intel ®

UNIX System V Application Binary Interface

January 2000

intel.

Contents

Figures

Tables

int_el ®

Introduction 1

1.1 The Intel IA-64 Architecture and the System V ABI

The *System V Application Binary Interface* defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement the interfaces defined in the *X/Open Common Application Environment Specification, Issue 4.*2 (also known as the "Single UNIX Specification") and the *System V Interface Definition, Issue 4*. This includes, but is not limited to, systems that have implemented UNIX System V, Release 4.

This document is the result of consensus among operating system vendors intending to provide UNIX and UNIX workalike operating systems on the IA-64 architecture. The vendors participating in this effort include Intel, Sun Microsystems, SCO, IBM, SGI, Cygnus Solutions, VA Linux Systems, HP, and Compaq. This specification builds upon the definitions of the *System V ABI* and supplies those aspects of the *System V ABI* which are indicated as being processor-specific. In combination with the *System V ABI* and the documents included by reference by this specification, constitutes a specification for compiler, linker and object model compatibility for implementations of UNIX and UNIX workalike operating systems on systems that utilize the processor architecture of Intel IA-64 microprocessors.

1.2 How to Use the System V ABI for Intel IA-64 Processors

The IA-64 architecture supports a 64 bit instruction set and also provides compatibility with the IA-32 instruction set. Binaries using the IA-64 instruction set may program to either a 32-bit model, in which the C data types `int` and `long` and all pointer types are 32-bit objects (*ILP32*); or to a 64-bit model, in which the C `int` type is 32-bits but the C `long` type and all pointer types are 64-bit objects (*LP64*). This specification describes information needed to construct, link and execute binaries using the LP64 programming model. In addition, the IA-64 architecture allows both big-endian (most-significant byte first) and little-endian (least-significant byte first) encoding. This specification may be used to instantiate a big-endian and/or a little-endian ABI.

This specification does not fully describe the ILP32 programming model. Since some vendors will support this model, some non-binding considerations will be covered in Chapter 7. The specification also does not describe the compatibility mode for IA-32 instruction set binaries. That mode is described by a separate ABI document.

This document is a supplement to the generic *System V ABI* and contains information referenced in the generic specification that may differ when System V is implemented on different processors. Therefore, the generic ABI is the prime reference document, and this supplement is provided to fill gaps in that specification.

As with the *System V ABI*, this specification references other available documents, especially the *IA-64 Processor Programmer's Reference Manual*, the *Intel IA-64 Software Conventions and Runtime Architecture Guide* and the *32-Bit Little-Endian IA-64 Software Conventions Addendum for IA-64 UNIX*. All the information referenced by this supplement should be considered part of this specification unless otherwise noted, and just as binding as the requirements and data explicitly included here.

1.3 Evolution of the ABI Specification

This specification will evolve over time to address new technology and market requirements, and will be reissued periodically. Each new edition of the specification is likely to contain extensions and additions that will increase the potential capabilities of applications that are written to conform to the ABI.

1.4 Additional Documents

The following documents are included by reference into this specification:

- *IA-64 Processor Programmer's Reference Manual*, Intel Reference Number SC-2766, SC-2767, SC-2893, SC-2769.

- *IA-64 Software Conventions and Runtime Architecture Guide*, Rev 2.5E, Intel Reference Number SC-2847

- *32-Bit Little-Endian IA-64 Software Conventions Addendum for IA64-UNIX*, Intel Reference Number SC-2790

Software Installation 2

For future use.

intel®

Low-level System Information 3

3.1 Introduction

The *System V ABI* leaves processor-specific low-level system information to the *Processor Supplement* (this document). The majority of this required information is documented in the Intel *IA-64 Software Conventions and Runtime Architecture Guide ("Conventions")*, which is operating-system independent. Only information that is specific to implementing the ABI on the IA-64 architecture will be described here.

Object files (relocatable files, executable files and shared object files) that are supplied as part of an ABI-conforming application must use position-independent code as described in Chapter 12 of *Conventions*.

3.2 Machine Interface

3.2.1 Fundamental Types

The following additional C language scalar data types are required. `long long` is an integral type, while `long double` is a floating-point type.

Table 3-1. Additional Fundamental Data Types

Data Model	C Type	Size	Align	Hardware Representation
ILP32	`long long` `unsigned long long`	8	4	Signed doubleword Unsigned doubleword
LP64	`long long` `unsigned long long`	8	8	Signed doubleword Unsigned doubleword
ILP32	`long double`	12	4	IEEE Double-Extended floating point
LP64	`long double`	16	16	IEEE Double-Extended floating point

NOTE: `long double` in the LP64 model is allocated 16 bytes (128 bits) of storage but uses the 80-bit extended double format internally.

Figure 3-1. Double-Extended (80-bit) Floating-point Formats
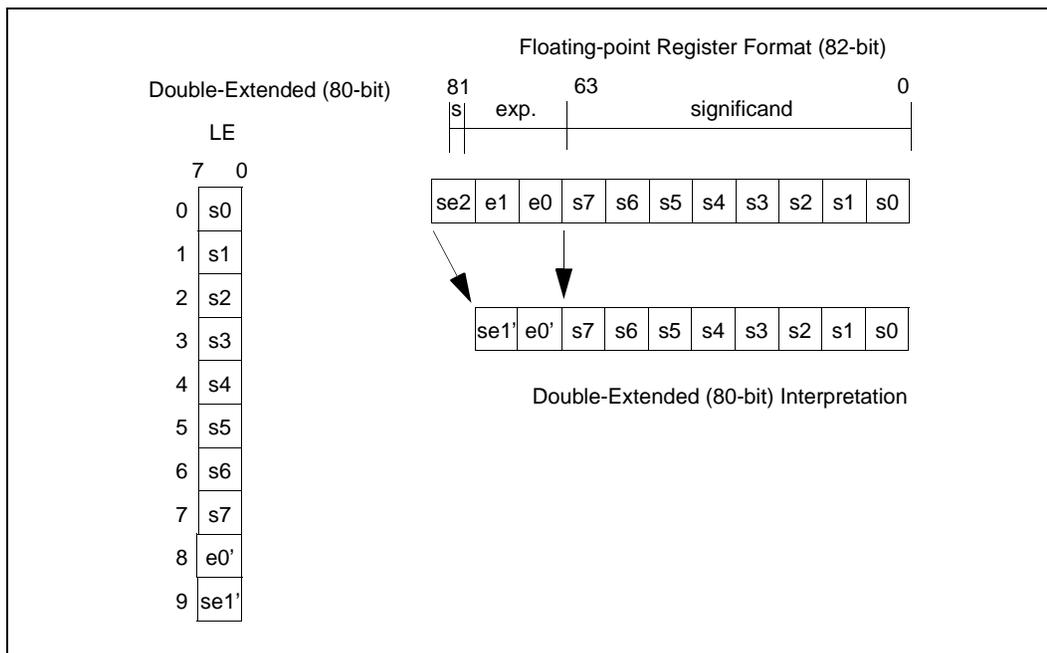
3.3 Operating System Interface

3.3.1 Exception Interface

As the IA-64 architecture manuals describe, the processor changes mode to handle *exceptions*. Some exceptions can be explicitly generated by a process. This section specifies those exception types with defined behavior. Figure 3-2 shows the signal number (si_signo) and the code (si_code) values that will be delivered for each type of hardware exception that has an effect on program execution.

