1_23	GPR23:GPR31	_restgpr1_23	GPR23:GPR31
_savegpr1_24	GPR24:GPR31	_restgpr1_24	GPR24:GPR31
_savegpr1_25	GPR25:GPR31	_restgpr1_25	GPR25:GPR31
_savegpr1_26	GPR26:GPR31	_restgpr1_26	GPR26:GPR31
_savegpr1_27	GPR27:GPR31	_restgpr1_27	GPR27:GPR31
_savegpr1_28	GPR28:GPR31	_restgpr1_28	GPR28:GPR31
_savegpr1_29	GPR29:GPR31	_restgpr1_29	GPR29:GPR31
_savegpr1_30	GPR30:GPR31	_restgpr1_30	GPR30:GPR31
_savegpr1_31	GPR31	_restgpr1_31	GPR31

For Table 3−19, the calling conventions for the _savegpr1_ family of entry points are as follows:

- R12 must point to the starting address of the GPR save area.
- LR must contain the callee’s link return address.

The _restgpr1 family of entry points must return via a blr instruction.

Calling conventions for the _restgpr1_ family of entry points are as follows:

- R12 must point to the starting address of the GPR save area.
- LR must contain the return address (which may be to the function’s epilog or caller.)

```
_savegpr1_13:   stw     r13,-76(r12)     #save r13
_savegpr1_14:   stw     r14,-72(r12)     #save r14
_savegpr1_15:   stw     r15,-68(r12)     #save r15
_savegpr1_16:   stw     r16,-64(r12)     #save r16
_savegpr1_17:   stw     r17,-60(r12)     #save r17
_savegpr1_18:   stw     r18,-56(r12)     #save r18
_savegpr1_19:   stw     r19,-52(r12)     #save r19
_savegpr1_20:   stw     r20,-48(r12)     #save r20
_savegpr1_21:   stw     r21,-44(r12)     #save r21
_savegpr1_22:   stw     r22,-40(r12)     #save r22
_savegpr1_23:   stw     r23,-36(r12)     #save r23
_savegpr1_24:   stw     r24,-32(r12)     #save r24
_savegpr1_25:   stw     r25,-28(r12)     #save r25
_savegpr1_26:   stw     r26,-24(r12)     #save r26
_savegpr1_27:   stw     r27,-20(r12)     #save r27
_savegpr1_28:   stw     r28,-16(r12)     #save r28
_savegpr1_29:   stw     r29,-12(r12)     #save r29
_savegpr1_30:   stw     r30,-8(r12)      #save r30
_savegpr1_31:   stw     r31,-4(r12)      #save r31
                blr                      #return to prolog
```

Figure 3−15. GPR Save.

```
_restgpr1_13:   lwz     r13,-76(r12)     #restore r13
_restgpr1_14:   lwz     r14,-72(r12)     #restore r14
_restgpr1_15:   lwz     r15,-68(r12)     #restore r15
_restgpr1_16:   lwz     r16,-64(r12)     #restore r16
_restgpr1_17:   lwz     r17,-60(r12)     #restore r17
_restgpr1_18:   lwz     r18,-56(r12)     #restore r18
_restgpr1_19:   lwz     r19,-52(r12)     #restore r19
_restgpr1_20:   lwz     r20,-48(r12)     #restore r20
_restgpr1_21:   lwz     r21,-44(r12)     #restore r21
_restgpr1_22:   lwz     r22,-40(r12)     #restore r22
_restgpr1_23:   lwz     r23,-36(r12)     #restore r23
_restgpr1_24:   lwz     r24,-32(r12)     #restore r24
_restgpr1_25:   lwz     r25,-28(r12)     #restore r25
_restgpr1_26:   lwz     r26,-24(r12)     #restore r26
_restgpr1_27:   lwz     r27,-20(r12)     #restore r27
_restgpr1_28:   lwz     r28,-16(r12)     #restore r28
_restgpr1_29:   lwz     r29,-12(r12)     #restore r29
_restgpr1_30:   lwz     r30,-8(r12)      #restore r30
_restgpr1_31:   lwz     r31,-4(r12)      #restore r31
                blr                      #return to epilog or caller
```

Figure 3−16. GPR Restore.

Table 3–20. FPR and LR Save and Restore Entry Points.

Save		Restore	
Save Entry Point	**Registers Saved**	**Restore Entry Point**	**Registers Restored**
_savefpr_13	Save FPR13:FPR31	_restfpr_13	Restore FPR13:FPR31
_savefpr_14	Save FPR14:FPR31	_restfpr_14	Restore FPR14:FPR31
_savefpr_15	Save FPR15:FPR31	_restfpr_15	Restore FPR15:FPR31
_savefpr_16	Save FPR16:FPR31	_restfpr_16	Restore FPR16:FPR31
_savefpr_17	Save FPR17:FPR31	_restfpr_17	Restore FPR17:FPR31
_savefpr_18	Save FPR18:FPR31	_restfpr_18	Restore FPR18:FPR31
_savefpr_19	Save FPR19:FPR31	_restfpr_19	Restore FPR19:FPR31
_savefpr_20	Save FPR20:FPR31	_restfpr_20	Restore FPR20:FPR31
_savefpr_21	Save FPR21:FPR31	_restfpr_21	Restore FPR21:FPR31
_savefpr_22	Save FPR22:FPR31	_restfpr_22	Restore FPR22:FPR31
_savefpr_23	Save FPR23:FPR31	_restfpr_23	Restore FPR23:FPR31
_savefpr_24	Save FPR24:FPR31	_restfpr_24	Restore FPR24:FPR31
_savefpr_25	Save FPR25:FPR31	_restfpr_25	Restore FPR25:FPR31
_savefpr_26	Save FPR26:FPR31	_restfpr_26	Restore FPR26:FPR31
_savefpr_27	Save FPR27:FPR31	_restfpr_27	Restore FPR27:FPR31
_savefpr_28	Save FPR28:FPR31	_restfpr_28	Restore FPR28:FPR31
_savefpr_29	Save FPR29:FPR31	_restfpr_29	Restore FPR29:FPR31
_savefpr_30	Save FPR30:FPR31	_restfpr_30	Restore FPR30:FPR31
_savefpr_31	Save FPR31	_restfpr_31	Restore FPR31

For Table 3–20, the calling conventions for the _savefpr_ family of entry points are as follows:

- R1 must point to the starting address of the FPR save area.
- LR must contain the return address.
- R0 must contain the value of LR on function entry.

Calling conventions for the _restfpr_ family of entry points are as follows:

Table 3−21. FPR Save and Restore Entry Points.

Save		**Restore**	
Save Entry Point	**Registers Saved**	**Restore Entry Point**	**Registers Restored**
_savefpr13	Save FPR13:FPR31	_restfpr13	Restore FPR13:FPR31
_savefpr14	Save FPR14:FPR31	_restfpr14	Restore FPR14:FPR31
_savefpr15	Save FPR15:FPR31	_restfpr15	Restore FPR15:FPR31
_savefpr16	Save FPR16:FPR31	_restfpr16	Restore FPR16:FPR31
_savefpr17	Save FPR17:FPR31	_restfpr17	Restore FPR17:FPR31
_savefpr18	Save FPR18:FPR31	_restfpr18	Restore FPR18:FPR31
_savefpr19	Save FPR19:FPR31	_restfpr19	Restore FPR19:FPR31
_savefpr20	Save FPR20:FPR31	_restfpr20	Restore FPR20:FPR31
_savefpr21	Save FPR21:FPR31	_restfpr21	Restore FPR21:FPR31
_savefpr22	Save FPR22:FPR31	_restfpr22	Restore FPR22:FPR31
_savefpr23	Save FPR23:FPR31	_restfpr23	Restore FPR23:FPR31
_savefpr24	Save FPR24:FPR31	_restfpr24	Restore FPR24:FPR31
_savefpr25	Save FPR25:FPR31	_restfpr25	Restore FPR25:FPR31
_savefpr26	Save FPR26:FPR31	_restfpr26	Restore FPR26:FPR31
_savefpr27	Save FPR27:FPR31	_restfpr27	Restore FPR27:FPR31
_savefpr28	Save FPR28:FPR31	_restfpr28	Restore FPR28:FPR31
_savefpr29	Save FPR29:FPR31	_restfpr29	Restore FPR29:FPR31
_savefpr30	Save FPR30:FPR31	_restfpr30	Restore FPR30:FPR31
_savefpr31	Save FPR31	_restfpr31	Restore FPR31

Actions for the save_fpr0 family of entry points are as follows:

- Save the registers as indicated in Table 3−13.
- Save the value in register R0 at offset 8, relative to R1.

3.8.6 Updating the Stack Pointer

The PowerPC Store Word With Update instruction has a signed 16-bit displacement field that can only represent a maximum positive value of 32K−1. A stack frame size 32K or greater requires two instructions to update the SP, and the update must be done atomically.

The restoring of caller's SP can be done by either adding the *frame_size* to the current SP whenever *frame_size* is known, or by reloading it from offset 0 from the current SP. The first approach is generally more efficient, but it cannot be used by functions that use **alloca** to dynamically allocate stack space.

The two assembly code examples that follow illustrate how to update the SP in a prolog.

Figure 3−17 shows a sample sequence of instructions that computes a new SP and saves the old SP for stack frames greater than or equal to 32K.

```
addis   R12,r0,(<-frame_size> >16)&0XFFFF
                                    #Set r12 to left half of frame size

ori     R12,R12,(<-frame_size>)&oXFFFF
                                    #Add right halfword of frame size
stwux   R1,R1,R12                   #Save old SP and compute new SP
```

Figure 3−17. Computing a New SP and Saving the Old SP.

Figure 3−18 shows a sample sequence of instructions that computes a new SP and saves the old SP for stack frames less than 32K.

```
addis   R12,r0,(<-frame_size> >16)&0XFFFF
                                    #Add left halfword of frame size
stwu    r1,<-frame_size>(r1)        #update SP and save caller's SP
ori     R12,R12,(<-frame_size>)&0XFFFF
                                    #Add right halfword of frame size

stwux   R1,0(R1,R12)                #Save old SP and compute new SP
```

Figure 3−18. Computing a New SP and Saving the Old SP.

3.8.7 alloca Function

alloca allows user programs to dynamically acquire storage from the stack. This can be done explicitly using the **alloca** function, as follows:

```
char *alloca (int size)
p = alloca(size);
```

The number of bytes specified by `size` (rounded up to a multiple of 16 bytes) is allocated on the stack of the function that invokes the **alloca** function. This temporary space is automatically freed when the calling function completes its execution. The **alloca** function returns a pointer to the requested storage suitably aligned to a multiple of 16 bytes.

The following rules must be followed in the implementation of the **alloca** function:

- The **alloca** function must atomically update (decrease) the current SP by the size of the required space rounded up to a multiple of 16.
- The link area must begin at offset zero from the current SP, and caller's SP (back chain) must always be available at offset zero from the current SP.
- Compilers must choose a non-dedicated GPR to address the local variables area and the input parameter area. If a traceback table is provided by the language processor, this chosen register must be recorded in the **alloca_reg** field in the traceback table so that debuggers can address the local variables and argument area based on it.
- Compilers must set the **uses_alloca** bit-field in the traceback table if a traceback table is provided. The actual amount of storage allocated in the argument area by invoking the **alloca(***size***)** function is represented by the following computation:

  floor((*size* + 15)/16)*16

Note: The **setjmp** and **longjmp** functions must save and restore the old SP in order to work correctly for programs that invoke the **alloca** function.

Figure 3−19 illustrates stack growth with **alloca**. The space reserved for catcher's link area and argument area, prior to the invocation of **alloca**, is used as part of the space required to satisfy the **alloca** request, and a new argument area and link area are allocated above (lower address) the allocated space. The only data that needs to be copied from the old argument area and old link area is *sender's* SP (old SP), which must always be saved at offset 0 in the new link area.

