intel®

Intel® Itanium™ Processor-specific Application Binary Interface (ABI)

May 2001

Document Number: 245370-003

Contents

Figures

Tables

Introduction 1

1.1 The Intel® Itanium™ 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 Itanium™ 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® Itanium™ architecture microprocessors.

1.2 How to Use the System V ABI for Intel® Itanium™ Processors

The Itanium architecture supports a 64 bit instruction set and also provides compatibility with the IA-32 instruction set. Binaries using the Itanium architecture 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 Itanium 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 *Intel® IA-64 Architecture Software Developer's Manual*, *Itanium™ Software Conventions and Runtime Architecture Guide*, and *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 available at developer.intel.com web site (http://developer.intel.com/design/ia-64/devinfo.htm) are included by reference into this specification:

- *Intel® IA-64 Architecture Software Developer's Manual*, Vol. 1 Rev. 1.1: *IA-64 Application Architecture*
- *Intel ®IA-64 Architecture Software Developer's Manual*, Vol. 2 Rev. 1.1: *IA-64 System Architecture*
- *Intel® IA-64 Architecture Software Developer's Manual*, Vol. 3 Rev. 1.1: *Instruction Set Reference*
- *Intel® IA-64 Architecture Software Developer's Manual*, Vol. 4 Rev. 1.1: *Itanium™ Processor Programmer's Guide*
- *Intel® IA-64 Architecture Software Developer's Manual Specification Update*
- *Itanium™ Software Conventions and Runtime Architecture Guide (Document Number 245358)*
- *IA-64 Assembly Language Reference Guide (Document Number 248801)*

Software Installation 2

For future use.

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 *Itanium™ Software Conventions and Runtime Architecture Guide (“Conventions”)*, which is operating system-independent. Only information that is specific to implementing the ABI on the Itanium 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

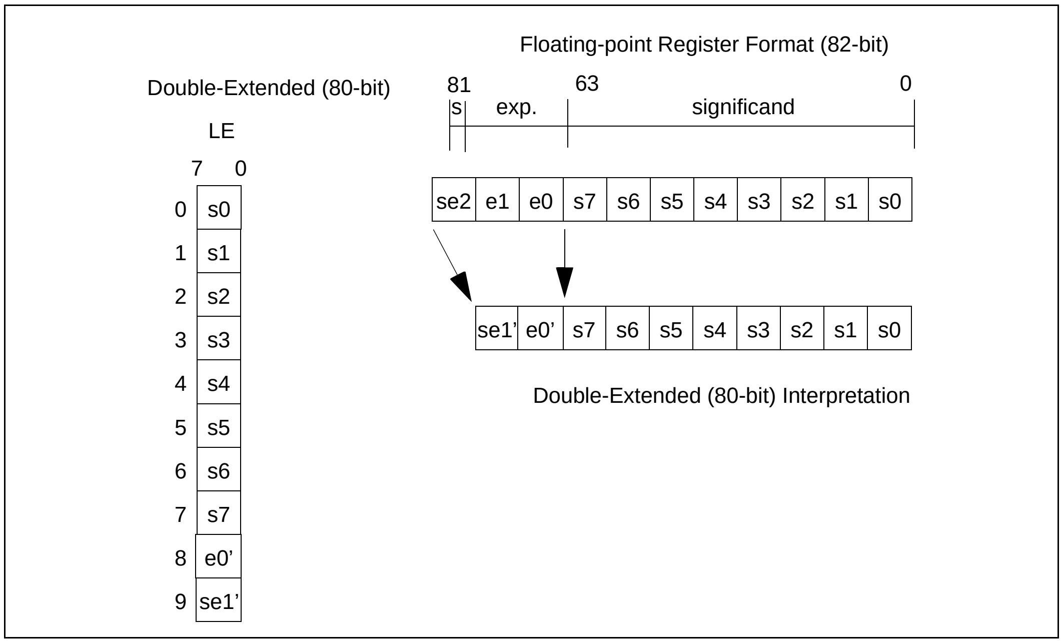
ILP32	`long long`
	`unsigned long long`
		8	4	Signed doubleword
				Unsigned doubleword