Table 3-2. Hardware Exceptions and Signals

Type of Exception	si_signo	si_code	Notes
TLB faults	SIGSEGV	SEGV_MAPERR	(1)
Access faults	SIGSEGV	SEGV_ACCERR	
Privilege violations	SIGILL	ILL_PRVOPC	
Register NaT consumption	SIGILL	ILL_PRVREG	
NaT page consumption	SIGSEGV	__ILL_REGNAT	
Speculative operation	None	SEGV_MAPERR	(2)
Unaligned data	SIGBUS	BUS_ADRALN	(3)
Floating-point exceptions	SIGFPE	see Table 3-3	
Illegal instructions	SIGILL	ILL_ILLOPC	

Table 3-2. Hardware Exceptions and Signals (Continued)

Break 0 (unknown error)	SIGILL	ILL_ILLOPC	
Break 1 (integer divide by zero)	SIGFPE	FPE_INTDIV	
Break 2 (integer overflow)	SIGFPE	FPE_INTOVF	
Break 3 (range check/bounds check)	SIGFPE	FPE_FLTSUB	
Break 4 (null pointer dereference)	SIGSEGV	SEGV_MAPERR	
Break 5 (misaligned data)	SIGBUS	BUS_ADRALN	
Break 6 (decimal overflow)	SIGFPE	__FPE_DECOVF	
Break 7 (decimal divide by zero)	SIGFPE	__FPE_DECDIV	
Break 8 (packed decimal error)	SIGFPE	__FPE_DECERR	
Break 9 (invalid ASCII digit)	SIGFPE	__FPE_INVASC	
Break 10 (invalid decimal digit)	SIGFPE	__FPE_INVDEC	
Break 11 (paragraph stack overflow)	SIGSEGV	__SEGV_PSTKOVF	
Break 12-0x03ffff (reserved)	undefined		
Break 0x040000-0x07ffff (application)	SIGILL	__ILL_BREAK	
Break 0x080000-0x0fffff (debugger)	SIGTRAP	TRAP_BRKPT	(4)
Break 0x100000-0x1fffff (reserved)	undefined		

Notes:
1. TLB faults are first serviced by the system to determine if the attempted access was to a page to which the process has access. A signal is delivered to the application only if the attempted access is determined to be invalid.
2. Speculative operation faults are the result of a speculative check or floating-point check flags operation. The system services this fault, and emulates the instruction as a pc-relative branch when the fault is taken.
3. The system may emulate unaligned data references, possibly depending on flags set in the executable object file or on the executable's setting of the PSR.ac bit. If it does, no signal is delivered. Applications that rely on such behavior are not ABI conforming.
4. If the process is being controlled by a debugger, these faults generate debugger events, and do not cause a signal to be delivered to the process.

Table 3-3 details the possible reasons for a SIGFPE signal caused by a floating-point exception:

Table 3-3. Floating Point Exceptions

Code	Reason
FPE_FLTDIV	floating point divide by zero
FPE_FLTOVF	floating point overflow
FPE_FLTUND	floating point underflow
FPE_FLTRES	floating point inexact result
FPE_FLTINV	invalid floating point operation
FPE_FLTSUB	subscript out of range

3.3.2   Signal Delivery

The *Single UNIX Specification* defines information that is made available in the `siginfo_t` structure for specific signals. That information is reproduced, for informational purposes, in Table 3-4. Table 3-5 lists additional information delivered for specific signals on IA-64.

Table 3-4. Standard Signal Delivery

Signal	Member	Value
SIGILL SIGFPE	void * si_addr	Address of faulting instruction
SIGSEGV SIGBUS	void * si_addr	Address of faulting memory reference
SIGCHLD	pid_t si_pid int si_status uid_t si_uid	Child process ID Exit value or signal Real user ID of the process that sent the signal
SIGPOLL	long si_band	Band event for POLL_IN, POLL_OUT or POLL_MSG

Table 3-5. Signal Delivery – Additional Details for IA-64

Signal	Member	Value
SIGTRAP	void * si_addr int si_imm	Address of faulting instruction `break` instruction immediate operand
SIGILL	int si_imm	`break` instruction immediate operand (for `__ILL_BREAK`)

When a signal handler is installed, the application passes a function pointer to the system. As defined by *Conventions*, a function pointer points to a function descriptor, which contains the handler's entry point address and its global pointer register (gp) value. The implementation must be aware of the structure of the function descriptor in order to deliver a signal correctly.

When delivering a signal, the implementation must do the following:

1. Build the signal info and signal context records at the top of the user stack. If SA_SIGINFO was not set when installing the signal handler, these records are not required.

2. Create a new 16-byte scratch area at the top of the user stack, for the handler's use.

3. Create a new register stack frame with three output argument registers, and place the signal handler's arguments in these registers.

4. Set the global pointer register (gp) to the handler's gp value.

5. Initialize the floating-point status register (ar.fpsr) to the standard value, as defined by the common runtime conventions.

6. Transfer control to the signal handler, providing the appearance that the handler has been called, so that a return from the handler will reinstall the saved (and possibly modified) context.

3.3.3   Signal Handler Interface

According to the *Single UNIX Specification*, if the SA_SIGINFO flag is used when a signal handler is installed, the handler will be called with three arguments, according to the following prototype:

```
void handler(int signo, siginfo_t *info, void *context);
```

In addition to the several members required by *Single UNIX Specification*, the `siginfo_t` structure contains the following fields for IA-64:

int si_imm	Immediate operand for break instruction

The *Single UNIX Specification* defines the `si_addr` field as the address of the faulting instruction or the faulting memory reference. When it is an instruction address, the value is represented as a bundle address with the low-order two bits set to indicate the particular instruction within a bundle.

The *Single UNIX Specification* allows the application to cast the context argument to the type `ucontext_t`, which contains the following fields (at least):

stack_t uc_stack	The stack used by this context.
mcontext_t uc_mcontext	A machine-specific representation of the saved context.

The `stack_t` structure contains the following fields (at least):

void *ss_sp	Stack base or pointer
size_t ss_size	Size of the stack
int ss_flags	Flags

The stack described by this structure includes both the memory stack and the backing store.

The `mcontext_t` structure is an opaque structure. Its size must be specified by the ABI, but its layout is implementation specific. Each implementation may provide an API for accessing and modifying the context.

Note: REVIEW NOTE: Specification of the size is left to an external standards body.

3.3.3.1   Signal Delivery – Implementation Notes

Note: This section is informational and does not form part of the specification.

The `si_imm` field may be placed in the `_fault` member of the `siginfo_t` structure, since it is delivered only for SIGTRAP signals, when `si_addr` is also delivered.

A signal handler's return pointer must be some value that causes the saved signal context to be reinstalled when the signal handler returns; thus, it can not be an address within the range of any of the application's loaded segments. Typically, it will be the address of a kernel entry point, mapped into a shared portion of the application's address space.

The signal context record placed on the stack marks a discontinuity in the stack. While the signal handler's frame itself is an ordinary stack frame, its caller appears to be a routine whose stack frame is the context record. The system's unwind routines will need a way of recognizing the

discontinuity. The common runtime conventions provide a special implementation-dependent unwind descriptor format (P10) for this purpose. A recommended, but not required, mechanism is for the system to provide a special unwind table for the signal handler return point, using this special unwind descriptor to indicate to the unwind library that it has reached a signal context record on the stack. This unwind table is made available to the unwind library through an implementation-specific mechanism.

Implementations will likely choose not to copy the stacked general registers into the signal context record, relying instead on accessing the backing store as needed. Thus, the API routines for reading and writing the context record will need to understand the layout of the backing store in order to access and modify the stacked general registers.

If the backing store overflows as a result of flushing the register stack in preparation for signal delivery, the system may need to provide space in the `mcontext_t` record for saving the remainder of the register stack. Thus, there may be a discontinuity in the backing store, and API routines for accessing the general registers must take this into account.

The API set should include read and write routines for each element of user-visible state, plus read and write routines for the stacked general registers. The APIs should provide an abstraction layer to help the programmer deal with the complexities of NaT bits, the layout of the backing store, the frame marker, and the location of the instruction pointer within the current bundle.

3.3.4   Debugging Support

A program may use the break instruction subject to the restrictions documented in Chapter 2 of *Conventions*. A break instruction with an immediate operand with the high-order two bits set to `01` is reserved for debugger breakpoints. For purposes of implementing the *System V ABI*, a value of zero in the remaining bits (i.e. an operand of `0x80000`) is defined as the debugger breakpoint; all other values in this range are undefined.

3.3.5   Process Startup

This section describes the initial program state that the `exec` functions create when constructing a new process image. Programming language systems use this initial program state to establish a standard environment for their application programs. As an example, a C program begins executing at a function named `main`, conventionally declared in the following way.