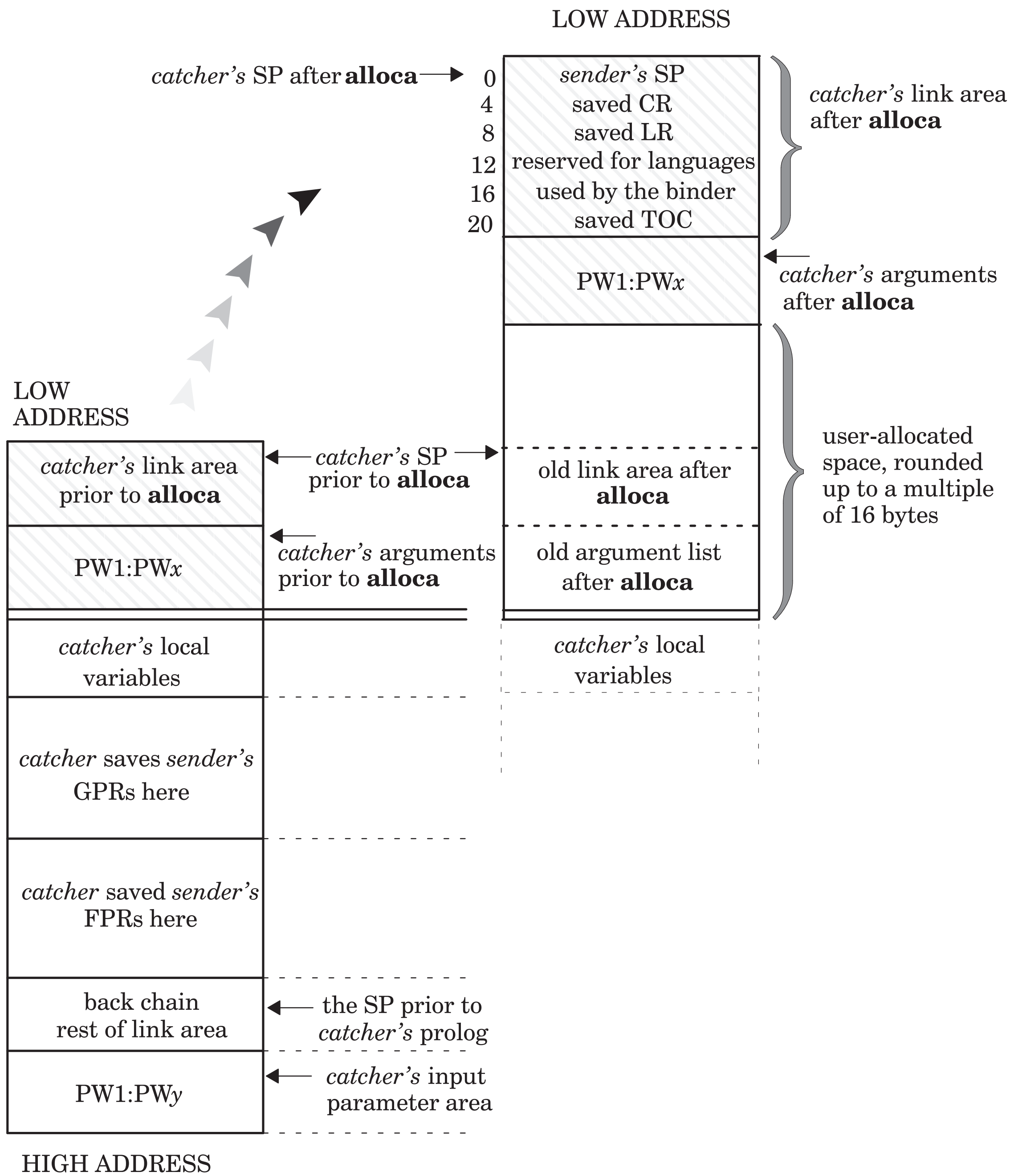

Figure 3–19. Run-Time Stack for a Function that Calls **alloca**.

3.9 Function Calling Sequence

This section describes the specifications regarding function calling, parameter and argument passing, and related internal conventions.

3.9.1 Argument Passing Rules

The parameter passing and related conventions take advantage of the available GPRs and FPRs by passing arguments to a function in two designated register lists. The designated lists are

- GPR3:GPR10 for non-floating-point values
- FPR1:FPR13 for floating-point values

The parameter area, located at a fixed offset from the stack frame pointer, is reserved in each stack frame for use as an argument list. (A minimum of eight words is always reserved.) The size of this area must be sufficient to contain the longest argument list being passed by the function that owns the stack frame. Although not all arguments for a particular call are located in storage, consider them to be forming a list in this area, each argument occupying one or more words.

If more arguments are passed than one or both of the lists can hold, the remaining words are stored in the argument area. The values passed on the stack are identical to those that would have been placed in registers, thus they are *register images*. The argument area space for the additional words must immediately follow the space reserved for PW1:PW8 (the first eight words of parameters passed), as shown in Figure 3−3.

For storage considerations, arguments can be classified as follows:

Floating-point values
Occupy one,two, or four consecutive words, for single−precision, double-precision, or 128-bit floating-point, respectively.

Non-floating-point values
Stored exactly as they exist in GPRs. Each non-floating-point value occupies one word.

Other scalar values
Other scalars, such as FORTRAN complex numbers, are mapped by the number of words required by their size.

Structure values
Stored in successive words beginning at word alignment and satisfying all the internal structure alignment requirements discussed earlier.

Other aggregate values
Some languages pass other aggregate values as value by reference; that is, the compiler passes storage addresses of the values and arranges for a copy of the values to be made in the called function.

The rules for argument and parameter passing are as follows:

- The argument area (PW1:PW*x*) must be large enough to hold the longest argument list required by a function.
- Each argument is mapped to as many words of the argument area as are required to hold its value. The mapping rules include:
  - Other scalars, such as FORTRAN complex numbers, are mapped by the number of words required by their size.
  - Non-floating-point values are mapped to a single word.

- Single-precision floating values are mapped to a single word.
- Double-precision floating values are mapped to two consecutive words.
- Structures passed by value are mapped to as many consecutive words of the argument area as the value uses in memory.

- If the callee has a known prototype, arguments are converted to the type of the corresponding parameter before being mapped into the argument area. For example, if a double-precision value is used as an argument to a single-precision parameter, the value is converted to single-precision and mapped to a single word in the argument area.
- The first eight words mapped to the argument area are never stored in the argument area by the calling function. Instead, these words are passed in registers as described below.
- Argument values mapped to the ninth and subsequent words must be stored in the argument area whether or not they are passed in the FPRs.
- GPRs are used to pass some values. The first eight words mapped to the argument area correspond to the registers GPR3:GPR10. That is, word PWn corresponds to GPR($2+n$). (If the arguments are mapped to fewer than eight words of the argument area, registers corresponding to these unused words are not used.)
  - If there is no known function prototype for a callee, or the known prototype contains an ellipsis, the first eight words mapped to the argument area are passed in their corresponding registers.
  - If the callee has a prototype with no ellipsis, the non-floating-point values in PW1:PW8 are passed in their corresponding registers. GPRs corresponding to floating-point argument words are not used — the floating-point words are placed in floating-point registers.
- FPRs are used consecutively to pass up to 13 floating-point values. Each floating-point register holds at most a single value, whether the argument is single- or double-precision.
  - If there is no known prototype for a callee, or the known prototype contains an ellipsis, up to 13 floating-point values are passed in FPR1:FPR13. In this case, floating-point values mapped to the first eight words of the argument area are also passed in GPRs.
  - If there's a known prototype for the callee, and it doesn't contain an ellipsis, up to 13 floating-point values are passed in FPR1:FPR13. In this case, floating-point values mapped to the first eight words of the argument area are *not* also passed in GPRs.
  - If the callee has a declaration with an ellipsis, up to 13 floating-point values corresponding to floating-point arguments are received in FPR1:FPR13.
- If the callee takes the address of any of its parameters, then GPR3:GPR10 must be stored by the callee into PW1:PW8, except when any of these GPRs correspond to floating-point parameters, in which case the FPR(s) themselves are stored rather than the GPR(s). This is so because the caller may have had a function prototype for the callee and, therefore, put floating-point parameters in floating-point registers only.

Note: If the compilation unit for the caller contains a function prototype for the callee, but the callee has a mismatching definition, and if the callee takes the address of any of its parameters, the wrong values may be stored in the first eight words of the argument area.

Note: If any part of a multi-word scalar is mapped onto GPR10, any remaining words from that scalar will be passed on the stack.

3.9.2 Application of Parameter and Argument Rules

This section provides examples showing how to apply parameter and argument passing rules for ABI-conformant systems. In Figure 3−20, the caller (*pitcher*) contains a prototype definition for the callee (*catcher*), which defines its parameters similarly.

Figure 3−21 is an example of argument passing when the caller has no prototype for the callee.

Caller

```
pitcher()
{
double catcher(int,double,int);

d=catcher(5,1.0,2);
}
```

Callee

```
double catcher(int i, double x, int y)
{
}
```

Arguments are passed as follows:	Parameters are used as follows:
R3=5	Use contents of R3 as value of i
R4:unused	
R5:unused	
F1=1.0	Passed as double; use contents of F1 as value of x
R6=2	Use contents of R6 as value of y

Figure 3−20. Caller With Prototype Definition for Callee.

The language processor performs the following processing tasks:

- Passes the first argument (value 5) in the first of the GPRs designated for passing arguments (R3)
- Passes each floating-point value in the corresponding FPR; that is, the first floating-point argument in F1, the second in F2, and so on
- Skips R4 and R5 (which correspond to the two words being passed in F1)
- Passes the third argument (value 2), which is the fourth word, in R6

Caller	**Callee**
`pitcher()`	`double catcher(int i,double x,int y)`
`{`	`{`
`double catcher();`	`}`
`d=catcher(5,1.0,2);`	
`}`	
`Arguments are passed as follows:`	`Parameters are used as follows:`
`R3=5`	`Use contents of R3 as value of i`
`R4:1st word of 1.0`	
`R5:2nd word of 1.0`	
`F1=1.0`	`Passed as double; use contents of F1 as value of x`
`R6=2`	`Use contents of R6 as value of y`

Figure 3−21. Caller Without Prototype Definition for Callee.

In the above example, *pitcher* passes the floating-point value (1.0) in both the GPRs (R4 and R5) and in the FPR (F1). Because it lacks a function prototype for *catcher*, it assumes that *catcher* may look for the second parameter in either GPRs or FPRs.

Note: Because *catcher* has the correct description of its parameters, it uses F1 as having the value of its second parameter, *x*.

Figure 3−22 is an example of a C code function invocation elaborating use of the argument area.

```
catcher (L1, L2, L3, d1, f1, c1, d2, s1, s2_L4_f2);
```

Figure 3−22. Function Invocation Using Argument Area.

Table 3−22 indicates the data types of arguments for this example.

Table 3−22. C Data Types.

An argument beginning with:	**Denotes:**
L	long (word)
d	double-precision float
f	single-precision float
s	short (halfword)
c	char (one character)
s*n*_L*n*_f*n*	A structure with three elements of the types short, long, and single-precision float, respectively

Table 3–23 shows the application of argument passing rules, assuming there is no known prototype for *catcher* in Figure 3–22, using data type of Table 3–22.

Table 3–23. Application of Argument Passing Rules.

Passed In	Offset	Argument	Position in a Word
GPR3	0	L1	
GPR4	4	L2	
GPR5	8	L3	
FPR1, also in GPR6, GPR7 as no prototype	12 16	d1	
FPR2, also in GPR8 and GPR9 as it is promoted to double if no prototype	20 24	f1	
GPR10	28	≡≡≡≡≡≡≡≡| c1	Right-most byte
FPR3 and the argument area	32 36	d2	
Argument area	40	≡≡≡≡≡ | s1	Right-most byte
Argument area	44	s2 |≡≡≡≡	Left-most byte
Argument area	48	L4	
Argument area	52	f2	

3.9.3 Function Descriptor

A function descriptor is a three-word data structure that contains the following values:

- The first word contains the address of the entry point of the function.
- The second word contains the address of the TOC directory of the module that contains the function.
- The third word contains the environment pointer for languages such as Pascal and PL/I.

Figure 3–23 depicts the descriptor contents:

Address of code
Address of TOC
Environment pointer

Figure 3–23. Function Descriptor.

3.9.4 Function Calling By Name

To call a function (for example, *catcher*) without passing any arguments, language processors must produce the following instructions in order to support multiple TOCs and shared libraries.

```
bl                .catcher          #Branch to catcher
ori               R0,R0,0           #OR immediate (No-Op)
```

Figure 3–24. Instructions Supporting Multiple TOCs and Shared Libraries.

When the binder sees this combination of instructions during binding, it performs one of the following:

- If *catcher* is bound into the same module as its caller, then the sequence shown above remains unchanged.
- If *catcher* is *imported* (accessed and linked from a different module, such as with shared libraries), then the binder changes the branch to *catcher* to a branch to a special routine (the **glink** routine). Each out-of-module function has its own uniquely-named **glink** routine that is inserted by the binder (referred to in Figure 3–25). In such cases, the **ori** is replaced with a **load** instruction, as follows:

```
bl                <glink_catcher> #Branch to glink for catcher
lwz               R2,20(R1)       #Restore TOC pointer
```

Figure 3–25. Load Instruction Replacing **ori**.

Note: If language processors implicitly or explicitly know that a function being called will be statically bound with its caller, then they need not have an **ori** (No-Op) instruction following the branch instruction.

Note: For backward compatibility, binders must accept the instructions cror R15, R15, 15 and cror R31, R31, 31 as equivalent to the **ori** R0, R0, 0 instruction.