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


Floating-point Register Format (82-bit)
Double-Extended (80-bit)
81
63
0
s
exp.
significand
LE
7
0
0 s0
1 s1
2 s2
3 s3
4 s4
5 s5
6 s6
7 s7
8 e0'
9 se1'
se2 e1 e0 s7 s6 s5 s4 s3 s2 s1 s0
se1' e0' s7 s6 s5 s4 s3 s2 s1 s0
Double-Extended (80-bit) Interpretation

3.3 Operating System Interface

3.3.1 Exception Interface

As the Itanium 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. Table 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`	a
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`	b
Unaligned data	`SIGBUS`	`BUS_ADRALN`	c
Floating-point exceptions	`SIGFPE`	see Table 3-3	
Illegal instructions	`SIGILL`	`ILL_ILLOPC`	

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

Type of Exception	si_signo	si_code	Notes
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`	d
Break 0x100000-0x1fffff (reserved)	undefined		

a. 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.
b. 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.
c. 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.
d. 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 Itanium architecture.

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 Itanium™ Architecture

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 Itanium architecture:

`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® Itanium™ Architecture and the System V ABI" on page 1-1, binaries using the Itanium architecture 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`. Itanium architecture files using the 32-bit programming model may not be combined with Itanium architecture files using the 64-bit programming model.

4.1.1.2 File Class

For Itanium architecture 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 LP64 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]`, Itanium architecture 64-bit objects can use either ELFDATA2MSB or ELFDATA2LSB. That is, Itanium architecture 64-bit ELF files may use either the big endian or little endian data encoding. Itanium architecture files using ELFDATA2MSB encoding may not be combined with Itanium architecture 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_NONE`	0	Reserved
`ELFOSABI_HPUX`	1	HP-UX
`ELFOSABI_NETBSD`	2	NetBSD
`ELFOSABI_LINUX`	3	Linux
“Unspecified”	4	[IA-32 GNU Mach/Hurd]
“Unspecified”	5	[86 Open common IA-32 ABI]
`ELFOSABI_SOLARIS`	6	Solaris
`ELFOSABI_MONTEREY`	7	AIX
`ELFOSABI_IRIX`	8	IRIX
`ELFOSABI_FREEBSD`	9	FreeBSD
`ELFOSABI_TRU64`	10	Compaq TRU64 UNIX
`ELFOSABI_MODESTO`	11	Novell Modesto
`ELFOSABI_OPENBSD`	12	Open BSD

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. Itanium™ 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 `f6-f11` for integer arithmetic. If the program does not perform explicit floating-point calculations, registers `f6-f11` 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 Itanium 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 Itanium 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_PRIORITY_INIT	0x79000000

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`.
SHT_IA_64_PRIORITY_INIT	The section contains priority initialization records, each of which is a pair consisting of an Elfxx_Word priority and an Elfxx_Addr function address. An implementation is not required to support this section type, beyond the gABI requirements for the handling of unrecognized section types (i.e. linking them into a contiguous section in the object file created by the static linker).

4.2.2 Section Attribute Flags

A section header `sh_flags` member holds 1-bit flags that describe the attributes of the section. The Itanium 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 Itanium 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

Table 4-5. Special Sections (Continued)

Name	Type	Attributes
`.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`
`.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 uninitialized 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 Itanium 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 Itanium 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