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

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

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

The implementation will call (or appear to call) the program entry point recorded in the `e_entry` field of the ELF header, hereafter referred to as "`main`", according to standard calling conventions. The system is responsible for initializing the process state to satisfy the common runtime conventions (see *Conventions*). These initializations include, but are not limited to, the following:

1. The current frame marker must be configured for zero input and local registers, and at least four output registers.

2. The stack pointer register (sp) must be aligned to a 16-byte boundary. An initial stack frame must exist for the routine in the implementation responsible for calling main, with space for a 16-byte scratch area for use by main.

3. The RSE backing store pointer registers must be valid.

4. The return pointer register (rp) is a valid return address, such that if the program returns from the main routine, the implementation will cause the program to exit normally, using the main's return value as the exit status.

5. The unwind information for this "bottom-of-stack" routine in the implementation must provide a mechanism for recognizing the bottom of the stack during a stack unwind.

6. The global pointer register (gp) contains main's global pointer.

7. The floating-point status register (ar.fpsr) is initialized as described in *Conventions*.

The first two argument registers (r32-r33, named out0-out1 at entry to `main`) must contain `argc` and `argv`, respectively. The third and fourth argument registers (r34-r35, out2-out3) must be allocated as required by the common runtime conventions, but are not defined by this ABI.

Object Files 4

4.1 ELF Header

4.1.1 Machine Information

4.1.1.1 Programming Model

As described in Section 1.1, "The Intel IA-64 Architecture and the System V ABI" on page 1-1, binaries using the IA-64 instruction set may program to either a 32-bit model, in which the C data types `int` and `long` and all pointer types are 32-bit objects (ILP32); or to a 64-bit model, in which the C `int` type is 32-bits but the C `long` type and all pointer types are 64-bit objects (LP64). This specification describes both binaries that use the ILP32 and the LP64 model. For LP64 binaries, the `e_flags` member of the ELF header will include the value `EF_IA_64_ABI64` (see Table 4-2 below). For ILP32 binaries `e_flags` will not include `EF_IA_64_ABI64`. IA-64 files using the 32-bit programming model may not be combined with IA-64 files using the 64-bit programming model.

4.1.1.2 File Class

For IA-64 ILP32 relocatable (i.e. of type ET_REL) objects, the file class value in `e_ident[EI_CLASS]` must be ELFCLASS32. For LP64 relocatable objects, the file class value may be either ELFCLASS32 or ELFCLASS64, and a conforming linker must be able to process either or both classes. ET_EXEC or ET_DYN object file types must use ELFCLASS32 for ILP32 and ELFCLASS64 for LP64 programs.

Addresses appearing in ELFCLASS32 relocatable objects for LP64 programs are implicitly extended to 64 bits by zero-extending.

Note: Some constructs legal in ILP64 programs, e.g. absolute 64-bit addresses outside the 32-bit range, may require use of an ELFCLASS64 relocatable object file.

4.1.1.3 Data Encoding

For the data encoding in `e_ident[EI_DATA]`, IA-64 64-bit objects can use either ELFDATA2MSB or ELFDATA2LSB. That is, IA-64 64-bit ELF files may use either the big endian or little endian data encoding. IA-64 files using ELFDATA2MSB encoding may not be combined with IA-64 files using ELFDATA2LSB encoding.

4.1.1.4 Operating System Identification

The `e_ident[EI_OSABI]` value identifies the operating system and ABI to which the object is targeted, as listed in Table 4-1.

Table 4-1. Operating System Identification, `e_ident[EI_OSABI]`

Name	Value	Meaning
ELFOSABI_SYSV	0	UNIX System V
ELFOSABI_HPUX	1	HP-UX
ELFOSABI_NETBSD	2	NetBSD
ELFOSABI_LINUX	3	Linux
ELFOSABI_HURD	4	Hurd
"Unspecified"	5	Reserved
ELFOSABI_SOLARIS	6	Solaris
ELFOSABI_MONTEREY	7	Monterey
ELFOSABI_IRIX	8	IRIX
ELFOSABI_FREEBSD	9	FreeBSD
ELFOSABI_TRU64	10	TRU64 UNIX
ELFOSABI_STANDALONE	255	Standalone application - no ABI

NOTE: ELFOSABI_STANDALONE may be used to indicate applications that have no operating system dependency. Such applications are not ABI-conforming since ABI-conforming programs by definition import basic system services from a shared object library.

4.1.1.5   Processor Identification

Processor identification resides in the ELF header's `e_machine` member and must have the value `EM_IA_64`.

4.1.1.6   Processor-Specific Flags

The ELF header `e_flags` member holds bit flags associated with the file, as listed in Table 4-2.

Table 4-2. IA-64 Processor-Specific Flags, `e_flags`

Name	Value
EF_IA_64_MASKOS	0x00ff000f
EF_IA_64_ABI64	0x00000010
EF_IA_64_REDUCEDFP	0x00000020
EF_IA_64_CONS_GP	0x00000040
EF_IA_64_NOFUNCDESC_CONS_GP	0x00000080
EF_IA_64_ABSOLUTE	0x00000100
EF_IA_64_ARCH	0xff000000

EF_IA_64_MASKOS   All bits in this mask are reserved for operating system specific values.

EF_IA_64_ABI64   If this bit is set, the object uses the LP64 programming model, as described above. If the bit is clear, the object uses the ILP32 programming model.

EF_IA_64_REDUCEDFP

> If this bit is set, the object has been compiled with a reduced floating-point model. The compiler uses only floating point registers FP6-FP11 for integer arithmetic. If the program does not perform explicit floating-point calculations, registers FP6-FP11 are the only floating-point registers that need to be saved by interrupt handlers. When combining relocatable objects, a linker should set the `EF_IA_64_REDUCEDFP` flag in the resulting object only if all of the objects to be combined have the flag set.

EF_IA_64_CONS_GP   If this bit is set, the global pointer (gp) is treated as a program-wide constant. The gp is saved and restored only for indirect function calls. Objects with this bit set may not be combined with objects that do not have this bit set. This model is intended for use primarily in standalone programs, such as operating system kernels. Objects with this bit set are not ABI-conforming.

EF_IA_64_NOFUNCDESC_CONS_GP

> If this bit is set, the global pointer (gp) is treated as a program-wide constant. The gp is never saved or restored across function calls. In this model, a function's address is not treated as the address of a two-word function descriptor. Rather, it is the actual address of the function definition itself. This model is intended for use primarily in standalone programs, such as operating system kernels. Objects with this bit set are not ABI-conforming.

EF_IA_64_ABSOLUTE   If this bit is set, the program loader is instructed to load the executable at the addresses specified in the program headers. Objects with this bit set are not ABI-conforming.

EF_IA_64_ARCH   The integer value formed by these eight bits identifies the architecture version. This field is reserved for use when the IA-64 architecture is extended with backward-compatible features. It records the minimum level of the architecture required by the object code. The only currently defined value is one.

4.2   Sections

4.2.1   Section Types

The IA-64 architecture defines two processor-specific section types and a reserved range to be used in the `sh_type` member of the ELF section header in addition to the standard section types.

Table 4-3. Section Types, `sh_type`

Name	Value
SHT_IA_64_EXT	0x70000000
SHT_IA_64_UNWIND	0x70000001
SHT_IA_64_LOPSREG	0x78000000
SHT_IA_64_HIPSREG	0x7fffffff

SHT_IA_64_EXT   The section contains product specific extension bits. These consist of at least one 64-bit word of attribute flags that identify specific non-architectural extensions that are required by the object code. See Section 4.2.4, "Architecture Extensions" on page 4-6.

SHT_IA_64_UNWIND   The section contains unwind function table entries for stack unwinding. See *Conventions* for details.

SHT_IA_64_LOPSREG to SHT_IA_64_HIPSREG
Sections in this range are reserved for implementation-specific section types. A portion of this range is allocated for use by implementations which have assigned Operating System Identification values (see Section 4.1.1.4, "Operating System Identification" on page 4-1). If the high-order 8 bits of sh_type contain `0x78` then the next 8 bits contain the EI_OSABI value. For example, if the EI_OSABI value for an implementation is `0x03`, the reserved range for that implementation is `0x78030000` to `0x7803ffff`.