Figure 3–26 illustrates the required assembly code for the **glink** routine for *catcher.*

```
#       R0                             #Used as scratch
#       R1                             #Stack pointer (SP)
#       R2                             #TOC register (R2)
#       R12                            #Reserved for glink scratch use
#       CTR                            #Count register used to branch to
                                       #glink of caller

#       Word at offset 20 from the SP is used by glink

lwz     R12,<descriptor offset>(R2) #Load address of catcher's
                                       #descriptor
stw     R2,20(R1)                      #Save caller's R2 in link area
lwz     R0,0(R12)                      #Get address of catcher
lwz     R2,4(R12)                      #Get TOC address of catcher
mtctr   R0                             #Move catcher's address
bctr                                   #to CTR
                                       #Branch to catcher
```

Figure 3–26. Assembly Code for **glink** Routine.

Note: The **glink** routine does not save and restore the environment pointer (R11). Therefore, language processors for languages that support an environment pointer (such as Pascal) must generate code to save and restore the pointer.

Note: There will only be one **glink** routine for each out-of-module (shared) function.

3.9.5 Function Calling By Pointer

Function pointer is the name given to a data type whose values provide access to functions. Variables of this type occur in several programming languages such as C and FORTRAN, and they are commonly used in procedure call contexts where the function to be called cannot be determined at compile time. Function pointer values are generally obtained by an explicit or implicit operation applied to a function constant. An example in C is the reference to a function identifier without following parentheses; for example, *function*, not *function*(). Support for the use of function pointers in contexts such as the following is provided:

- The target of a call statement
- An actual argument of a call statement
- The source of an assignment or return statement
- An element of certain arithmetic expressions (notably, the comparison and algebraic difference of two pointers)

Note: Function pointers do not denote activation instances of functions; calling a function with a pointer always results in a new invocation. A function pointer must be a word that is the address of a function descriptor.

A system routine called **.ptrgl** (pointer global linkage) is provided by the system environment (such as the C library) to facilitate calls by function pointer. Figure 3–27 shows the required assembly code for the **._ptrgl** routine. This routine is provided by the system environment to support calls by the function pointer. The **._ptrgl** routine expects GPR11 to contain the descriptor address for the function being called by the pointer.

```
lwz     R0,0(R11)                #Load address of code of called function
stw     R2,20(R1)                #Save current TOC address
mtctr   R0                       #Move branch address from GPR0 to CTR

lwz     R2,4(R11)                #Set new TOC address
lwz     R11,8(R11)               #Set environment pointer
bcctr   20,0                     #Go to callee
```

Figure 3−27. Assembly Code for **.ptrgl** Routine.

Figure 3−28 shows compiler-generated code. It illustrates a call that uses a function pointer in which the address of the descriptor is passed in GPR11:

```
bl      ._ptrgl                  #Branch to system pointer linkage
lwz     R2,20(R1)                #Restore my R2
```

Figure 3−28. Address of Descriptor Being Passed in GPR11.

Note: The link register is not altered by the **._ptrgl** routine, so the called function returns directly to its caller and not to the **._ptrgl** routine.

Note: It is possible for language processors to generate the **._ptrgl** code inline. Inline code may lead to better performance but will increase program size.

WARNING: The ._ptrgl routine must always be statically bound with functions that use it.

3.9.6 Function Return Values

The linkage convention designates the registers for values returned by a function. How these values are returned depends on their type, in accordance with the following rules:

Note: Structures of any length and character strings longer than four bytes are returned in a storage buffer allocated by the caller. The address of this buffer is passed as a hidden first argument in GPR3, which causes the first explicit argument word to be passed in GPR4. This hidden argument is treated as a formal argument and corresponds to the first word of the argument area.

- Values of the types int, long, short, pointer, and char (length is less than or equal to four bytes), as well as bit values of lengths less than or equal to 32 bits, must be returned right-justified in GPR3 with signed values sign-extended and unsigned values zero-extended, as necessary.
- Non-float scalar values of eight bytes must be returned in GPR3:GPR4.
- Eight-byte complex values can be passed in FPR1:FPR2.
- Float scalar values of 16 or 32 bytes must be returned in FPR1:FPR2 and FPR1:FPR4, respectively.

3.10 Symbol Names

The following are the rules for symbol naming:

- The external symbol names from the system libraries must have no prefix or suffix added to the base name.

Note: For more information on external symbol compatibility, refer to Appendix B, page B-3.

- A function's name in an XCOFF file must match the name of its descriptor. This is the name used by the binder for name resolution. System libraries require external names to be unique through the first 255 characters.
- C language names are case-sensitive. To continue the language tradition as well as allow for inter-language calling, FORTRAN and Pascal names have traditionally been folded to one case type (upper or lower).
  - FORTRAN and Pascal symbol names fold to lowercase by default. A compiler option must be provided to allow mixed case without folding.
  - The binder looks for exact match only but may provide a mechanism for renaming and aliasing.

3.11 Sharing Data Between Languages

A procedure must be able to access data that is defined in another language. The data can be external to the routines involved or can be passed as arguments. All language processors must follow consistent implementation methods for storing and passing data that is of a type supported by multiple languages. Many of these conventions are established in this document in the parameter passing sections that follow. This section elaborates on the conventions that must be followed by all the language processors for sharing data types.

3.11.1 Common Data Types

Different languages offer different data types and different semantics associated with each data type. However, the inter-language convention offers a common set of data types for which multiple languages have compatible support. For inter-language calls to work, conventions are established for mapping the common data types to storage and for passing them as arguments.

In addition to the common data types, each language can have its own unique data types. This document refers to these as private data types. New common data types will be introduced as necessary. Passing of private data types or use of private parameter passing mode is not defined by the ABI.

3.11.2 Private Data Types

Many language-specific data types that do not map to any of the common data types identified in the following tables do not participate in inter-language calls (e.g., packed decimal for COBOL). Examples of language specific data types follow:

- C bit-fields
- Pascal sets
- Dynamic types that need run-time descriptors
- COBOL decimal

The common data types can be expanded whenever the possibility arises, such as the introduction of a new language or the extension of an existing language.

3.11.3 Common Argument Passing Modes

Two common modes are defined for inter-language argument passing support.

value mode In this mode the value of the parameter is passed.

reference mode
In this mode the address of the argument is passed.

To make inter-language calling possible, all languages must support the above common modes of argument passing for the common data types. Common modes do not have to be the default.

Each language can have various private modes of argument passing. Private argument passing modes can exist for private data types as well as common data types.

3.11.4 Data Type Conventions

Conventions for storage mapping, alignment, and argument passing modes for each common data type are described in the following tables.

Note: Not all language processors can support all parameter passing modes for the common data types. For methodology, see page 3−45, Argument Passing Rules.

Table 3–24. Conventions for One-Byte Signed Integers.

Attribute	Convention
Size	One byte
Function return	Value returned in low-order byte of GPR3 (right-most byte), sign-extended to 32 bits
Parameter passing modes	Value and reference
Value mode	Pass value in low-order byte of register or register image in the argument area sign-extended to 32 bits
Reference mode	Pass address of the byte in the register or register image in the argument area

Table 3–25. Conventions for One-Byte Unsigned Integers.

Attribute	Convention
Size	One byte
Function return	Value returned in low-order byte of GPR3, zero-extended to 32 bits
Parameter passing modes	Value and reference
Value mode	Pass value in low-order byte of GPR3 or register image in the argument area zero-extended to 32 bits
Reference mode	Pass address of the byte in the register or register image in the argument area

Table 3–26. Conventions for Two-Byte Signed Integers.

Attribute	Convention
Size	Two bytes
Function return	Value returned in low-order two bytes of GPR3, sign-extended to 32 bits
Parameter passing modes	Value and reference
Value mode	Pass value in low-order two bytes of register or register image in the argument area with sign-extended to 32 bits
Reference mode	Pass address of the high-order byte in the GPR3 or register image in the argument area

Table 3−27. Conventions for Two-Byte Unsigned Integers.

Attribute	Convention
Size	Two bytes
Function return	Value returned in low-order two bytes of GPR3, zero-extended to 32 bits
Parameter passing modes	Value and reference
Value mode	Pass value in low-order two bytes of register or register image in the argument area with zero-extended to 32 bits
Reference mode	Pass address of the high-order byte in the register or register image in the argument area

Table 3−28. Conventions for Four-Byte Signed Integers.

Attribute	Convention
Size	Four bytes
Function return	Value returned in four bytes of GPR3
Parameter passing modes	Value and reference
Value mode	Pass value in the register or register image in the argument area
Reference mode	Pass address of the high-order byte in the register or register image in the argument area

Table 3−29. Conventions for Four-Byte Unsigned Integers.

Attribute	Convention
Size	Four bytes
Function return	Value returned in four bytes of GPR3
Parameter passing modes	Value and reference
Value mode	Pass value in the register or register image in the argument area
Reference mode	Pass address of the high-order byte in the register or register image in the argument area

Table 3−30. Conventions for Pointers.

Attribute	Convention
Size	Four bytes
Function return	Value returned in four bytes of GPR3
Parameter passing modes	Value and reference
Value mode	Pass value in the register or register image in the argument area
Reference mode	Pass address of the high-order byte in the register or register image in the argument area

Table 3−31. Conventions for Four-Byte Real Numbers.

Attribute	Convention
Size	Four bytes
Function return	Value returned in FPR1
Parameter passing modes	Value and reference
Value mode	Pass value in the FP register or storage image (four bytes) in the argument area
Reference mode	Pass address of the high-order byte in the register or register image in the argument area

Table 3−32. Conventions for Eight-Byte Real Numbers.

Attribute	Convention
Size	Eight bytes
Function return	Value returned in FPR1
Parameter passing modes	Value and reference
Value mode	Pass value in the FP register or storage image (eight bytes) in the argument area
Reference mode	Pass address of the high-order byte in the register or register image in the argument area

Table 3−33 lists the conventions for fixed-length arrays containing scalar elements of any of the previously described data types.

Table 3−33. Conventions for Fixed-Length Arrays.

Attribute	Convention
Size	Length of element times the number of elements required
Function return	Not applicable to inter-language calling
Parameter passing modes	Reference required
Reference mode	Pass address of the high-order byte in the register or register image in the argument area. Do not pass the length.

Note: Multi-dimensional arrays do not have a common ordering of elements. Some languages store them in row-major and others in column-major order. Programmers must be aware of this when making inter-language calls.

Table 3−34 lists the conventions for aligned structures containing any of the previously described data types.

Table 3–34. Conventions for Aligned Structures.

Attribute	Convention
Size	Specified by the semantics of the source language of the program
Function return	Value is placed at the address passed by the caller as a hidden parameter preceding the actual parameters
Parameter passing modes	Reference required
Reference mode	Pass address of the first element in a parameter register if one is available. Otherwise, pass the address in the argument area. Do not pass the length.

Table 3–35 lists the conventions for arrays of aligned structures containing any of the previously described data types.

Table 3–35. Conventions for Arrays of Structures.

Attribute	Convention
Storage amounts	Storage requirement of the structure plus alignment needed at the end of the structure to make the next structure begin at the right alignment) times the number of elements (structures) in the array
Function return	Not applicable to inter-language calling
Parameter passing modes	Reference required
Reference mode	Pass address of the first byte of the first field in a parameter register if one is available. Otherwise, pass the address in the argument area. Do not pass the length.

Note: For more information on inter-language calling, refer to Appendix A, page A-3.

3.11.5 Exception Interface and Stack Unwinding

Language processors must support generation of the mandatory part of the traceback tables, and system libraries must contain at least the mandatory part of the traceback tables. The mandatory and optional parts follow Table 3–36.

Note: Compilers should provide a facility to turn off traceback generation to save space when the diagnostic information is not needed.

3.11.5.1 Traceback Generation by Language Processors

To provide full traceback capabilities, language processors must generate a traceback table in the .text section at the end of the code for every function. Debuggers and exception handlers can locate the traceback tables by scanning forward from the instruction address at the point of interruption. The beginning of the traceback tables is marked by a word of zeroes, which is an illegal instruction. If read-only constants are compiled into the .text section of a file, they must follow the traceback table. A word of zeroes as read-only data shall not be the first word following the code. A traceback table is word-aligned and contains information used by debuggers and exception handlers.

This section describes only the traceback table's content.

3.11.5.2 Mandatory Traceback Table Fields

Following are the mandatory fields of the traceback table.

version Eight-bit field. This defines the type code for traceback table format. The only value currently defined is 0.

lang Eight-bit field. This one-byte code identifies the source language for the compiler that generated the code to which this traceback table applies. The defined values for this field are in Table 3−36.

Table 3−36. Source Language Identifiers.