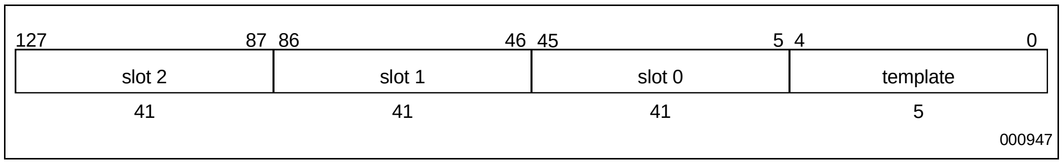


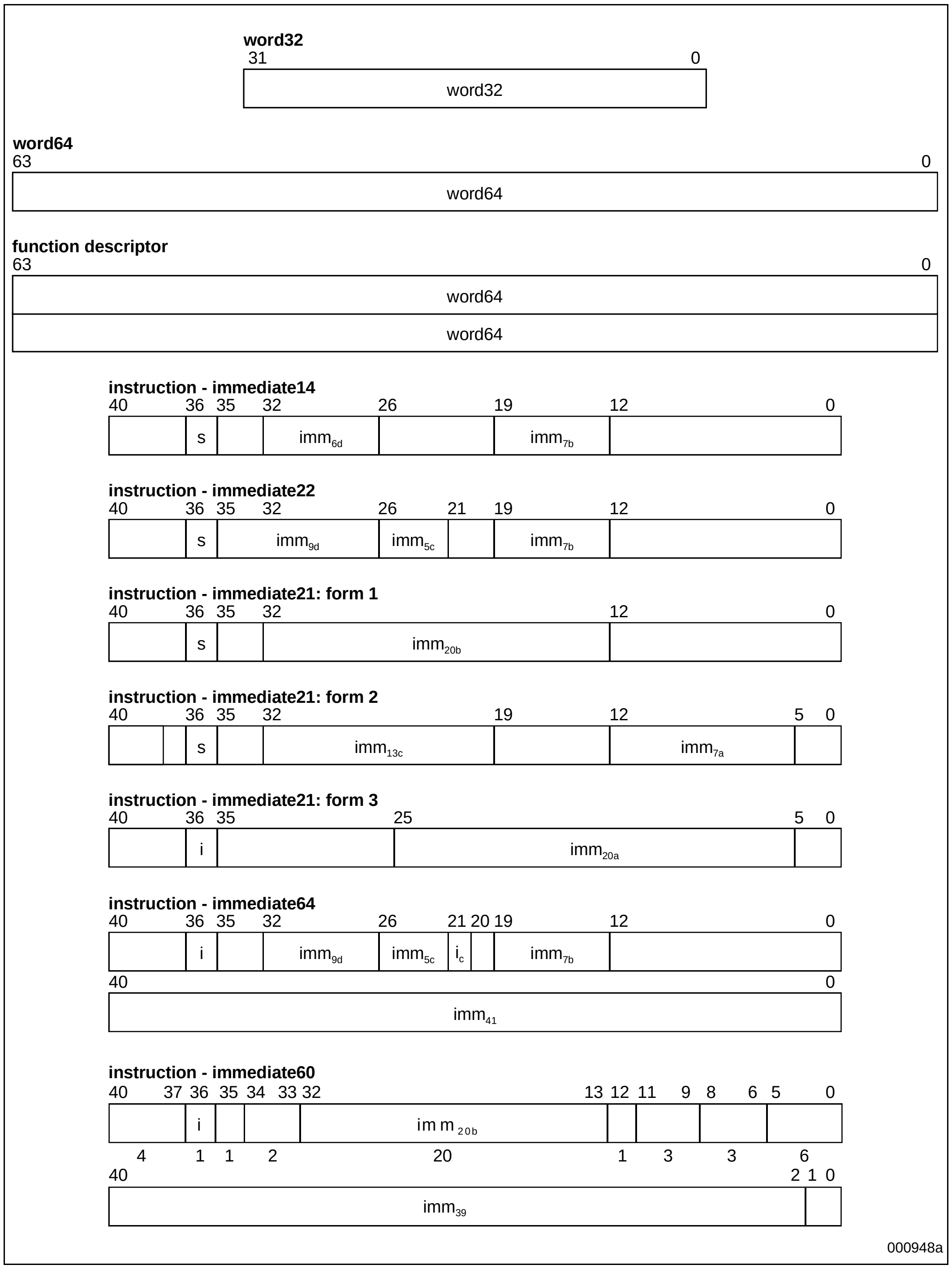
Table 4-6. Relocation Offset Instruction Slot Encoding

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).

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).

Figure 4-2. Relocatable Fields



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). `i` contains the high-order bit (sign bit).
instruction - immediate64	A 64-bit immediate value. The value is contained within two 41-bit instruction slots (slots 1 and 2 of a bundle). $\mathtt{imm}_{\mathtt{7b}}$ contains bits 0 through 6 (low order bits). $\mathtt{imm}_{\mathtt{9d}}$ contains bits 7 through 15. $\mathtt{imm}_{\mathtt{5c}}$ contains bits 16 through 20. $\mathtt{i}_{\mathtt{c}}$ contains bit 21. $\mathtt{imm}_{\mathtt{41}}$ contains bits 22 through 62 and takes the entire width of slot 1 (the second instruction slot). `i` contains bit 63.
instruction - immediate60	A 60-bit immediate value which is left shifted 4 bits to form 64-bit value for long branch or call. The value is contained within two 41-bit instruction slots (slots 1 and 2 of a bundle). $\mathtt{imm}_{\mathtt{20b}}$ contains bits 0 through 19 (low order bits). imm_{39} contains bits 20 through 58. `i` contains bit 59.