4.2.2   Section Attribute Flags

A section header `sh_flags` member holds 1-bit flags that describe the attributes of the section. The IA-64 architecture defines two processor-specific values in addition to the standard values.

Table 4-4. Section Attribute Flags, `sh_flags`

Name	Value
SHF_IA_64_SHORT	0x10000000
SHF_IA_64_NORECOV	0x20000000

SHF_IA_64_SHORT   The section contains objects that will be referenced using an offset from the global pointer (gp), so the section must be placed near gp.

SHF_IA_64_NORECOV   The section contains code that uses speculative instructions without recovery code. ABI-conforming implementations are not required to execute binaries that do not have recovery code associated with them.

4.2.3   Special Sections

The following special sections are defined for use on the IA-64 architecture.

Table 4-5. Special Sections

Name	Type	Attributes
.IA_64.archext	SHT_IA_64_EXT	None
.IA_64.pltoff	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+SHF_IA_64_SHORT
.IA_64.unwind	SHT_IA_64_UNWIND	SHF_ALLOC+SHF_LINK_ORDER
.IA_64.unwind_info	SHT_PROGBITS	SHF_ALLOC
.got	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+SHF_IA_64_SHORT

Table 4-5. Special Sections (Continued)

Name	Type	Attributes
.plt	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.sbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE+SHF_IA_64_SHORT
.sdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+SHF_IA_64_SHORT
.sdata1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+SHF_IA_64_SHORT

.IA_64.archext	This section holds product-specific extension bits (see `SHT_IA_64_EXT` in Section 4.2.1, "Section Types" on page 4-3 for details). The link editor will perform a logical "or" of the extension bits of each object it combines when creating an executable so that it creates only a single `.IA_64.archext` section in the executable.
.IA_64.pltoff	This section holds local function descriptor entries. See "Coding Examples" in *Conventions* and Section 5.3.6, "Procedure Linkage Table" on page 5-7 for more information.
.IA_64.unwind	This section holds the unwind function table. The contents are described in *Conventions*.
.IA_64.unwind_info	This section holds stack unwind and exception handling information. The contents specific to unwind information are described in *Conventions*. The exception handling information is programming language specific and is unspecified.
.got	This section holds the global offset table. See "Coding Examples" in *Conventions* and Section 5.3.4, "Global Offset Table" on page 5-6 for more information.
.plt	This section holds the procedure linkage table. See Section 5.3.6, "Procedure Linkage Table" on page 5-7 for more information.
.sbss	This section holds unitialized data that contribute to the program's memory image. Data objects contained in this section are recommended to be eight bytes or less in size. The system initializes the data with zeroes when the program begins to run. The section occupies no file space, as indicated by the section type `SHT_NOBITS`. The `.sbss` section is placed so it may be accessed using short direct addressing (22-bit offset from gp). See "Protection Areas" in *Conventions*.
.sdata and .sdata1	These sections hold initialized data that contribute to the program's memory image. Data objects contained in these sections are recommended to be eight bytes or less in size. The `.sdata` and `.sdata1` sections are placed so they may be accessed using short direct addressing (22-bit offset from gp). See "Protection Areas" in *Conventions*.

4.2.4   Architecture Extensions

The `.IA_64.archext` section allows a compiler to record dependencies on certain features and capabilities of a specific processor, that are extensions to the IA-64 architecture. Currently, there are no such extensions defined, and this section is not expected to be used by the compilers. Nevertheless, linkers and loaders should provide the proper implementation of this section in preparation for future architectural extensions.

The contents of the `.IA_64.archext` section, if present, is interpreted as a series of individual bits grouped into 64-bit doublewords. The first doubleword of the group is defined to correspond bitwise to the bits in CPUID Register 4 (General Features/Capability Bits). Additional doublewords in the section have no defined meaning, unless and until the IA-64 architecture is extended with additional CPUID Registers defining additional capability bits.

All `.IA_64.archext` sections must be of section type SHT_IA_64_EXT, and should have no flags set in the `sh_flags` field. Each section must be a multiple of 8 bytes in length, with 8 byte alignment. The linker must combine such sections by a bitwise OR operation on each corresponding doubleword of each section (i.e., the first doubleword of one section OR'ed with the first doubleword of the other section, and so on). If some sections are shorter than others, the shorter ones are padded with zeroes at the end, so that the combined output section is equal in length to the largest input section.

If a `.IA_64.archext` section exists in the output file, the linker must create a program header table entry of type PT_IA_64_ARCHEXT to communicate this information to the loader. This program header table entry must precede all entries of type PT_LOAD. If the `.IA_64.archext` section exists, but its contents are all zeroes, the linker may omit the section and program header table entry, but it is not required to.

When an executable or shared library is loaded, and a PT_IA_64_ARCHEXT entry is present in the program header table, the loader should compare the contents of the first doubleword of the section with CPUID Register 4. If any bits are set in the section that are not also set in CPUID Register 4, the implementation must refuse to load the file. If, in the future, additional CPUID registers are defined to identify further capability bits, the loader should check additional double-words of this section with those registers as well. If the section contains excess doublewords that do not correspond to defined CPUID registers, the loader should check that all excess bits are zero.

The linker should be prepared to deal with `.IA_64.archext` sections of arbitrary length, but it is permissible to truncate the sections to a reasonable length. It is recommended that all tools should be prepared to deal with at least four doublewords in this section.

4.3   Relocations

4.3.1   Relocation Types

A relocation entry's `r_offset` value designates the offset or virtual address of the affected storage unit. For data relocations, this is the first byte of the word or doubleword being relocated. For instructions, it is the address of the bundle containing the instruction being relocated. The least significant two bits of the offset identify the instruction slot to which the relocation applies, as described below. Each instruction bundle is 16 bytes long and 16 byte aligned; each instruction slot is 41 bits long. Whether a given relocation type applies to an instruction or data field is noted in the *Field* column of the table of relocations, below.

Figure 4-1. Instruction Bundle Layout

127	87 86	46 45	5 4	0
slot 2	slot 1	slot 0	template	
41	41	41	5	

000947
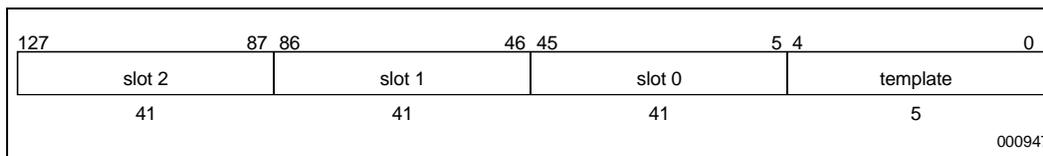
Table 4-6. Relocation Offset Instruction Slot Encoding

Encoding (last two bits)	Instruction slot
00	Slot 0
01	Slot 1
10	Slot 2
11	Invalid

Relocation entries describe how to alter the following instruction and data fields (bit numbers appear to the upper left of the field they label; all fields are numbered from bit 0).

Figure 4-2. Relocatable Fields

word32   A 32-bit field occupying four bytes with arbitrary alignment. The byte
                    order for these values is specified by the relocation type.

word64   A 64-bit field occupying eight bytes with arbitrary alignment. The byte
                    order for these values is specified by the relocation type.

function descriptor Two contiguous 64-bit words occupying 16 bytes with 8-byte alignment.
                    The byte order for the function descriptor is specified by the relocation
                    type. Function descriptor entries are created by the linker and/or the
                    dynamic linker and are used in resolving function addresses. The first
                    64-bit word contains the function address. The second 64-bit word
                    contains the value of the global pointer (gp) for the object containing the
                    definition of the function. Function descriptor entries are referenced by

relocations contained in shared objects and executable objects only and
are intended to be processed at run-time.

instruction - immediate14   A signed 14-bit immediate value. imm_{7b} contains bits 0 through 6 (low-
order bits). imm_{6d} contains bits 7 through 12. s contains the high-order
bit (sign bit).

instruction - immediate22   A signed 22-bit immediate value. imm_{7b} contains bits 0 through 6 (low-
order bits). imm_{9d} contains bits 7 through 15. imm_{5c} contains bits 16
through 20. s contains the high-order bit (sign bit).

instruction - immediate21 - form 1
A signed 21-bit immediate value. This value is formed by taking a 25-bit
displacement and shifting it right by four bits. For the resulting value,
imm_{20b} contains bits 0 through 19 (low-order bits). s contains the high-
order bit (sign bit).

instruction - immediate21 - form 2
A signed 21-bit immediate value. This value is formed by taking a 25-bit
displacement and shifting it right by four bits. For the resulting value,
imm_{7a} contains bits 0 through 6 (low-order bits). imm_{13c} contains bits 7
through 19. s contains the high-order bit (sign bit).

instruction - immediate21 - form 3
A signed 21-bit immediate value. This value is formed by taking a 25-bit
displacement and shifting it right by four bits. For the resulting value,
imm_{20a} contains bits 0 through 19 (low order bits).

instruction - immediate64   A 64-bit immediate value. The value is contained within two 41-bit
instruction slots (slots 1 and 2 of a bundle). imm_{7b} contains bits 0
through 6 (low order bits). imm_{9d} contains bits 7 through 15. imm_{5c}
contains bits 16 through 20. i_c contains bit 21. imm_{41} contains bits 22
through 62 and takes the entire width of slot 1 (the second instruction
slot). i contains bit 63.