Language	Compiler Source	Code (in hexadecimal)	Identifier
C	TB_C	0x0000	0
FORTRAN	TB_FORTRAN	0x0001	1
Pascal	TB_PASCAL	0x0002	2
Ada	TB_ADA	0x0003	3
PL/I	TB_PL/1	0x0004	4
Basic	TB_BASIC	0x0005	5
LISP	TB_LISP	0x0006	6
COBOL	TB_COBOL	0x0007	7
Modula2	TB_MODULA2	0x0008	8
C++	TB_CPLUSPLUS	0x0009	9
RPG	TB_RPG	0x000A	10
PL8, PLIX	TB_PL8	0x000B	11
Assembly	TB_ASM	0x000C	12
Reserved		0x000D−0x00FA	
Reserved for IBM		0x00FB−0x00FF	

globallink One-bit field. This field is set to 1 if this routine is one of the routines used to support the linkage convention (**glink**, **._ptrgl**). Because the register usage and stack format within these routines are different from those defined elsewhere, a traceback routine must be able to identify this context.

is_eprol One-bit field. This field is set to 1 if this routine is an out-of-line prolog or epilog routine. Because such a routine will be establishing the proper stack frame format, a traceback routine must know that the context is not stable.

has_tboff One-bit field. This field is set to 1 if the offset of the traceback table from the start of the procedure is stored in the **tb_offset** field.

int_proc One-bit field. This field is set to 1 if this function is an internal stackless leaf procedure that does not have a separate stack frame. The details of stack format in such a case are dependent on the language that generated the code.

has_ctl One-bit field. This field must be set to 1 if **ctl_info** is provided; otherwise, this field must be 0.

tocless One-bit field. This field is set to 1 if this procedure does not have a TOC. For example, a stackless leaf assembly language routine with no references to external objects.

fp_present One-bit field. This field is set to 1 if floating-point processor instructions are present within this procedure.

log_abort One-bit field. Reserved.

int_handl One-bit field. Reserved.

name_present
One-bit field. This field is set to 1 if the name for this procedure is present following the traceback table, as determined by the **name_len** and **name** fields.

uses_alloca One-bit field. This field is set to 1 if the procedure uses the **alloca** function (only C) to allocate dynamic storage in the stack. These procedures continue to use GPR1 as the stack pointer. However, to address their local variables, these procedures require a different register to hold the stack pointer value before the **alloca** function is called. The register number can be chosen by the compiler and must be indicated by setting the value of the **alloca_reg** field.

cl_dis_inv Three-bit field. Reserved.

saves_cr One-bit field. If this routine saves the CR, this field is set to 1. If so, the CR saved word in the link area must be used to restore the proper condition setting before returning from this procedure.

saves_lr One-bit field. This field is set to 1 if this routine stores the LR in the argument area. If the LR is not stored in the link area, then it is not modified by this routine and contains the return address. For stackless leaf procedures, this bit is set to 0.

stores_bc One-bit field. If this function saves the back chain (the SP of its caller) in its link area, this field is set to 1.

fixup A one-bit field that indicates that the binder replaced the original instruction by a branch instruction to a special fixup instruction sequence.

fp_saved Six-bit field. This field represents the number of non-volatile FPRs that are saved by this procedure. Because the last register saved is always FPR31; e.g., a value of 2 in the **fp_saved** field indicates that FPR30 and FPR31 are saved.

spare3 Two-bit field. Reserved.

gpr_saved Six-bit field. This six-bit field represents the number of non-volatile GPRs saved by this procedure. As with the **fp_saved** field, the last register saved is always GPR31.

fixedparms Eight-bit field. This field records the number of fixed-point parameters.

floatparms Seven-bit field. This field records the number of floating-point parameters.

Note: If either **fixedparms** or **floatparms** is set to 1, **parminfo** exists (see below).

parmsonstk One-bit field. This field is set if all of the parameters are placed in the argument area.

Note: A debugger can use the **fixedparm**, **floatparm**, and **parmsonstk** fields to support displaying the parameters passed to a procedure. They specify the number of parameters passed in GPRs and the number of parameters passed in FPRs; they also determine whether the parameters are stored in the argument area. The parameters are stored in the argument area if the number of parameters is variable, or if the address of one of the parameters is taken, or if optimization is such that the parameters are always stored in the argument area. If either the **fixedparms** or **floatparms** field is set to 1, then the next field, **parminfo**, is used by a debugger to determine the relative order and the types of the parameters for presentation.

3.11.5.3 Optional Traceback Table Fields

The following are the optional fields of the traceback table:

parminfo Unsigned int. This field is only present if either **fixedparns** or **floatparms** is set to 1. It is used by a debugger to determine which registers were used to pass parameters to the routine and to determine the layout of the parameter area in the argument area. This word will be interpreted from left to right, as follows:

- If the current bit is set to 0, the corresponding parameter is a fixed point passed in a GPR or a single word in the parameter area.
- If the current bit is set to 1, then the corresponding parameter is floating-point and the following bit determines whether the parameter is single-precision (the following bit is 0) or double-precision (the following bit is 1).

Note: Since this field is only 32 bits long, there is a limit to how many parameters can be described, this limit is in the range of 16 to 32 depending on the type of parameters. Note that it takes two bits for each floating-point parameter and one bit for each non-float parameter.

tb_offset unsigned int. This word is present only if the **has_tboff** field is set. It represents the code length of the procedure.

hand_mask int. Reserved.

ctl_info int. If the **has_ctl** field is set to 1, this field gives the number of controlled automatic anchor blocks defined for this procedure. If an error handler is unwinding the stack to restart some earlier process, then the controlled automatic storage must be released. This field is only present if the **has_ctl** bit is set. If 0, there is no controlled automatic anchor.

ctl_info_disp int[*]. This is the displacement that indicates the location of information. This field exists only if the **has_ctl** bit-field is set to 1.

name_len short. This halfword is the length of the procedure name that immediately follows this field. This field is only present if name_present is set to 1. The name is in seven-bit ASCII, and is not delimited by a null character.

name char[*]. This is the actual name of the procedure. The **name** field only exists if the **name_len** field is greater than zero.

alloca_reg char. This field is present only if the **uses_alloca** bit is a 1. It contains the register number that is used as the base for automatic variable accesses (instead of GPR1).

3.11.5.4 Line Number Table

Language processors may support line number generation. If generated, the line number table contains one entry for each source line that has produced code. A description of the format, length, and contents of each entry in the table, as well as the table's location in the XCOFF file are described in Chapter 4, Object File Format.

3.12 Type-Checking Extensions

3.12.1 Compiler Encoding for Types

Execution time errors are difficult to find, and many are caused by mismatching function interfaces or conflicting data definitions. To help uncover such errors early, a scheme is defined by which language processors encode information about all external symbols (data and programs). This information is then checked at bind or load time for consistency. The encoding can be turned on or off at compile time, optionally bypassed by the binder, or both. Type-checking support for encoding is required for systems and language processors. This an optional requirement for application.

Compiler-generated information for checking can be divided into two parts: information related to the compilation unit and information related to each external symbol. (See Chapter 4, Object File Format.)

Checking the information for each external symbol encompasses data attributes and external functions such as

- Data type
- Size
- Scalar
- Array
- Structure
- Parameters
- Parameter types
- Parameter passing mode (by value or reference)
- Return values

The information can be represented in a canonical form that is encoded by a common hashing algorithm that can be used by all language processors. (The algorithm is provided in 3.12.6.) The interface to this hashing routine is described in later sections. The resultant hash code for each symbol is then saved by the language processor in the .typchk section of the XCOFF file in Chapter 4, Object File Format, which is then used in the binding or loading process to ensure consistency.

In a system that has a variety of programming languages and inter-language calling support, it is important to design a mechanism that enforces the maximum checking required by the semantics of each language. However, the checking mechanism should be more flexible to accommodate languages that have lesser enforcement rules.

The checking mechanism is a four-part hash encoding that provides the following:

- Checking flexibility ranging from the strictest required for intra-language bind and load time type-checking to weaker forms that provide protection for applications written in more than one language.
- A unique hash encoding termed *universal hash* that has the value 0x20202020. The universal hash can be used as an escape mechanism by language processors for type-encoding of functions and data where full attributes are not known.

For more information on type-encoding, refer to Chapter 4, Object File Format, page 4−27.

Table 3−37 depicts the arrangement of the four type-encoding fields.

Table 3−37. Type-Encoding Fields.

Hash Length Field	Language ID Field	General Hash Field	Language Hash Field
Two-byte field Value of total length of next three fields (value of 10)	Two-byte field Contains language ID	Four-byte field Contains general hash	Four-byte field Contains language hash

Mismatched hash type information discovered by the binder and/or loader can be used for diagnostics. (A description of how and where the hash fits in the XCOFF can be found in Chapter 4, Object File Format.)

3.12.2 Canonical Representation Rules for Data

The following terms are defined for canonical representation of data:

General string
: The language-independent canonical representation of the type attribute of an external symbol (data or function).

Language string
: A language-specific canonical representation of the type attribute of an external symbol (data or function).

Universal string
: A string of four blank ASCII characters (0x20202020).

The following sections provide rules regarding the canonical representation of data types that participate in the general hash. The list of rules does not include all data types and features that may be unique to one language or set of languages.

3.12.2.1 Scalar Representation

Table 3−38 lists rules for the canonical representation of integers and characters.

Table 3–38. Canonical Representation of Integers and Characters.

Rules for Canonical Representation	Data Type Example	General and Language String Representation
Singed integer types begin with the letter *i*, followed by a number representing the size of the integer in bytes.	int *abc* short	i4 i2
Unsigned integer data types are represented by the letter *u* preceding the type, followed by the number of bytes.	unsigned int *abc* unsigned short	ui4 ui2
Floating-point types begin with the letter *r*, followed by a number indicating its size in bytes.	float double *xyz* long double (16 bytes)	r4 r8 r16
Character types begin with the letter *c*, followed by a number indicating the number of characters.	unsigned char char	c1 c1
Signed characters must be represented as a one-byte integer. (Characters are assumed to be unsigned.)	signed char	i1
Bit types begin with the letter *b*, followed by the number of bits. A bit type defaults to unsigned.	BIT(5) bit-field:b5	b5
Signed bit-fields are preceded by the letters *sgn* and followed by the number of bits.	signed *def*:b5	sgnb5

The following canonical representation define the data alignment. This list of rules does not include all aggregate data types that may be unique to any one language or set of languages.

3.12.2.2 Bit-Field Representation

Figure 3–29 illustrates the canonical representation of properly aligned bit-fields that are multiples of eight bits in length.

```
bit(8), logical*1, unsigned :8, unsigned char    ===> c1
bit(16), logical*2, unsigned :16, unsigned char  ===> ui2
bit(32), logical*4, unsigned :32, unsigned char  ===> ui4

integer*1, (signed) char, int :8                 ===> i1
integer*2, short, int :16                        ===> i2
integer*4, int, long, int :32                    ===> i4
```

Figure 3–29. Properly-Aligned Bit-Fields in Multiples of Eight.

3.12.2.3 Array Representation

For canonical representation, language processors must represent arrays as follows in the order listed:

1. Left bracket.
2. Number of elements of information for each dimension, separated by commas.
3. Right bracket, followed by the canonical representation of an array element's type information. If a dimension has no bound information, then the number of elements is represented by an asterisk.

4. If any dimension information is unknown, the language string is the universal string. If the array element's language string is the universal string, then the array's language string is the universal string.
5. In the general string, array dimensions are always represented by an asterisk (*).

Figure 3−30 illustrates the canonical representations for an array.

```
Array Declaration       General String  C Language String
------------------      --------------- -----------------
float XYZ[5][10];       [*,*]r4         [5,10]r4
int XYZ[][10];          [*,*]i4         Universal string
int XYZ[];              [*]i4           Universal string
```

Figure 3−30. Canonical Representation for an Array.

3.12.2.4 Pointer Representation

The language string for a pointer is always a universal string.

For general string representation, pointers are considered addresses and are indicated by the following elements in the order listed (the language string for a pointer is always the universal string):

1. The letter *a*.
2. A number indicating the size of the pointer in bytes (four).
3. The canonical string for the item pointed to.
4. A pointer to a structure is represented as a4str; the structure itself is not represented.
5. A pointer to a union is represented as a4un.
6. A pointer to a function is represented as a4fcn.

Figure 3−31 is an example of canonical representation for arrays of pointers.

```
Declaration             General String          C Language String
------------            ---------------         ------------------
int (*x)[5];            a4[*]i4                 Universal string
int (**x)[5];           a4a4[*]i4               Universal string
int *(*x)[5];           a4[*]a4i4               Universal string
```

Figure 3−31. Canonical Representation for an Array of Pointers.

3.12.2.5 Structure Representation

For canonical representation, structures are indicated by the following elements in the order listed:

1. The letters *str*.
2. A left brace.
3. The types of each member, separated by commas.
4. A right brace.

If padding (filler) is implicitly inserted or explicitly stated (e.g., COBOL filler or asterisk in PL/I) between two elements of the structure or at the end of the structure to fulfill an alignment requirement, then the filler is represented as indicated in Table 3−39.

Table 3−39. Canonical Representation of Filler in Structures.