The calculations below assume one of two contexts:

1. 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 Itanium architecture 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 Itanium architecture.
2. 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. The relocation dealt with at run-time would be aligned.

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.
@tprel(expr)	Computes a tp-relative displacement -- the difference between the effective address and the value of the thread pointer. The expression must evaluate to an effective address within a thread-specific data segment.
@dtpmod(expr)	Computes the load module index corresponding to the load module that contains the definition of the symbol referenced by the relocation. When used in conjunction with the @ltoff() operator, it evaluates to the gp-relative offset of a linkage table entry that holds the computed load module index.
@dtprel(expr)	Computes a dtv-relative displacement -- the difference between the effective address and the base address of the thread-local storage block that contains the definition of the symbol referenced by the relocation. When used in conjunction with the @ltoff() operator, it evaluates to the gp-relative offset of a linkage table entry that holds the computed displacement.

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. Itanium™ Architecture 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

Table 4-7. Itanium™ Architecture Relocation Types (Continued)

Name	Value	Field	Calculation
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
R_IA_64_GPREL22	0x2a	instruction - immediate22	@gprel(S + A)
R_IA_64_GPREL64I	0x2b	instruction - immediate64	@gprel(S + A)
R_IA_64_GPREL32MSB	0x2c	word32 MSB	@gprel(S + A)
R_IA_64_GPREL32LSB	0x2d	word32 LSB	@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_PCREL60B	0x48	instruction - immediate60	S + A – P
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	word32 MSB	@ltoff(@ftpr(S + A))
R_IA_64_LTOFF_FPTR32LSB	0x55	word32 LSB	@ltoff(@fptr(S + A))
R_IA_64_LTOFF_FPTR64MSB	0x56	word64 MSB	@ltoff(@fptr(S + A))
R_IA_64_LTOFF_FPTR64LSB	0x57	word64 LSB	@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)

Table 4-7. Itanium™ Architecture Relocation Types (Continued)

Name	Value	Field	Calculation
`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
`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_PCREL21BI`[a]	`0x79`	instruction - immediate21 form 1	S + A - P
`R_IA_64_PCREL22`	`0x7A`	instruction - immediate22	S + A - P
`R_IA_64_PCREL64I`	`0x7B`	instruction - imm64	S + A - P
`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
`R_IA_64_TPREL14`	`0x91`	instruction - immediate14	@tprel(S+A)
`R_IA_64_TPREL22`	`0x92`	instruction - immediate22	@tprel(S+A)
`R_IA_64_TPREL64I`	`0x93`	instruction - immediate64	@tprel(S+A)
`R_IA_64_TPREL64MSB`	`0x96`	word64 MSB	@tprel(S+A)
`R_IA_64_TPREL64LSB`	`0x97`	word64 LSB	@tprel(S+A)
`R_IA_64_LTOFF_TPREL22`	`0x9A`	instruction - immediate22	@ltoff(@tprel(S+A))
`R_IA_64_DTPMOD64MSB`	`0xA6`	word64 MSB	@dtpmod(S+A)
`R_IA_64_DTPMOD64LSB`	`0xA7`	word64 LSB	@dtpmod(S+A)
`R_IA_64_LTOFF_DTPMOD22`	`0xAA`	instruction - immediate22	@ltoff(@dtpmod(S+A))
`R_IA_64_DTPREL14`	`0xB1`	instruction - immediate14	@dtprel(S+A)
`R_IA_64_DTPREL22`	`0xB2`	instruction - immediate22	@dtprel(S+A)
`R_IA_64_DTPREL64I`	`0xB3`	instruction - immediate64	@dtprel(S+A)
`R_IA_64_DTPREL32MSB`	`0xB4`	word632 MSB	@dtprel(S+A)
`R_IA_64_DTPREL32LSB`	`0xB5`	word32 LSB	@dtprel(S+A)
`R_IA_64_DTPREL64MSB`	`0xB6`	word64 MSB	@dtprel(S+A)
`R_IA_64_DTPREL64LSB`	`0xB7`	word64 LSB	@dtprel(S+A)
`R_IA_64_LTOFF_DTPREL22`	`0xBA`	instruction - immediate22	@ltoff(@dtprel(S+A))