The calculations below assume one of two contexts. First, the relocations may be contained within
a relocatable file; the actions are transforming the relocatable file into an executable or a shared
object file. Conceptually, the link editor merges one or more relocatable files to form the output. It
first decides how to locate and combine the input files, then updates the symbol values, and finally
performs the relocation. Because many IA-64 instructions have small immediate fields, the longer
form of relocation entry containing an explicit addend (`Elf32_Rela` or `Elf64_Rela`) is
always used for relocatable objects on IA-64. Second, the relocations may be contained within an
executable file or shared object; the actions complete the job of relocation by fixing addresses for
position-independent code. Relocations contained within executable files or shared objects may
use either the shorter form (`Elf32_Rel` or `Elf64_Rel`) or the longer form (`Elf32_Rela` or
`Elf64_Rela`). These relocations always apply to word or doubleword data objects.

Descriptions below use the following notation.

A   The Addend used to compute the value of the relocatable field.

BD   The Base address Difference, a constant that must be applied to a virtual
address. This constant represents the difference between the run-time
virtual address and the link-time virtual address of a particular segment.
The segment is implied by the value of the link-time virtual address. See
Section 5.2, "Program Loading" on page 5-1 for details.

P

The "Place" (section offset or address) of the storage unit being relocated (computed using `r_offset`). If the relocation applies to an instruction, this is the address of the bundle containing the instruction.

S

The value of the Symbol whose index resides in the relocation entry.

@gprel(*expr*)

Computes a gp-relative displacement - the difference between *expr* and the value of the global pointer (gp) for the current module.

@ltoff(*expr*)

Requests the creation of a global offset table (GOT) entry that will hold the full value of *expr* and computes the gp-relative displacement to that GOT entry. See Section 5.3.4, "Global Offset Table" on page 5-6 for more information.

@pltoff(*symbol*)

Requests the creation of a local function descriptor entry for the given symbol and computes the gp-relative displacement to that function descriptor entry. See Section 5.3.6, "Procedure Linkage Table" on page 5-7 for more information.

@segrel(*expr*)

Computes a segment-relative displacement - the difference between *expr* and the address of the beginning of the segment containing the relocatable object. This relocation type is designed for data structures that reside in read-only segments, but need to contain pointers. The relocatable object and effective address must be contained within the same segment. Applications using these pointers must be aware that they are segment-relative and must adjust their values at run-time, using the load address of the containing segment. No output relocations will be generated for @segrel relocations.

@secrel(*expr*)

Computes a section-relative displacement - the difference between *expr* and the address of the beginning of the (output) section that contains *expr*. This relocation type is designed for references from one non-allocatable section to another. Applications using these values must be aware that they are section-relative and must adjust their values at run-time, using the adjusted address of the target section. No output relocations will be generated for @secrel relocations.

@fptr(*symbol*)

Evaluates to the address of the "official" function descriptor for the given symbol. See *Conventions* for more information.

The MSB and LSB suffixes on the following relocation types indicate whether the target field is stored most significant byte first (big-endian) or least significant byte first (little-endian), respectively.

Table 4-7. IA-64 Relocation Types

Name	Value	Field	Calculation
R_IA_64_NONE	0	None	None
R_IA_64_IMM14	0x21	instruction - immediate14	S + A
R_IA_64_IMM22	0x22	instruction - immediate22	S + A
R_IA_64_IMM64	0x23	instruction - immediate64	S + A
R_IA_64_DIR32MSB	0x24	word32 MSB	S + A
R_IA_64_DIR32LSB	0x25	word32 LSB	S + A
R_IA_64_DIR64MSB	0x26	word64 MSB	S + A
R_IA_64_DIR64LSB	0x27	word64 LSB	S + A

Table 4-7. IA-64 Relocation Types (Continued)

Name	Value	Field	Calculation
R_IA_64_GPREL22	0x2a	instruction - immediate22	@gprel(S + A)
R_IA_64_GPREL64I	0x2b	instruction - immediate64	@gprel(S + A)
R_IA_64_GPREL64MSB	0x2e	word64 MSB	@gprel(S + A)
R_IA_64_GPREL64LSB	0x2f	word64 LSB	@gprel(S + A)
R_IA_64_LTOFF22	0x32	instruction - immediate22	@ltoff(S + A)
R_IA_64_LTOFF64I	0x33	instruction - immediate64	@ltoff(S + A)
R_IA_64_PLTOFF22	0x3a	instruction - immediate22	@pltoff(S + A)
R_IA_64_PLTOFF64I	0x3b	instruction - immediate64	@pltoff(S + A)
R_IA_64_PLTOFF64MSB	0x3e	word64 MSB	@pltoff(S + A)
R_IA_64_PLTOFF64LSB	0x3f	word64 LSB	@pltoff(S + A)
R_IA_64_FPTR64I	0x43	instruction - immediate64	@fptr(S + A)
R_IA_64_FPTR32MSB	0x44	word32 MSB	@fptr(S + A)
R_IA_64_FPTR32LSB	0x45	word32 LSB	@fptr(S + A)
R_IA_64_FPTR64MSB	0x46	word64 MSB	@fptr(S + A)
R_IA_64_FPTR64LSB	0x47	word64 LSB	@fptr(S + A)
R_IA_64_PCREL21B	0x49	instruction - immediate21 form 1	S + A − P
R_IA_64_PCREL21M	0x4a	instruction - immediate21 form 2	S + A - P
R_IA_64_PCREL21F	0x4b	instruction - immediate21 form 3	S + A − P
R_IA_64_PCREL32MSB	0x4c	word32 MSB	S + A − P
R_IA_64_PCREL32LSB	0x4d	word32 LSB	S + A − P
R_IA_64_PCREL64MSB	0x4e	word64 MSB	S + A − P
R_IA_64_PCREL64LSB	0x4f	word64 LSB	S + A − P
R_IA_64_LTOFF_FPTR22	0x52	instruction - immediate22	@ltoff(@fptr(S + A))
R_IA_64_LTOFF_FPTR64I	0x53	instruction - immediate64	@ltoff(@fptr(S + A))
R_IA_64_LTOFF_FPTR32MSB	0x54	word32MSB	@ltoff(@ftpr(S + A))
R_IA_64_LTOFF_FPTR32LSB	0x55	word32LSB	@ltoff(@fptr(S + A))
R_IA_64_LTOFF_FPTR64MSB	0x56	word64MSB	@ltoff(@fptr(S + A))
R_IA_64_LTOFF_FPTR64LSB	0x57	word64LSB	@ltoff(@fptr(S + A))
R_IA_64_SEGREL32MSB	0x5c	word32 MSB	@segrel(S + A)
R_IA_64_SEGREL32LSB	0x5d	word32 LSB	@segrel(S + A)
R_IA_64_SEGREL64MSB	0x5e	word64 MSB	@segrel(S + A)
R_IA_64_SEGREL64LSB	0x5f	word64 LSB	@segrel(S + A)
R_IA_64_SECREL32MSB	0x64	word32 MSB	@secrel(S + A)
R_IA_64_SECREL32LSB	0x65	word32 LSB	@secrel(S + A)
R_IA_64_SECREL64MSB	0x66	word64 MSB	@secrel(S + A)
R_IA_64_SECREL64LSB	0x67	word64 LSB	@secrel(S + A)
R_IA_64_REL32MSB	0x6c	word32 MSB	BD + A
R_IA_64_REL32LSB	0x6d	word32 LSB	BD + A
R_IA_64_REL64MSB	0x6e	word64 MSB	BD + A