Filler	Canonical Representation
Less than one byte (1−7 bits)	b1 to b7
One or more bytes on a byte boundary	A comma-separated list of c1s, sufficient to pad the necessary space
More than one byte, not a multiple of eight (non-byte boundaries)	b*n* (*n* less than or equal to 7) pads any byte that is not completely filled. If a field must begin on a word boundary, and there are unused bits and bytes preceding it, the unused space will be filled with b*n* to the first byte boundary and c1s thereafter, up to the next required boundary.

The example in Figure 3−32 illustrates the canonical representation for a structure declaration.

```
struct {                        General string
        int e;
        short f;                str{i4,i2,c1,c1,r4,str{r8,c1,b5,b3,c1,c1}}
        float g;
        struct  {
                double h;
                char y;
                int z: 5;
                } x;
        } a;
```

Figure 3−32. Canonical Representation for a Structure Declaration.

Word alignment is required for the variable *g*, which is of type float. Therefore, two pad characters are inserted after the variable *f*. Also, the presence of a filler at the end of structure *x* (the inner structure) is required by the C language semantics. (C semantics require that the ending alignment of a structure must be the same as the alignment of the first element of the structure.)

3.12.2.6 Union Representation

Union alignment is indicated by the letters *alg*, followed by a number representing the byte alignment.

Table 3−40. Canonical Representation of Alignment.

Alignment	Canonical Representation
Byte	alg1
Halfword	alg2
Word	alg4
Doubleword	alg8

For canonical representation, unions are indicated by the following elements in the order listed:

1. The letters *ar* (for area).
2. Left brace.
3. Canonical representation of maximum alignment, size of union in bytes.
4. Right brace.

Details about the union are not hashed, thereby allowing different union member descriptions to coexist.

The example in Figure 3−33 illustrates the canonical representation for a union.

```
union {                       General string
        int b;
        short c;                ar{alg4,8}
        double d;
        } xyz;
```

Figure 3−33. Canonical Representation for a Union.

3.12.2.7 Miscellaneous Representations

Table 3−41 lists rules for the canonical representation of languages.

Table 3−41. Canonical Representations from Various Languages.

Data Type	General and Language String Representation
Varying length character string (excluding null-terminated strings) PL/I declaration DCL STR CHAR(100) VARYING;	A structure consists of an unsigned length field followed by the string str{ui*n*,c*s*} (*s*=size of string in bytes; *n*=number of bytes used to represent length of string)
FORTRAN blank COMMON	Only universal string
FORTRAN named COMMON	If the common block is named, use the letters s*n* in the language hash. The general string is the universal string. The language string is s*n*. (*n* is the size of block in bytes)
Double-byte characters	ui2
FORTRAN label	lbl

3.12.3 Canonical Representation Rules for Functions

Historically, few languages have provided a syntax by which programmers could describe interfaces to their programs and to other programs that they invoke. An example of such syntax is the function prototype in C. It is therefore important to have an encoding mechanism that works with existing code, yet is flexible enough to support the extra checking for languages that currently support it or will support it in the future.

3.12.3.1 Type Encoding for Programs

The type encoding for programs consists of the same four-part hash described in the previous sections for data. The four fields are detailed again below.

3.12.3.1.1 General String

The following list of rules are applicable to the general hash field:

- The general string is used to describe the function's parameters, if any, and what it returns (if anything).
- When incomplete or no parameter information is available, then the general string must be the universal string.
- If there is no information about a function's return, then the general string must be the universal string.
- C language semantics concerning the passing of arrays, and limiting reference parameters to values representing addresses, requires that all reference parameters be treated as addresses to a given type. Consider the following examples:
  - An array of integers passed by reference is represented as a4i4.
  - A structure passed by reference is represented as a4str.
  - For consistency, an explicit pointer is passed as an address (double**xyz* is represented as a4r8). This has the effect of treating all parameters as value parameters, because a reference parameter is a value that represents an address of a variable of a given type.
- For FORTRAN, the compiler infers the arguments' types and return information from the procedure invocation. If there is more than one call to the same procedure, the compiler should check that the parameters and return attributes are consistent. If inconsistencies occur, the compiler should generate a universal hash for the called procedure.

3.12.3.1.2 Language String

Following are rules for the language string:

- The language string for functions must never be the universal string, even when the general string is the universal string.
- The language string should consist of the function return information only. If no information is provided for the return, the function is assumed to return a four-byte integer.

Following are rules for FORTRAN language programs:

- The language string must allow a FORTRAN caller that ignores the return from a function to match with the language hash of the callee.
- The language string should detect when the calling function expects a return value while the callee function is a function (no return).

3.12.3.2 Canonical Representation Rules for Functions

Following are general rules for the canonical representation for programs — the list is not inclusive:

- A function is defined by the word fcn, left parenthesis, parameters' descriptions, right parenthesis; and if there is a returned value, the word ret, left parenthesis, type of return, and right parenthesis.
- Hidden or implied parameters must be included in the same order in which they are being passed.
- Unspecified parameters (ellipsis) are indicated by the word ell, and must always be at the end of the argument list.

Table 3–42 describes some examples of canonical representation for programs.

Table 3–42. Examples of Canonical Representations.

Declaration	**Language ID**	**General String**	**Language String**
int f (int);	C	fcn(i4)ret(i4)	ret(i4)
int f (void);	C	fcn()ret(i4)	ret(i4)
void c (void);	C	fcn()	ret()
No declaration for *sender*	C	universal hash	ret(i4)
Double *sender* (int, float)	C	fcn(i4,r4)ret(r8)	ret(r8)
int *sender* (char, ...)	C	fcn(c1,ell)ret(i4)	ret(i4)

The example in Figure 3–34 shows the general string for the function **precis** when the actual function resides in the same compilation unit. The universal string is generated for the function **nod** because it is in a different compilation unit.

```
COMPILATION UNIT 1                        General string
------------------                        ------------
main()                                    fcn()ret(i4)
{

        precis();                         fcn()ret(i4)
        nod();                            Universal string
}
int precis()                              fcn()ret(i4)
{
        printf("hello\n");
};

==========================================================


COMPILATION UNIT 2                        General string
------------------                        ------------
char nod(x)                               fcn(i4)ret(c1)
{
        int x;
        precis();                         Universal string
}
```

Figure 3–34. General String Constructed for **precis**.

As in the case for data, the general string is language-independent, and the hash of the general string must match between the languages for the binding to succeed.

The rules for the canonical string for parameters are the same for those for data, plus the following table. Table 3–43 gives examples of the applications of the previously discussed rules.

Table 3–43. Additional Rules for Argument Passing.

Argument Being Passed	Canonical Strings for General Hash
Character string (both length and address) by reference	a4c1 for the string address; ui4 for the length of the string
Untyped constants	universal string
Labels	FORTRAN labels in an argument list are ignored in the general hash

3.12.4 Hash Generations for FORTRAN Programs

For the following main FORTRAN program,

```
PROGRAM TEST            General String  Language String
                        --------------  ---------------
                        fcn()           fcn()main
```

Figure 3–35. Main FORTRAN Program.

The following FORTRAN data declarations are assumed for the remaining example programs presented in this section:

```
        INTEGER*2               I
        CHARACTER               C
        INTEGER*4               ARR(10)
        REAL                    Z
```

Figure 3–36. Assumed FORTRAN Data Declarations.

Table 3–44 shows the hashes generated for several FORTRAN statements.

Table 3–44. Hashes Generated for FORTRAN Statements.

FORTRAN Statement	General Hash	Language Hash
CALL SENDER(C)	fcn(a4c1,i4)	Universal
CALL CATCHER(I)	fcn(a4i2)	fcn(a4i2)
CALL PITCH(C)	fcn(a4c1,i4)ret(r4)	Universal
Z = PITCH(ARR)	fcn(a4i4)ret(r4)	fcn(a4i4)ret(r4)
CALL ARB(*10,I)	fcn(a4i2)ret(i4)	fcn(lbl,a4i2)

For the following function:

```
FUNCTION PITCH(C)                 General Hash    Language Hash
                                  -------------   -------------
                                  fcn(a4c1,i4)    fcn(a4cl)
```

Figure 3–37. FORTRAN Function.

For the following function:

```
FUNCTION ARB(*,I)                 General Hash    Language Hash
                                  ------------    -------------
                                  fcn(a4i2)       fcn(lbl,a4i2)
```

3.12.5 Interface To Hash Function

The **hash** function provides a consistent encoding for a given character string. The character string can be encoded all at once or in parts based on the value of *n*, which is the fourth parameter to the **hash** function.

```
int hash (phash,input,len,n)
    int  phash;
    char input[];
    int len;
    int *n;
```

Figure 3–38. Interface to Hash Function.

The parameters have the following meanings:

phash An integer value containing the cumulative hash, which is updated and returned.

input A character array of 256 characters containing the canonical string representation to be hashed. If the string to be hashed is longer than 256 characters, then multiple calls to the hash function can be made.

len An integer value containing the number of characters in the array input that is to be hashed.

n A pointer to an integer. The integer pointed to by *n* will be set by the **hash** function to contain the number of characters that have been hashed so far, as follows:

$n = n + len;$

Note: For every new input string to be hashed, *n* and *phash* must be set to zero before calling the **hash** function. However, if a string is being hashed in pieces (more than one call to the **hash** function), then *n* and *phash* should be ignored until the entire string has been hashed.

3.12.6 hash.c Function

The following example illustrates the source for the **hash.c** function:

```
/* Computes a hash based on the input string (”input”), the
    input phash and the number n. If n=0, phash is ignored,
    otherwise it is factored into the returned hash. n is
    updated by the length of the input string.

    Usage:
      For convenience, hash is written to be serially
      reusable, and can be called several times with a series
      of character string arguments, supplying the returned
      (partial) hash and the current value of n. For example,
      if len_i is the length of the ith string of interest,
      string_i, then

         n=0;
         for (each_string_i__of interest,i++)
             phash = hash(phash,string_i,len_i,&n);

    For language processors, the canonical string should be ASCII
    encodings of the system–defined type code strings,
    independent of what system the compiler is running on
    or compiling for. */


extern struct{
        char RAND[4096];
}_RBYTES;
int hash (phash,input,len,n)
int phash;
char input[];
int len;
int *n;
{
        int X;
        union t0 {
                char hchar[4];
                int hnum;
        } u0;
        union t1 {
                char nchar[4];
                int nnum;
        } u1;
        union t2 {
                short k;
                struct s {
                        unsigned f1 : 12;
                        unsigned khex : 4;
                } s1;
```

```
        } u2;
        union t3 {
                char next;
                struct {
                        unsigned nextbit: 8;
                } s2;
        } u3;
        short i,j,index,r16,sub;
        short used[16];

        if (*n == 0) u0.hnum = 0;
        else u0.hnum = phash;
        for (i=0; i < len; i++) {

                u3.next = input[i];
                r16=(((*n+i)*16) % 4096);
                u2.k = (*n+i) / 256;
                sub = u2.k*256;
                for (u2.k=0; u2.k <16; u2.k++)

                        used[u2.k]=0;

                for (j=0; j<4; j++) {
                        index=((j+u3.s2.nextbit) % 256);
                        u2.k = 0;
                        X = sub+index;
                u2.s1.khex=((RBYTES.RAND[(X/2)]>>(4*((X+1)%2)))& 0X0F);

                        if (used[u2.k] = 1)

                        {
                                do {
                                        u2.k= ((u2.k+1) % 16);
                                }         while (used[u2.k] != 0);

                        }
                        used[u2.k] =1;
                        u1.nchar[j] = RBYTES.RAND[r16+u2.k];

                }
                u0.hnum = u0.hnum + u1.nnum;
        }
        *n = *n+len;
        return u0.hnum;
}
```

Figure 3−39. Source Code for **hash.c**.