a. The `PCREL21BI` relocation works just like `PCREL21B`, but it marks a call for which gp has not been saved, thus requiring that the target reside within the same load module as the call. It is needed it for the cases where we choose to bind a symbol locally, optimizing the call sequence, but where we don't want to, or can't, mark the symbol "protected" or "hidden."

Note: Relocation information not used at run-time may be unaligned. It is expected that linkers will have to deal with them. Relocations dealt at run-time will always be aligned.

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 are used only by the dynamic linker. Object files may contain these relocations. Static linkers should pass these along for the dynamic linker. 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 Section 5.3.6, "Procedure Linkage Table" on page 5-7 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 computed 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.

Program Loading and Dynamic Linking 5

5.1 Program Header

The Itanium 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 Itanium 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 unreferenced. 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 Itanium architecture 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 *Intel® IA-64 Architecture Programmer's Reference Manual*. Virtual addresses and file offsets for Itanium architecture 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

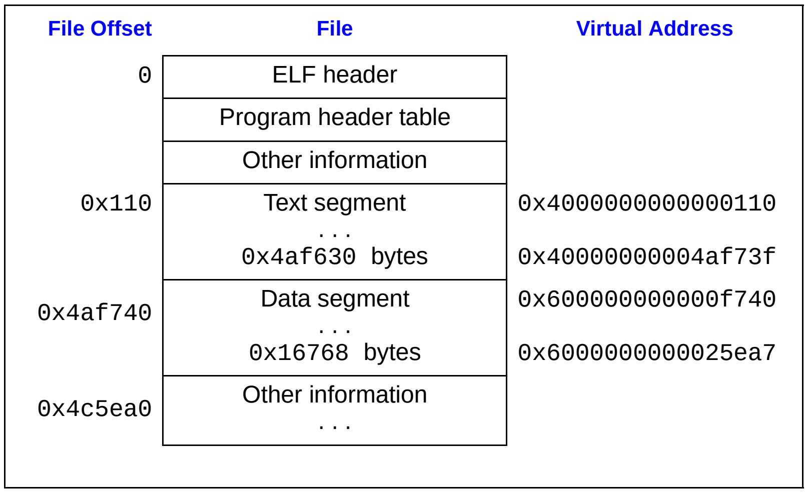


Figure 5-2. Example Program Header Segments

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
`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 Itanium 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 Itanium architecture 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 Linktime and Runtime 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, runtime virtual addresses differ from linktime virtual address by a constant value. Since there is no fixed address relationship at runtime among segments created at linktime, 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 Runtime 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`
Runtime 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
Itanium™ Architecture	ILP32	Little-Endian	`/usr/lib/ia64l32/ld.so.1`
	ILP32	Big-Endian	`/usr/lib/ia64b32/ld.so.1`
	LP64	Little-Endian	`/usr/lib/ia64l64/ld.so.1`
	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 runtime 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 Itanium architecture, this entry's `d_ptr` member gives the address contained in the global pointer (gp) for the object.

The Itanium 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 Itanium architecture, 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
Itanium™ Architecture	ILP32	Little-Endian	`/usr/lib/ia64l32`
	ILP32	Big-Endian	`/usr/lib/ia64b32`
	LP64	Little-Endian	`/usr/lib/ia64l64`
	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 Section 4.3, "Relocations" on page 4-6). 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 Itanium 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 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 runtime 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 Itanium 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.