Table 4-7. IA-64 Relocation Types (Continued)

Name	Value	Field	Calculation
`R_IA_64_REL64LSB`	`0x6f`	word64 LSB	BD + A
`R_IA_64_LTV32MSB`	`0x74`	word32 MSB	S + A (see below)
`R_IA_64_LTV32LSB`	`0x75`	word32 LSB	S + A (see below)
`R_IA_64_LTV64MSB`	`0x76`	word64 MSB	S + A (see below)
`R_IA_64_LTV64LSB`	`0x77`	word64 LSB	S + A (see below)
`R_IA_64_IPLTMSB`	`0x80`	function descriptor MSB	see below
`R_IA_64_IPLTLSB`	`0x81`	function descriptor LSB	see below
`R_IA_64_SUB`	`0x85`	Instruction-imm64	A – S
`R_IA_64_LTOFF22X`	`0x86`	instruction - immediate22	see below
`R_IA_64_LDXMOV`	`0x87`	instruction - immediate22	see below

Note: Values above 0xe0 are available for use in implementation-defined ways. All other values are reserved for future use.

The relocation type values have been chosen so that the expression type can be easily extracted by masking off the lower three or four bits, and the data/instruction format can be determined in most cases by looking only at the low-order four bits.

R_IA_64_LTV32MSB, R_IA_64_LTV32LSB, R_IA_64_LTV32MSB and R_IA_64_LTV32LSB
These relocations appear only in relocatable objects. They behave identically to the `R_IA_64_DIR*` family of relocations, with one exception: while it is expected that the addresses created will need further relocation at run-time, the linker should not create a corresponding relocation in the output executable or shared object file. The run-time consumer of the information provided is expected to relocate these values.

R_IA_64_IPLTMSB and R_IA_64_IPLTLSB
These relocations appear only in dynamic executables and shared objects. When used with the shorter form of relocation entry (`Elf32_Rel` or `Elf64_Rel`), they instruct the dynamic linker to initialize the corresponding function descriptor entry with the address of the referenced function and the value of the global pointer (gp) for the object containing the function's definition. When used with the longer form of relocation entry containing an explicit addend (`Elf32_Rela` or `Elf64_Rela`), the addend is additionally added to the address of the referenced function. See for more information.

R_IA_64_LTOFF22X and R_IA_64_LDXMOV
These relocations are used to support link-time rewriting of the indirect addressing code sequences. The `R_IA_64_LTOFF22X` relocation is used on the `addl` instruction that computes the address of a linkage table entry in place of the normal `R_IA_64_LTOFF22` relocation. It has exactly the same semantics as `R_IA_64_LTOFF22` unless the linker determined that the symbol could be addressed directly, in which case the linker transforms this into an `R_IA_64_GPREL22` relocation. An ABI-conforming implementation must recognize this relocation, but may choose to treat it as a synonym for `R_IA_64_LTOFF22`. The `R_IA_64_LDXMOV` relocation is used on an `ld8` instruction, where no

relocation would ordinarily be seen. The `ld8` instruction normally extracts the address of the referenced object from the linkage table by dereferencing the address conputed by the `addl`. Its symbol and addend fields must match exactly those of a corresponding `R_IA_LTOFF22X` relocation. If the linker determines that the symbol can be addressed directly, it rewrites the `ld8` as a `mov`. This can be done by masking out all but the qp, r1, and r3 fields of the instruction, then or'ing in the bit pattern `0x8000000000`. If an ABI-conforming implementation is choosing to treat `R_IA_64_LTOFF22X` as a synonym for `R_IA_64_LTOFF22`, this relocation is ignored.

intel

Program Loading and Dynamic Linking 5

5.1 Program Header

The IA-64 architecture defines two processor-specific values to be used in the `p_type` member of the program header.

Table 5-1. Program Header Types, `p_type`

Name	Value
PT_IA_64_ARCHEXT	0x70000000
PT_IA_64_UNWIND	0x70000001

PT_IA_64_ARCHEXT   The segment contains a section of type `SHT_IA_64_EXT` as described in Section 4.2, "Sections" on page 4-3. If this entry is present, it must precede all entries of type `PT_LOAD`.

PT_IA_64_UNWIND   The segment contains the stack unwind tables. See *Conventions* and Section 4.2, "Sections" on page 4-3 for details.

The IA-64 architecture defines one processor-specific value to be used in the `p_flags` member of the program header.

Table 5-2. Program Header Flags, `p_flags`

Name	Value
PF_IA_64_NORECOV	0x80000000

PF_IA_64_NORECOV   If this bit is set, the segment contains code that uses speculative instructions without recovery code. Executbles with this flag bit set are not ABI conforming.

5.2 Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When–and if–the system physically reads the file depends on the program's execution behavior, system load, and so on. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages un-referenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

The preferred page size for virtual memory management purposes for an IA-64 64-bit segment is contained in the `p_align` field of the program header entry describing that segment. The `p_align` field must contain 4 KB (`0x1000`) or a page size as defined in Section 7 of the *IA-64*

Processor Programmer's Reference Manual. Virtual addresses and file offsets for IA-64 64–bit segments are congruent modulo either the value contained in the `p_align` field or 4KB (`0x1000`), whichever is larger.

The following examples show a 64k alignment; virtual addresses and file offsets for segments are congruent modulo 64k (`0x10000`).

Figure 5-1. Example Executable File

File Offset	File	Virtual Address
0	ELF header	
	Program header table	
	Other information	
0x110	Text segment ... 0x4af630 bytes	0x4000000000000110 0x40000000004af73f
0x4af740	Data segment ... 0x16768 bytes	0x600000000000f740 0x6000000000025ea7
0x4c5ea0	Other information ...	

Figure 5-2. Example Program Header Segments

Member	Text	Data
`p_type`	PT_LOAD	PT_LOAD
`p_offset`	0x110	0x4af740
`p_vaddr`	0x4000000000000110	0x600000000000f740
`p_paddr`	unspecified	unspecified
`p_filesz`	0x4af630	0x16768
`p_memsz`	0x4af630	0x46b90
`p_flags`	PF_R+PF_X	PF_R+PF_W+PF_X
`p_align`	0x10000	0x10000

Although the example's file offsets and virtual addresses are congruent modulo 64KB for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.
- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

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

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. "Impurities" in the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 64KB (0x10000) pages.

Figure 5-3. Example Process Image Segments

Address	Contents	Segment
	Address	
0x4000000000000000	*Header padding* 0x110 bytes	
0x4000000000000110	Text segment ... 0x4af630 bytes	Text
0x40000000004af740	*Data padding* 0x8c0 bytes	
0x6000000000000000	*Text padding* 0xf740 bytes	
0x600000000000f740	Data segment ... 0x16768 bytes	Data
0x6000000000025ea8	Uninitialized data 0x30428 zero bytes	
0x60000000000562d0	*Page padding* 0x9d30 zero bytes	

On the IA-64 architecture, both executable and shared object segments contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior. Furthermore, there is no assumption that the individual segments for a given executable or shared object are fixed relatively in relation to one another. For example, the system might load all read-only segments for a process in one range of memory addresses and all read-write segments in a different range of addresses. Therefore, while the addresses shown in the example in Figures 5-3, 5-4 and 5-5 show the data segment for an executable immediately following the text segment, there is no requirement that it does so. The addresses assigned for each segment by the link editor, however, must not overlap.

Because dynamically linked IA-64 64-bit executable files are position-independent, the `exec` routines may choose to load such files at different addresses than those specified in the file's program header. The dynamic linker must be prepared to deal with this possibility.