3.12.7 RBYTES Data

Figure 3−40 is the code for the RBYTES data structure with data. This information is used by the **hash** function to generate hash codes from the general and language strings.

```
.data
.align 4
.globl _RBYTES
_RBYTES:
.long   0X4C208DB8,0XB01759FA,0X3FF36FDF,0X98E18712
.long   0X406869DB,0XB4417115,0X0EE0E42C,0X5E214CA7
.long   0XF97B038A,0XD4AB342D,0XD9F537DF,0X31DA2F65
.long   0X3317A134,0XA6DF3AC5,0XBC773CD1,0X95884842
```

```
.long   0X8B8DA236,0X07D15587,0X2FCD7C47,0X4A6BD8EA
.long   0XE25241B8,0X3AAda149,0X4BB38202,0XF6BBB7B4
.long   0XDC896273,0X8628ACB6,0X01A7CC13,0X59D43256
.long   0X514BB099,0X3213908E,0X69B3FA79,0X3A6B8DCD
.long   0XF162C8F2,0XD7514409,0X22D1EE4C,0XBE5B4E92
.long   0XDF8B9C53,0XBD3FDBE5,0XAda55A0D,0XB7962EFC
.long   0X19C5ACBB,0X93387CF2,0X2658FB6A,0X641212E9
.long   0XBA926BA3,0X2882D84D,0X8432728D,0X139534DA
.long   0X0873B106,0X0FEDD65F,0X4B60F3A9,0X033F5E05
.long   0X5DBEAD94,0XD70CCD9F,0XDB49AC86,0X0BEBE5E9
.long   0XD768021F,0X7135C852,0X1269B95C,0X8268E4DC
.long   0XBD075392,0X3DC8C1E2,0X9A3EEEBF,0XCCA0465C
.long   0XE89BAD58,0X7102668B,0XF0112CF6,0X2F78CD72
.long   0X80036D0A,0X4A71C575,0XFBB3DEAC,0XE80C3577
.long   0XC688BCC5,0X5DC7A4C2,0XD47A1EA5,0XB5DC9063
.long   0X5D11C1A7,0XF6EF7723,0XE4EFDC65,0X3B1BCC2D
.long   0X5A0DF740,0XCADF8916,0X7E215A3E,0XF78D450D
.long   0X83B1898A,0XF8D387B4,0XB7570314,0X08C2C5E1
.long   0XE350C181,0XEBFE2E34,0X0917DFAC,0X2552145E
.long   0XD6EB7E2C,0X251ED434,0X8414B813,0X6A7E8826
.long   0X5E9428F1,0X98A363D8,0XD78B8FBF,0X5E1C1FDF
.long   0XA0D5458F,0XD7A4E3F0,0X961A7458,0XC067D5EB
.long   0X66CB14E7,0X021EB9C5,0XA362DC9D,0X79ABA727
.long   0X8281FB39,0X3B974C93,0X44C377B6,0XE977E243
.long   0X415DE7BA,0X28EA15A6,0X297BF962,0X75BED458
.long   0X3A61D7E3,0X2392EAE8,0X0E4CE108,0X85246A73
.long   0X0190ED4C,0X288BC4B3,0X4DA77E69,0X406057CC
.long   0X3465E5CA,0XEE9DBB19,0XF88EA355,0X0D5054C6
.long   0X05B04053,0X8B593A02,0X79CECDA5,0X83A2212F
.long   0X19297355,0X2B8C8B7C,0X96942EA9,0X758A32F5
.long   0X39481D12,0X0C1E4395,0X7626AF5C,0XF5FCC9F8
.long   0X967434D9,0X92158A0F,0X8478E56C,0XA2E494FF
.long   0XE6F7EF2A,0XC2943FBB,0XE0D6AF7E,0XA325BF6E
.long   0X1EC86B49,0X3AD83CBA,0X7E4CB02D,0X5F78A1B9
.long   0X3DDB1BD4,0X6AD21FE6,0XAFDCD25A,0X3130F2D4
.long   0X107E56BC,0X993E73A3,0X76597139,0X792B0D76
.long   0X73C2B56D,0X9E17D967,0X11DC924A,0X28D0BD0F
.long   0X13703892,0XD7720E9F,0X09ACAD4E,0XE03D48D5
.long   0XD48A644E,0XD71E07F5,0XFB871A34,0XDBA3382E
.long   0XAFE75103,0XE3E0A4AA,0X538844E7,0X8755E524
.long   0XEE6102FC,0X97BE0F37,0X82CCE701,0X5C75B726
.long   0XA0591BCE,0X542DD825,0XE5A3F30A,0X5E40CC43
.long   0X8DCF57E2,0XB4C92B87,0X8A09AFB8,0X223E8AD7
.long   0X9D0EB8FF,0X91F8349E,0X704918A3,0XC46905F7
.long   0X625D7085,0XEEEB3654,0X6466EAFE,0X29FC7D6B
.long   0X3A87F33A,0XDE59D8A2,0X3FB21187,0X9A28C501
.long   0X5377F74A,0X4D009ACD,0XE494427D,0XD22D5567
.long   0XC945CC39,0X578E47B5,0X37F8EAD7,0X67596663
.long   0XED698C9A,0XE556C179,0X9EB294C2,0XF4277F14
.long   0X31AF89AB,0X7695BD20,0XB5DB872B,0X07FF89BB
.long   0XB440D102,0X63FB043A,0X5FA8B612,0X26F681A3
.long   0X91F58216,0XB0BC4B11,0XDE28F12F,0X05175F05
.long   0X4C436772,0X16551EBE,0XE3283F66,0X257ABA10
.long   0XF06D4EF3,0X1BEB8E8B,0X3943F54F,0X422917D1
.long   0XBD2597EE,0X2B76768C,0X56B1419B,0XA573AF11
.long   0X52A2B092,0XC9673017,0XF4C03A81,0XC3B9AF2E
.long   0XDA8496AA,0XA0AB1ED2,0X665F2D01,0XAF9B7918
.long   0X95A31B32,0X69A25CEC,0XFABB2AE5,0X2D316E47
.long   0XDCF3A7B3,0X745EB2CB,0XBA9DAFE9,0XB0A31825
.long   0XE11A86D1,0X1B45A9C9,0X5C82C01B,0X9BDB14E6
```

```
.long   0X5B9EBDEC,0X4AF9164B,0XFA11D162,0X4F41D24A
.long   0XEACF9755,0XE2AEE187,0X74BC1C91,0XCA2F1AC0
.long   0XA4BDC446,0XE5CC20C1,0XCE1EEE89,0XA9CD8362
.long   0XFAD43447,0X5223DBD1,0X6A07346A,0X188E63C9
.long   0X789D2500,0X175997D1,0X66A2E6DF,0X79A8DE42
.long   0X953CF472,0X3052D930,0X5EF23166,0X3082FBC5
.long   0X59007B62,0X9E51E131,0XDBD4D045,0XC4FF3345
.long   0X34CF4741,0X58B16DF1,0X8AAEEB37,0X3222B731
.long   0XDEFEF8D9,0XA74BE261,0XC887DA21,0X3FEA466B
.long   0X3B348410,0XB4EADE6E,0XE4BED3AD,0XEA055A41
.long   0X90C849DD,0XC162D281,0X57D76B37,0XFBDA7859
.long   0X099E3AF8,0X4D93099A,0X594914A3,0X5A2BECBD
.long   0X07388570,0XD28A08C0,0X2BB9F9A0,0XC5F2BF00
.long   0X97F5CC07,0X5C06A7AA,0XAC305307,0XE5673FCE
.long   0XB51311E8,0X3E1BA6A1,0X276AE447,0X8EA8FF39
.long   0X149F5A9C,0XEC4B05D0,0X55A1AF15,0X295A96AF
.long   0X223F3587,0XE3CCBE6E,0X0E9AB97E,0XA370B9BB
.long   0XAE8B2283,0XE779ADFF,0X35057D81,0X2EA75649
.long   0XC913AE3B,0X8EBD2085,0XEF89C34C,0XB6F3C143
.long   0X07A4AD29,0X25ADF41A,0X270F15A3,0X5F6E32D6
.long   0XB5703AA6,0XBB47390C,0X567CC288,0XA207EA63
.long   0X44559725,0XB0445214,0XFD0DC66B,0X5DDA8215
.long   0XE2F2867A,0XEC636E4C,0X393A80AF,0X6F0C521F
.long   0XA7B18E76,0X70FAC949,0X9221509C,0X4DA037E9
.long   0X46EC3DBC,0X2D2CB064,0X3AE8C691,0X028C318E
.long   0X65F81E9F,0X01E3F65D,0XC0B40BF4,0X8AB8D7E3
.long   0X35A24B4C,0X22EB082A,0X911025E9,0X088E57C0
.long   0X3C482382,0X940F412F,0X9D4FAD1B,0X503EB6DB
.long   0X3BC4F771,0X9AB178B0,0X635D51FF,0X97EFDE5E
.long   0XA0CCDC6E,0X43412CEC,0X7ACBC40D,0X1276E7C5
.long   0X59714913,0XEF22A84D,0X15DAF9A4,0X68244CAD
.long   0X1C7AADC7,0X7C7EEE66,0X55AEAA8B,0X56FCBA36
.long   0X0162D643,0XB063B4AA,0X09B07E18,0X3728D92F
.long   0X4C1C929E,0XC1784D8F,0X06D2DC8B,0X887B586F
.long   0XB9B38C69,0XE8A06C3E,0X50CA6A44,0XC62715A2
.long   0XA868D943,0XF903AAB6,0X1B6B3076,0XEF1BF5AA
.long   0XE4AC82A6,0XF1C1AA5E,0X589E0368,0X3FCD702B
.long   0X6B0431D0,0X94DA4B31,0XD63D57D6,0XE654BB12
.long   0X56D5C045,0X8ABC1314,0XAA4686EA,0X4B32BB40
.long   0X92944AEF,0X3CD75B7B,0X6BF7E0FD,0X36907387
.long   0XCE2838E9,0X11FF81E4,0X45A981F7,0X298431C7
.long   0X142E6BA0,0X7517DF2C,0XDFDCDE0A,0X0D16D530
.long   0X2F19C57A,0X7F44C520,0X6EC25F38,0X5717D1B0
.long   0X982177BF,0X3D65F116,0X18BF39BA,0X173A7901
.long   0X05DA7E3E,0XCF0350EF,0X63715DBF,0XBBC430A2
.long   0X36AA5A2A,0X32BCFE85,0X4574AB62,0XBEFFC05F
.long   0XEA0CD553,0X47184BB3,0XD0F6AC0E,0XD98D0DED
.long   0XA808FD74,0X15C09C99,0X97B10894,0X2A6290C1
.long   0X97B53CCA,0XAA71B4CF,0X10257E23,0XDC13978D
.long   0XF98F0077,0X7B3779AB,0XAC4D6207,0X84776DCF
.long   0XCA57AE30,0X1B07735B,0XD6567B55,0X4A646CC3
.long   0XAFB7ED3B,0XC34A54F5,0X5A621058,0X3C404A98
.long   0X712C9C01,0X84CF0E79,0X3B12959E,0X9C51822C
.long   0XB9D447D9,0X886FE45C,0XAA9D43D7,0XD1427D4A
.long   0XC0381163,0X0077A09E,0XF13B07C7,0X8A9C8145
.long   0XDB1149A0,0XA919681E,0X367C2B8F,0XFD155FB8
.long   0XDA662704,0X3880724C,0XECB2F4BC,0XA17E77F3
.long   0X829CE42B,0X331776A5,0XEAB39E5D,0XFEDA4713
.long   0XED1B6A58,0X149B4D88,0X768E7BD2,0XD8F89BC0
.long   0X43C5B783,0X5368FCDA,0X2956E7E0,0XF26ED12C
```

```
.long   0XEA088F55,0X597D2B14,0X315799BD,0X0FF2B581
.long   0X748F33BA,0XD343D8D6,0XD56FD6E6,0XCDFCAD93
.long   0X83B1E9B8,0XFB149D43,0X353FCC7F,0XB9C9B0BD
.long   0X15E26B46,0XE8993801,0XB2B5FC42,0X7B0D2E1E
.long   0X426E48C8,0XE74EAF41,0X57DF04B5,0X55B5C669
.long   0X7AFB591F,0X11C22266,0X3E42E7E3,0XCF0B6FE3
.long   0X3D053C6B,0X5238B1FC,0XE601F837,0X2A0E4167
.long   0X893FF99C,0X8FB5ABE4,0X74FA247C,0X95265D7F
.long   0X41E9A03B,0XD11A456C,0XC8F0ECEF,0XC2806C44
.long   0XDD2EE868,0X77394104,0X14AE6B1E,0XC1AFB8DA
.long   0X9FDA73AC,0XDDCB0EAD,0XE1A2BE3C,0XDEF4FF31
.long   0X992EA91D,0X4A55E27F,0X3033FD1B,0X7A5AEDF4
.long   0XB4752F18,0X58B93B7D,0X9B8C1FC9,0X3B98AEAA
.long   0XC9F3E9D7,0X898E7311,0X9C3308C3,0X6F31D309
.long   0X1943402C,0X0B59FFD5,0X1FDB2F86,0XD67C6C86
.long   0X4E2D9571,0XCFBB162E,0XF3F7560E,0X78C96B95
.long   0XA46E9B6F,0X70C26C7B,0X631271F8,0X1A1741FF
.long   0XE7237867,0X30E98E99,0X3A713BC1,0X3BE8412D
.long   0XC92891EA,0X40470101,0X96BBB310,0XFD8B29E2
.long   0XACB03416,0XDC0994E4,0X1D797705,0X9827F4E5
.long   0X48633628,0X7D38B593,0X27CD8D19,0XC7BB32D8
.long   0X4A430C0D,0X87CD04A4,0XE2ECB223,0X88BDB8DC
.long   0X89615F5F,0X7FDE03F0,0X44FCD842,0X68EB84B7
.long   0X290E49AC,0X0945ECDD,0X55D9353D,0X14F67888
.long   0X25C21AB3,0X09355C8C,0XE95EC71D,0XA79B7494
.long   0X1AFDE51A,0XAD39E242,0XA3F35D36,0X48439296
.long   0X4B2F0F90,0X27367849,0XBA881B44,0XEDF980A6
.long   0XD3CE3008,0XA4E91250,0XD2DB4A29,0XDE7B8D78
.long   0X29CCD5DF,0X95FD362F,0X5103F42A,0X2AF3BCD9
.long   0XB663F7D0,0X1A182519,0X17C541DF,0XBE2711EE
.long   0XF565E925,0X0F79C5A7,0X4E688756,0XD82668C6
.long   0X8261A738,0X885D7BDD,0XC90F79A8,0XB0A44C2E
.long   0XD288705D,0X61023953,0X09F09DE3,0XB2B6D988
.long   0X981B01A3,0X5722390F,0X2D91D03D,0XD4B4E78D
.long   0X270E17C2,0X49B34218,0XAAABF907,0XD6C07278
.long   0X15D30042,0XE836E2D0,0X36CEC340,0X6CBA25B0
.long   0X7D9BFBDA,0X0B2DC499,0XA4E25CAF,0X140E6C39
.long   0XB4535363,0X9E2F24B0,0X592EE0B2,0X5E4D58A7
.long   0XB9ABDF34,0X302A997F,0X6585FDD7,0X776C843B
.long   0X6DA24C10,0X1C2E7482,0XECA3E3E3,0X4D4E26A3
.long   0X1DD9BF1B,0X5C14328E,0X377A0593,0X05B8B0A5
.long   0X27929281,0X02F7B0CE,0XACFA8DC5,0X217AF00A
.long   0X9F4B2C75,0XFEFFC735,0XB9A1910C,0X5618D3FF
.long   0XADF5085A,0X0B37D470,0XF97A809D,0XE963AE74
.long   0X728C9139,0X3EBE415A,0X8D015D14,0X9FDE5E3A
.long   0X64025291,0X17034B90,0X63512BFF,0X5F6F6A12
.long   0X36F60539,0X8742016D,0XD810779A,0XD66EE4A2
.long   0X41316976,0X31E5Ada7,0XDF4D93B0,0X0F149D16
.long   0XCA6A39A8,0X4E7EA528,0XC4E8C1C5,0X9E1F2488
.long   0X497BCA49,0XA1CC7529,0XBB033D62,0X8099C08A
.long   0XF773EBFC,0XA073025B,0X014DF339,0X1ECE93F6
.long   0X41201FD2,0XC1B91962,0X999D49E2,0X66C473B1
.long   0X2AE01320,0XEACB2DE9,0X01D40416,0X8585D2BA
.long   0XA80240C8,0XECBDB6EF,0XA698DB3D,0X45A7EDBB
.long   0XAA2E41F7,0XFCE0330E,0X178F6641,0X67CE4E8D
.long   0X6EB09ECC,0XC0B35B60,0XF3685472,0XF0436B8F
.long   0X8F65ED5C,0X2C68D121,0X7FBF33A2,0XA0DE5299
.long   0X2F3FD4C4,0X12F4B04D,0XF3E76D89,0X4CAF17C0
.long   0X5A4EE58C,0XAB66FAEB,0XE8B68CE1,0XEA5B89CE
.long   0X229E6E8A,0X40B889F3,0XBFC6211B,0X5EB92586
```

```
.long   0X46E85D4F,0X339227CB,0X5E354168,0X8F6B1DF6
.long   0X540BAEE1,0XA5618B19,0X5351880F,0XE399D55D
.long   0X2AADF0D7,0XC2694B3E,0X9C3D1D41,0X1FF58371
.long   0XE2806034,0X331BC6BA,0X83E9FE51,0XA4B53C63
.long   0X859ACA13,0X570C7C23,0X4854E4CA,0XE5FC6BCB
.long   0XA57521BB,0XDCFAD49D,0XAD87922A,0X022C5BB1
.long   0X5A9509B0,0X99541229,0XB1B5321A,0X26DCD2B6
.long   0X5EB0AC87,0XA7976495,0X77F727DD,0X768F8732
.long   0X7A9237EC,0XFE80DAD2,0X25BC3727,0X169F2725
.long   0X5542BC79,0XE183BB0D,0XAA799079,0XFDA6FD9B
.long   0X2F8439FF,0X49DC551B,0X637D1180,0X64E56A00
.long   0X16AB7325,0XC37E5958,0XBCA78956,0X63808E96
.long   0X25305DB5,0X168092CF,0XFFCC7C7D,0XFDEA8925
.long   0X14E646BC,0XDC9F1139,0X2FE7F34E,0XAE30BC57
.long   0X5E69B350,0X67ADD96F,0X76E450D3,0XFABE8E04
.long   0X8B147E37,0XBA061CA7,0XB9D2622F,0X1A0E3AC1
.long   0X2611DD27,0XF026D4FE,0X3B8B7EFE,0X4B64CAA0
.long   0X84C11ED5,0X1865CBB3,0X21CB7319,0X9E770C42
.long   0X1D311E57,0XAEEC8CE1,0XD3D16492,0XB73D9829
.long   0X4AA92A10,0X829D1A1D,0XA0D4ED20,0XBEC96F93
.long   0XADE0B88B,0X8708832C,0X60848175,0XCD321509
.long   0XB9472771,0X676D7CF9,0X11FD33E1,0XBBE7F6D0
.long   0X808B0E99,0XB003A2C6,0XFDF10F1E,0X2AFCDD7F
.long   0X44E763A0,0X0093AC2D,0X6B8DAE33,0X7B3422E6
.long   0X568459A4,0XBDAE0A98,0X517EB42A,0X2874EC3C
.long   0X5E91E098,0X83C4E9F3,0X2D6863D5,0XB9EC9E0A
.long   0X0E744022,0XCA17E608,0X9AEC76DF,0X711621CB
.long   0XDB35EE7A,0XD5B5F62F,0XDA567768,0X91D204DD
.long   0XD586B9D4,0X46027AB5,0XCB7BD8C8,0X01F56B15
.long   0XED96CC98,0XAC7CE8F8,0X4345D2E6,0XD050CFA2
.long   0X6D8EBB23,0XCC4C77AE,0X12353FE7,0XE6116B64
.long   0XF94ABE9A,0X094602A9,0X683C2987,0XC9ED50AB
.long   0X741D674E,0X006A4F04,0X43274F9F,0XEB15DD12
.long   0X354D88FA,0X0F3A0BE6,0X0F344CC1,0XCA009C23
.long   0X2BEBCC50,0XC8FEAAD6,0X0143C26E,0X12654E7F
.long   0X689593BB,0X1337FC27,0X71C57A0B,0X6463EC8C
.long   0X8A6AF8B4,0X0BA4B652,0XFDDE8B99,0X12ACC4E4
.long   0X5E54046A,0X1F05F5A2,0XC216F906,0X24A2CC21
.long   0X90FBE88D,0XA24A4ECA,0X5D219E9C,0XFF77BA98
.long   0X56F3A9D2,0X8CA389A5,0XBACAF0E1,0XACBF5EA6
.long   0X2FD0C7A4,0X2C3A8F64,0X5BDB2B48,0XA0EF4151
.long   0X396226D0,0X92D7374F,0XF8845215,0X55F6777B
.long   0X1455ECE7,0XD530EC4C,0X7F241E01,0X2815B5F9
.long   0X1B41AD26,0XAD613A36,0X39860F32,0XB9E7BC95
.long   0XBCDC6945,0X37773141,0XCA997780,0XB35F5256
.long   0X2CF82B4C,0X1089002F,0X82D833DA,0X8021BC75
.long   0XE7052F9B,0X19050265,0XC2F5DE48,0X751F6D8B
.long   0X853C7C86,0X000B68ED,0X4EB47814,0XB75383AA
.long   0X045411B4,0XF8743B31,0X75731FE1,0XAE327085
.long   0X254A0B9A,0X242AFAAC,0X7EE7476E,0XE3803188
.long   0XF2C8FA1B,0XDD0869B7,0X15B783E1,0X9A8744AA
.long   0XC881FFF2,0X0B431484,0XDF825643,0X26A90DDC
.long   0X188336E5,0X24A0E842,0XFBEE3C86,0X0F0BF481
.long   0X74000023,0X285B607C,0XCE58F710,0XEEAFE825
.long   0XEF0F4147,0X811993DA,0X818A565D,0XB84FC835
.long   0X5AE2ADBF,0XB5F06B57,0XA9560234,0X19C9A95F
.long   0X1932A43C,0X35667EBF,0X36B13ED2,0X9BDFE2E0
.long   0X31975F77,0XB6A55ECC,0X845BB85A,0X3E6CD262
.long   0XE59D2333,0X224F4EE4,0X93698F24,0X7000628A
.long   0XB9254809,0X40A04FF6,0X7E870930,0XEA1D648D
```

```
.long   0XF7E6D194,0XF9B40364,0XF1E17BF3,0X809423F3
.long   0X2CA78AC2,0XD7ABF38F,0XC85BEED8,0X7A6B9AFB
.long   0X1339D733,0X3B533C9B,0X64775460,0X99CD4B0B
.long   0X8611A71F,0X21234B63,0X3ED99A45,0X55C2B848
.long   0X94B92450,0X78330846,0XEC71D484,0XFE56A3B1
.long   0X00A67A57,0X1122CFD7,0XC4BF4BA1,0XFA6747D3
.long   0X99985E16,0X7D1D9A1D,0X47BC14A9,0X42FA9699
.long   0X14D8E8A7,0XED0D103B,0X7E012A91,0XA0A4331D
.long   0XE9B43B15,0X4D55BE94,0X5C3D134E,0X0D9FF9C2
.long   0XC01E3D67,0X803198B8,0X6E01E36E,0X48F3AD50
.long   0XF7A11A5A,0X51873869,0XDF246305,0X9094796C
.long   0XBF411427,0X62846241,0X222ADCA0,0X749AB136
```