5.2.1 Link-Time and Run-Time Addresses

Virtual addresses assigned by the linker when creating an executable or shared object file are known as link-time virtual addresses. Since position-independent executables and shared objects may be loaded at different addresses than those assigned by the linker, run-time virtual addresses differ from link-time virtual address by a constant value. Since there is no fixed address relationship at run-time among segments created at link-time, the constant value must be calculated based on the segment containing the address in question. The constant is the difference between the address at which the containing segment was loaded and the address assigned for that segment by the linker. The following table illustrates the calculation for an example text object.

Table 5-3. Example Run-Time Address Calculation

Value or Calculation	Result
Address as determined by link editor	`0x40000000000532f0`
Segment address contained in program header	`0x4000000000000110`
Base address of segment in file	`0x4000000000000000`
Base address of segment in process	`0x4c80000000000000`
Run-time minus link-time base address	`0x0c80000000000000`
Address of object in process	`0x4c800000000532f0`

5.2.2 Initializations

As the implementation constructs the new process, it is responsible for a number of initialization actions. Some of these have been described in Section 3.3.5, "Process Startup" on page 3-6. In addition to those steps, the implementation must:

1. Ensure the process environment has been properly initialized .

2. The global variable `_environ` must be initialized to point to the environment, before the initialization routines are executed. The execution of the initialization routines may result in the modification of `_environ`.

3. Pre-initializations routines in the executable, described in "Dynamic Linking" in Chapter 5 of the *System V ABI*, must be called, according to standard calling conventions.

4. Initialization routines, described in""Dynamic Linking" in Chapter 5 of the *System V* ABI and in the following section, in the executable and in all loaded shared objects must be called, according to standard calling conventions. The only order specified is that, for every library dependency "A depends on B", the initialization routines for B must be called before those for A.

5.3 Dynamic Linking

5.3.1 Dynamic Linker

When building an executable file that uses dynamic linking, the link editor adds a program header element of type PT_INTERP to an executable file, telling the system to invoke the dynamic linker as the program interpreter. The location of the dynamic linker, to be recorded on the PT_INTERP string, varies depending on the code model, architecture and byte order.

Table 5-4. Dynamic Linker Location

Architecture	Code Model	Byte Order	Dynamic Linker Name
IA-64	ILP32	Little-Endian	`/usr/lib/ia64l32/ld.so.1`
IA-64	ILP32	Big-Endian	`/usr/lib/ia64b32/ld.so.1`
IA-64	LP64	Little-Endian	`/usr/lib/ia64l64/ld.so.1`
IA-64	LP64	Big-Endian	`/usr/lib/ia64b64/ld.so.1`

5.3.2   Dynamic Section

All dynamic section entries containing addresses (entries that use the `d_ptr` member) contain link-time virtual addresses, as described above. The dynamic linker must relocate these addresses based on the difference between the link-time and run-time addresses of the segments referenced by the `d_ptr` member.

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

DT_PLTGOT   On the IA-64 architecture, this entry's `d_ptr` member gives the address contained in the global pointer (gp) for the object.

The IA-64 architecture defines one processor-specific dynamic section tag value.

Table 5-5. Dynamic Section Tag, `d_tag`

Name	Value
`DT_IA_64_PLT_RESERVE`	`0x70000000`

DT_IA_64_PLT_RESERVE

This element's `d_ptr` member contains the address of the first of three 8-byte words in the short data segment reserved for use by the dynamic linker. The three words are contiguous, with the second and third words growing toward higher addresses.

5.3.3   Shared Object Dependencies

The *System V ABI* describes, in "Shared Object Dependencies" in Chapter 5, the mechanism by which the dynamic linker locates shared object files and attaches them to a process image. When implemented on IA-64, the *ABI* supports a variety of code models, and since mixing models is not allowed, the dynamic linker must be able to locate shared object files that match the model of an executable program which has shared object dependencies. When applying the algorithm in the *System V ABI*, the dynamic linker will treat the following locations as the "default directory" location:

Table 5-6. Default Shared Object Location

Architecture	Code Model	Byte Order	Shared Object Location

NOTE: The standard location `/usr/lib` is reserved to the IA-32 ABI.

Table 5-6. Default Shared Object Location

IA-64	ILP32	Little-Endian	`/usr/lib/ia64l32`
IA-64	ILP32	Big-Endian	`/usr/lib/ia64b32`
IA-64	LP64	Little-Endian	`/usr/lib/ia64l64`
IA-64	LP64	Big-Endian	`/usr/lib/ia64b64`

NOTE: The standard location `/usr/lib` is reserved to the IA-32 ABI.

5.3.4   Global Offset Table

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

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

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

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

5.3.5   Function Addresses

On the IA-64 architecture, when one function calls another it is the caller's responsibility to reset the global pointer (gp) to the correct value for the object containing the called function. Thus, to call a function a caller needs two pieces of information: the address of the function and the value its global pointer should have. These two pieces of information are contained in a structure known as a *function descriptor* (see *Conventions*). So that a function pointer may be passed from function to function and still retain enough information to enable the function to be called, a function pointer is defined to be a pointer to the function descriptor for that function.

Each executable or shared object can have its own copy of the function descriptor entry for any function it calls to make access to function descriptors more efficient. But, when any shared object or the executable needs to reference the address of a function, each such reference must always retrieve the same address or comparisons of function pointers will not be predictable. Thus, there must be a unique function descriptor entry that can be referenced whenever the address of a function is taken. This entry is known as the "official" function descriptor for a function. The "official" function descriptor for any function is created and initialized by the dynamic linker as

int₂l

needed in response to `R_IA_64_FPTR32MSB`, `R_IA_64_FPTR32LSB`, `R_IA_64_FPTR64MSB` and `R_IA_64_FPTR64LSB` relocations (see Section 4.3, "Relocations" on page 4-6).

5.3.6   Procedure Linkage Table

The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. So that function addresses can be assigned dynamically at run-time without compromising the position-independence and sharability of a program's text, function addresses must be kept in private data and retrieved at the time a function is called. On the IA-64 architecture, the function addresses are kept in local function descriptor entries. Each entry is a pair containing the address of the referenced function and the value of the global pointer (gp) for the object containing the function's definition. The dynamic linker determines the destinations' absolute addresses and global pointer value and modifies the function descriptor's memory accordingly.

The function address and global pointer values are retrieved from the local function descriptor by a portion of code known as an *import stub*. The import stub may be compiled inline at the point of call by the compiler, or it may be placed in the *procedure linkage table*. The procedure linkage table is contained in an object's read-only text. Each function called directly by the object, but external to the object, will have a local function desciptor.

The dynamic linker is allowed to implement *lazy binding*, where each local function descriptor is not bound until the first call using that function descriptor. Instead, the initial value of the function address field of each function descriptor is initialized by the link editor to the address of a secondary PLT entry that is unique to the function being called. The secondary PLT entry must transfer control to the dynamic linker's lazy binding entry point, which will then resolve the reference, update the local function descriptor, and complete the call.

In order for the implementation to perform lazy binding correctly, the application must conform to the following conventions for transfer of control to the dynamic linker's lazy binding entry point:

1. The link editor must allocate a PLT Reserve area, consisting of three contiguous doublewords in the object's data segment. The DT_IA_64_PLT_RESERVE dynamic section entry must identify the first of these three doublewords. These words are initialized by the dynamic linker at program startup.

2. The relocation index for the function being called must be placed into GR 15, so that the dynamic linker can identify the target of the call. This value is an index into the portion of the dynamic relocation table addressed by the DT_JMPREL dynamic section entry. The designated relocation entry will have type R_IA_64_IPLTMSB or R_IA_64_IPLTLSB, and its offset will specify the local function descriptor entry referenced by the call.

3. An 8-byte identifier unique to the calling module must be placed into GR 16, so that the dynamic linker can identify the object from which the call originated, and thereby locate that object's relocation table. This identifier is found in the first double-word of the PLT Reserve area.

4. The gp register must be set to the dynamic linker's own gp value. This value is found in the second double-word of the PLT Reserve area.

5. The dynamic linker's lazy binding entry point is found in the third double-word of the PLT Reserve area.

Note that, by the time control is transferred to the secondary PLT entry, the gp value cannot be trusted, since the gp field of the local function descriptor is not initialized until the function is bound. Therefore, the import stub must copy the gp value to a scratch register before loading the gp value from the function descriptor, so that the secondary PLT entry may recover the original value in order to locate the PLT Reserve area.

The link editor must create import stubs, secondary PLT entries, and allocate local function descriptors for any direct call that cannot be statically bound within the same object (including calls where a definition is present, but is not protected against pre-emption). If an import stub is inlined by the compiler, the linker must still allocate the local function descriptor in response to the R_IA_64_PLTOFF relocation, and a secondary PLT entry to which the local function descriptor should point initially.

The LD_BIND_NOW environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type R_IA_64_IPLTMSB and R_IA_64_IPLTLSB during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

Note: Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.

The following example shows a recommended implementation of these conventions.

Figure 5-4. Procedure Linkage Table Sample Entries

```
.PLT0:  (initial special reserved entry)
        mov     r2 = r14  ;;
        addl    r14 = @gprel(plt_reserve), r2  ;;
        ld8     r16 = [r14], 8  ;;
        ld8     r17 = [r14], 8  ;;
        ld8     gp = [r14]
        mov     b6 = r17
        br      b6
.PLT1:  (entry for symbol name1)
        addl    r15 = @pltoff(name1), gp  ;;
        ld8     r16 = [r15], 8
        mov     r14 = gp  ;;
        ld8     gp = [r15]
        mov     b6 = r16
        br      b6
.PLT1a: mov     r15 = reloc_index
        br      .PLT0
```

Following the steps below, the dynamic linker and the program "cooperate" to resolve symbolic references through the procedure linkage table and the global offset table.

5.3.7   Initialization and Termination Functions

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

intel.

Libraries 6

For future use.

intel®

Miscellaneous   **7**

7.1   Introduction

This chapter contains miscellaneous subjects which are agreed to need representation somewhere, but are not strictly issues for a binary standard. The intent here is to provide this chapter as a "place holder" rather than as the intended final destination for these issues.

7.2   Development Environment

To facilitate portability of source code, a compilation environment that is capable of producing ABI conforming objects will provide the following information available at compilation time.

7.2.1   Pre-Defined Preprocessor Symbols

__ia64	Describes the target architecture. The initial value is 1. This value should track future backward-compatible architectural extensions in the `EF_IA_64_ARCH` ELF header flags field.
_ILP32	32-bit ABI data model: int, long, and pointer are 32 bits, long long is 64 bits. Value if defined is 1.
_LP64	64-bit ABI data model: long, long long, and pointer are 64 bits, int is 32 bits. Value if defined is 1.

7.2.2   Pre-Defined Preprocessor Assertions

A compilation environment that is capable of producing ABI conforming objects will implement the C preprocessor assertion feature. This allows a preprocessor *assertion* of the form:

```
#assert predicate[(token-sequence)]
```

This assertion associates `token-sequence` with `predicate` in the assertion name space. All tokens involved are preprocessor tokens: the predicate must be an identifier token, and the `token-sequence` is an arbitrary sequence of tokens. The (`token`-sequence) may be omitted from the `#assert`, in which case it associates no token sequence with `predicate`, but may be useful to place `predicate` in the assertion name space in order to avert possible warning messages for testing unrecognized predicates.

Predicate assertion associations may then be tested with:

```
#if #predicate(token-sequence)
```

This assertion evaluates true if `token-sequence` is associated with `predicate` and false otherwise. The token-sequence must be non-empty in a predicate test.

Multiple token sequences may be associated with a single predicate identifier by using multiple assertions. Each association may be tested independently.

In addition to `#assert` definition of assertion associations, compilers generally support the equivalent command-line option:

```
-Apredicate(token-sequence)
```

A compilation environment capable of producing ABI-conforming objects will provide the following pre-defined preprocessor assertions:

machine(ia64)   Target architecture.

model(lp64)   64-bit ABI data model: long, long long, and pointer are 64 bits, int is 32 bits.

model(ilp32)   32-bit ABI data model: int, long, and pointer are 32 bits, long long is 64 bits.

endian(little)   Little-endian data model.

endian(big)   Big-endian data model.

7.2.3   Compiler Pragmas

A compilation environment that is capable of producing ABI conforming objects will support a pragma to control section attribute specification for variables:

```
// define a symbol in a section with "short" or "long" attributes.
#pragma alloc_section(symbol_name, "attribute-list")
```

"*attribute-list*" is a comma-separated list of attributes, the defined values are:

"short"
"long"

Examples:

```
#pragma alloc_section(var1, "short")
int var1 = 20;
#pragma alloc_section(var2, "short")
extern int var2;
```

It is left to the compiler to decide whether the symbol should go to a "data" or "bss" or "rdata" section.

7.3   ILP32 ABI

Note: The following section is included for comment. There is not agreement that either an ILP32 ABI is mandatory nor that the mechanisms described in this section are the only way to implement an ILP32 ABI. Some vendors are known not to intend to implement an ILP32 ABI at all and at least one plans a different implementation. Thus this section presents guidelines for a possible implementation which would have some commonality but ILP32 binaries are not ABI conforming.

This description along with the *Conventions* document describes the software conventions needed to support IA-64 programs which will run in 32 bit address space. The Intel 64-bit architecture (IA-64) is composed of today's 32-bit Intel Architecture (IA-32) along with the 64-bit Instruction Set Architecture (ISA). For Unix, the base IA-32 software conventions are contained in the *i386^(tm) Processor Application Binary Interface.* These 32 bit conventions here describe a data model which is completely compatible with the appropriate IA-32 conventions on UNIX.

The 64-bit runtime architecture along with the *32-bit Conventions* defines most of the conventions necessary to compile, link, and execute a program on an operating system that supports these conventions. Its purpose is to ensure that object modules produced by different compilers can be linked together into a single application, and to specify the interfaces between compilers and linker, and between linker and operating system.

7.3.1 Objectives of the 32-bit Little-endian Runtime Architecture

This document defines the software interfaces needed to ensure that software for IA-64 will operate correctly together. The intent is to define as small a set of interface specifications as possible, while still meeting the following goals:

- High performance
- Ease of porting, IA-32 data compatibility
- Commonality with IA-64 64 bit software conventions
- Ease of implementation and use

We would like to provide complete enough interfaces between the different software products that they can be provided by different ISVs and still work together. These include compilers, linkers, applications, and dynamic link libraries. The goal is to have one convention, so software will be portable on IA-64 Unix systems.

7.3.2 Changes from the 64-bit Software Conventions

In *32-bit Conventions* the data representations are identical to the existing IA-32 conventions.

In other words all sizes and alignments of data items match existing IA-32 conventions. Integer, pointer and long types are each 4 bytes in size in ILP32 conventions. ILP32 function descriptors are 2 4-byte words. Global offset table entries are 4 bytes each.

```
sizeof(long) = sizeof(int) = sizeof((void *))= 4.
```

Long long, doubles and double-extended are aligned on 0 mod 4 boundaries.

Alignment for the members of an aggregate match existing IA-32 conventions.

7.3.3 Addressing and Protection

The features of the processor architecture that are described in the Addressing and Protection section of the PRM are intended for the exclusive use of the operating system software, with the following exceptions:

- An application may use the `zxt4` instructions to convert a 32-bit virtual address to a 64-bit virtual address.
- Refer to Chapter 2, Section 2.4 Addressing and protection of *Conventions,* for other exceptions.

7.3.4 Data Allocation

7.3.4.1 Global Variables

Common blocks, dynamically allocated regions (such as `malloc`, etc.), and external data items greater than 4 bytes must all be aligned at least on a 4-byte boundary. Smaller data items must be aligned on the next larger power-of-two boundary.

7.3.5 Local Memory Stack Variables

Stack frames must always be aligned on a 16-byte boundary. That is, the stack pointer register must always be aligned on a 16-byte boundary.

7.3.6 Parameter Passing

Parameter passing and allocation of parameter slots are done as described in Chapter 8, Section 8.5 of *Conventions*. Each slot size remains 64 bits in ILP32 conventions to match the 64 bit calling conventions for IA-64.