Figure 3−40. RBYTES Data Structure with Data.

3.13 Operating System Interface

This section defines the interfaces that must be provided.

3.13.1 Virtual Address Space

Instruction and data references generated by programs are all 32-bit effective addresses. The memory management hardware translates 32-bit virtual addresses to real addresses via a 4-bit segment register identifier and a 28-bit offset. An application has three logical segments created when it is started, the text, data, and stack segments. The stack segment is initialized to a 16 byte boundary; i. e., the least significant 4 bits of the stack address are zeroes. The text segment has read (execute) permission, while the data and stack segment(s) each have write permissions in addition to read (execute). The limitation on the size of the executable is 256 MBytes since it must fit into the text segment. Applications must not assume the location of program data. The assignment of the location for program data is a function of the system.

Memory is organized by pages, which are the system's smallest units of virtual to real translation. The PowerOpen page size is 4k bytes. Applications can call *sysconf*(_SC_PAGE_SIZE**)** to determine the system's page size.

The page which starts at effective address 0 must allow read access to the application process. In addition, the first 256 bytes (effective addresses 0 through 0xFF) must be zero.

Page 3−10 has a list of functions which must have entry points at the specified effective addresses. The application must have read (execute) permissions on these effective addresses.

3.13.1.1 Small Memory Model

The operating system provides a memory segment that is used by the process for data and by the operating system for information about the process. This memory segment is called the process private segment and its layout is shown in Figure 3−41.

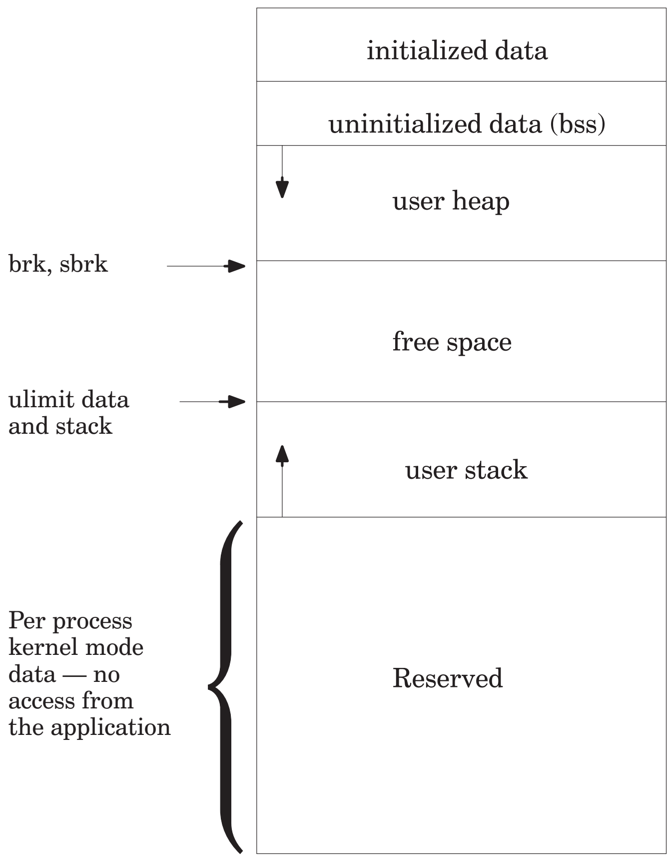

Figure 3−41. Data/Stack in the Small Model Memory Segment Layout.

As a result of placing user data and the user stack within a single segment, the system limits the maximum amount of stack and data to slightly less than 256 megabytes. (The reserved areas are relatively small and of fixed size.) While this is sufficient for the majority of applications, some require extremely large data areas. These are usually of the form of large uninitialized data areas in the program's data section or large data areas created dynamically via **malloc**, **brk**, or **sbrk**.

3.13.2 Large Memory Model

Another address space model is available to enable large data applications. Programs that do not need this large address space will continue to be described by the smaller, more efficient model. Programs that need the larger model indicate it to the system by setting the **o_maxdata** field in the auxiliary header file in the executable to indicate the actual amount of data needed. Figure 3−42 describes the large memory model.

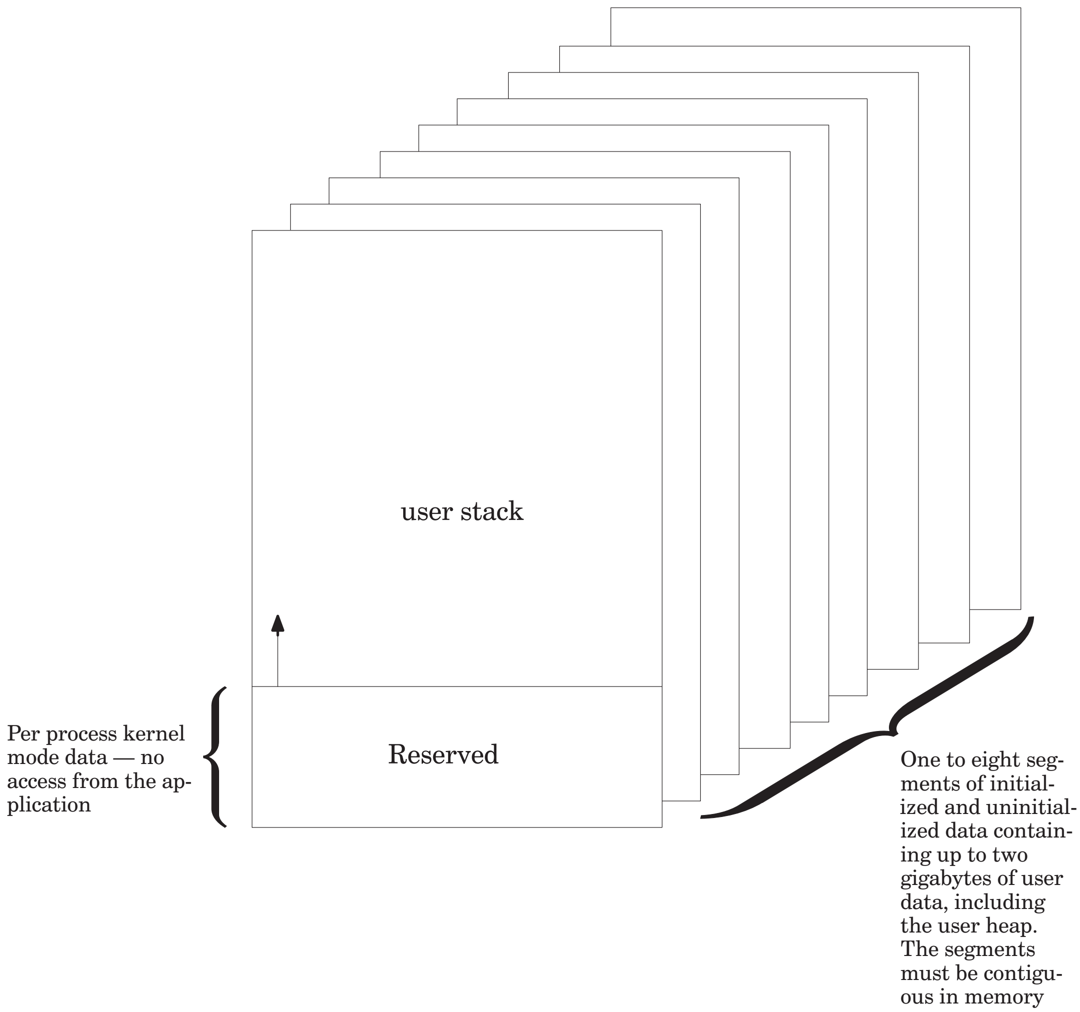

Figure 3−42. Large Memory Model Segment Layout.

With the large address space model, the program's data starts in a new segment that is distinct from the segment containing the stack. The program consumes as many segments as are needed to hold the amount of data indicated by **o_maxdata** up to a maximum of eight. This allows the process to have up to two gigabytes of data.

Other aspects of the process address space are unchanged. The user stack, kernel stack, and ublock continue to reside in the process private segment.

There are several consequences of this organization that are worth noting. First, the user stack is still bounded by the size of a segment (250 Mbytes) though it can be relocated into a shared memory segment, if desired. Second, fewer segments are available for mapped files and shared memory.

3.13.3 System Configuration Structure

The **<systemcfg.h>** file specifies the layout of a structure (_system_configuration) that is bound to an application at runtime. The operating system must create the _system_configuration structure at system initialization. It must be available to applications as read-only. This structure contains information about the architecture of the platform and the configuration of the specific machine. The header file is described in Figure 3−43.

```
extern struct {
    int architecture;   /* processor architecture                    */
    int implementation; /* processor implementation                  */
    int version;        /* processor version                         */
    int width;          /* width (32 || 64)                          */
    int ncpus;          /* 1 = UP, n = n-way MP                      */
    int cache_attrib;   /* cache attributes (bit flags)              */
                        /* bit  0/1 meaning                          */
                        /* ------------------------------------      */
                        /* 31  no cache / cache present              */
                        /* 30  separate I and D / combined           */
    int icache_size;    /* size of L1 instruction cache              */
    int dcache_size;    /* size of L1 data cache                     */
    int icache_asc;     /* L1 instruction cache associativity        */
    int dcache_asc;     /* L1 data cache associativity               */
    int icache_block;   /* L1 instruction cache block size           */
    int dcache_block;   /* L1 data cache block size                  */
    int icache_line;    /* L1 instruction cache line size            */
    int dcache_line;    /* L1 data cache line size                   */
    int L2_cache_size;  /* size of L2 cache, 0 = No L2 cache         */
    int L2_cache_asc;   /* L2 cache associativity                    */
    int tlb_attrib;     /* TLB attributes (bit flags)                */
                        /* bit  0/1 meaning                          */
                        /* ------------------------------------      */
                        /* 31  no TLB / TLB present                  */
                        /* 30  separate I and D / combined           */
    int itlb_size;      /* entries in instruction TLB                */
    int dtlb_size;      /* entries in data TLB                       */
    int itlb_asc;       /* instruction tlb associativity             */
    int dtlb_asc;       /* data tlb associativity                    */
    int resv_size;      /* size of reservation                       */
    int priv_lck_cnt;   /* spin lock count in supevisor mode         */
    int prob_lck_cnt;   /* spin lock count in problem state          */
    int rtc_type;       /* RTC type                                  */
    int virt_alias;     /* 1 if hardware aliasing is supported       */
    int cach_cong;      /* number of page bits for cache synonym     */
    int model_arch;     /* used by system for model determination    */
    int model_impl;     /* used by system for model determination    */
    int Xint;           /* used by system for time base conversion   */
    int Xfrac;          /* used by system for time base conversion   */
}_system_configuration; /* Values for architecture field             */

#define POWER_RS        0x0001          /* Power Classic architecture */
#define POWER_PC        0x0002          /* Power PC architecture      */

                /* Values for implementation field                   */
#define POWER_RS1       0x0001          /* RS1 class CPU              */
#define POWER_RSC       0x0002          /* RSC class CPU              */
#define POWER_RS2       0x0004          /* RS2 class CPU              */
```

```
#define POWER_601       0x0008          /* 601 class CPU              */
#define POWER_603       0x0020          /* 603 class CPU              */
#define POWER_604       0x0010          /* 604 class CPU              */
#define POWER_620       0x0040          /* 620 class CPU              */

                /* Sets of implementations                            */
#define POWER_RS_ALL (POWER_RS1|POWER_RSC|POWER_RS2)
#define POWER_PC_ALL (POWER_601|POWER_603|POWER_604|POWER_620)
                              /* Values for the version field         */
#define PV_601          0x10001         /* Power PC 601               */
#define PV_601a         0x10002         /* Power PC 601               */
#define PV_603          0x60000         /* Power PC 603               */
#define PV_604          0x50000         /* Power PC 604               */
#define PV_620          0x70000         /* Power PC 620               */
#define PV_RS2          0x40000         /* Power RS2                  */
#define PV_RS1          0x20000         /* Power RS1                  */
#define PV_RSC          0x30000         /* Power RSC                  */
                              /* Values for rtc_type */
#define RTC_POWER       1           /* rtc as defined by Power Arch.    */
#define RTC_POWER_PC    2           /* rtc as defined by Power PC Arch. */
                /* macros for runtime architecture and implementation
                   checks */
#define __power_rs() (_system_configuration.architecture == POWER_RS)
#define __power_pc() (_system_configuration.architecture == POWER_PC)
#define __power_set(a) (_system_configuration.implementation & (a))
#define __power_mp() (_system_configuration.ncpus > 1)#define __power_rs1()
(_system_configuration.implementation == POWER_RS1)
#define __power_rsc() (_system_configuration.implementation == POWER_RSC)
#define __power_rs2() (_system_configuration.implementation == POWER_RS2)
#define __power_601() (_system_configuration.implementation == POWER_601)
#define __power_603() (_system_configuration.implementation == POWER_603)
#define __power_604() (_system_configuration.implementation == POWER_604)
#define __power_620() (_system_configuration.implementation == POWER_620)
```

Figure 3−43. The **<systemcfg.h>** Header File.

3.13.4 Signals

The interfaces to specify signal handlers are defined in the API or in the appropriate kernel interfaces. Upon delivery of a signal, the receiving application's signal-handler routine can be declared as follows:

- *handler(Signal, Code, SCP)*
- int *Signal, Code;*
- struct **sigcontext** **SCP;*

3.13.4.1 Signal Delivery

The *Signal* parameter is the signal number. The *Code* parameter is provided only for compatibility with other UNIX(tm) compatible systems, and its value is always 0. The *SCP* parameter points to the **sigcontext** structure that is later used to restore the previous execution context of the process. The **sigcontext** structure is defined in the **<signal.h>** header file.

A signal handler can change the context of the process via changes to the mstsave structure in the sigcontext structure. The following registers can be changed by changing the appropriate mstsave variable:

- IAR
- CR
- LR
- CTR
- XER
- MQ (must be emulated in software)
- FPSCR
- msr_FE0 (bit 20)
- msr_FE1 (bit 23)
- msr_al (bit 24)
- All FPRs and GPRs

3.13.5 Process Initialization

This is the machine state that exists when an application is first entered from the operating system. Information about the invocation environment of the application is provided in certain machine registers. The C declaration is an example of an interface to this information:

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

`argc` is the argument count and is always greater than or equal to 1; `argv` is an array of argument strings with `argv[argc]==NULL`; and *envp* is an array of environment strings terminated by a null pointer. If there are no environment variables, then `envp[0]==NULL`.

The execution of a process will start at the entry point specified in the XCOFF file. See page 5–6, Import/Export Symbols for more information.

3.13.5.1 Registers

When an application is first entered from the operating system, the machine's registers are initialized by the kernel's process initialization routine as follows:

GPR0	Unspecified
GPR1	Initial stack pointer, aligned on a sixteen-byte boundary
GPR2	Contains the TOC pointer
GPR3	Contains the number of elements in the array of arguments
GPR4	Contains a pointer to the array of argument pointers (to *argv*)
GPR5	Contains a pointer to the array of environment pointers (an environment with no variables, an empty environment, is passed as a valid address that points to a NULL address)
GPR6:31	Unspecified
CR	Condition register; its contents are unspecified
LINK	Link register; its contents are unspecified
CTR	Count register; its contents are unspecified
XER	Fixed point exception register; its contents are unspecified
FPSCR	Floating point status and control register; its contents must be 0x00000000
FPR0:31	Floating point registers; their contents are unspecified

MSR machine status register (0x0000d0b0). External interrupts are enabled, problem state, floating point unavailable, machine check enabled, FP exception, program interrupt on FP enabled, exception disabled, alignment checking on, instruction and data relocation both on.

MQ unspecified.