1. When first creating the memory image of the program, the dynamic linker sets three reserved 8-byte words in each object's short data segment to special values. Steps below explain more about those values (see also the description for `DT_IA_64_PLT_RESERVE`, above).
2. For illustration, assume the program calls `name1`, transferring control to the label `.PLT1`.
3. The first instruction calculates the address of the local function descriptor entry for `name1` by adding its offset from gp to the value of gp. The address is saved in scratch register r15.
4. The third instruction saves the value of gp in scratch register r14.
5. The second and fourth instructions extract the information from the local function descriptor. The second instruction extracts the function address, storing its value in scratch register r16 while incrementing r15 by eight. The fourth instruction loads gp with the value stored in the local function descriptor. The link editor initializes the local function descriptor entry so that the function address contains the address of the mov instruction labeled .PLT1a. The procedure linkage table sets scratch branch register b6 to the address saved in r16 and branches to that address.
6. Consequently, the program saves a relocation index `reloc_index` in scratch register r15. The relocation index is a signed 22-bit immediate index into the portion of the 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 in the previous addl instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, name1 in this case.
7. After assigning the relocation index, the program then branches to .PLT0, the first entry in the procedure linkage table. The first five instructions in this entry de-reference the three special values reserved for the dynamic linker in the short data segment using the scratch register r14, which was set to the value of gp for the object calling name1. The first instruction saves r14 in scratch register r2. This allows the use of a 22-bit immediate value in the second instruction (the addl instruction can only be used with general registers r0, r1, r2 and r3). The second instruction adds to r2 the offset from the global pointer of the invoking object to the first of the three values set by the dynamic linker for that object. This value is stored back in r14. The third instruction stores the contents of the first reserved entry in scratch register r16, incrementing r14 by eight. This entry gives the dynamic linker an 8-byte word of identifying information. The fourth instruction extracts the second reserved entry, saving it in scratch register r17, while, again, incrementing r14 by eight. The second reserved entry is initialized by the dynamic linker to contain the address of a function binding routine within the dynamic linker itself. The fifth instruction sets the value of gp to the value contained in the third reserved entry. The dynamic linker sets this entry to contain the gp value for the object containing the dynamic linker, itself. The program then sets scratch branch register b6 to the address saved in r17 and branches to that address.
8. When the dynamic linker receives control, two scratch registers contain information it will use in relocating the function call: r15 contains the index of the relocation entry and r16 contains an 8-byte identifying word. The dynamic linker looks at the designated relocation entry, finds the symbol's value and the value of gp for the object containing the symbol, stores these values in the local function descriptor entry for name1, and transfers control to the desired destination.
9. Subsequent executions of the procedure linkage table entry will transfer directly to `name1` instead of to `.PLT0`, bypassing the call to the dynamic linker.

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.

The values contained in DT_INIT, DT_INIT_ARRAY, and DT_PREINIT_ARRAY are virtual address of functions within the shared object. It does not contain the address of the function descriptors.

Libraries 6

6.1 Unwind Library Interface

This section defines the Unwind Library interface, expected to be provided by any Itanium architecture psABI-compliant system. This is the interface on which the C++ ABI exception-handling facilities are built. We assume as a basis the unwind descriptor tables described in the base *Itanium™ Software Conventions and Runtime Architecture Guide*. The focus here will be on the APIs for accessing those structures.

It is intended that nothing in this section be specific to C++, though some parts are clearly intended to support C++ features.

The unwind library interface consists of at least the following routines:

```
_Unwind_RaiseException,
_Unwind_Resume,
_Unwind_DeleteException,
_Unwind_GetGR,
_Unwind_SetGR,
_Unwind_GetIP,
_Unwind_SetIP,
_Unwind_GetRegionStart,
_Unwind_GetLanguageSpecificData,
_Unwind_ForcedUnwind
```

In addition, two data types are defined (`_Unwind_Context` and `_Unwind_Exception`) to interface a calling runtime (such as the C++ runtime) and the above routines. All routines and interfaces behave as if defined `extern "C"`. In particular, the names are not mangled. All names defined as part of this interface have a "`_Unwind_`" prefix.

Lastly, a language and vendor specific personality routine will be stored by the compiler in the unwind descriptor for the stack frames requiring exception processing. The personality routine is called by the unwinder to handle language-specific tasks such as identifying the frame handling a particular exception.

6.1.1 Exception Handler Framework

6.1.1.1 Reasons for Unwinding

There are two major reasons for unwinding the stack:

- exceptions, as defined by languages that support them (such as C++)
- "forced" unwinding (such as caused by `longjmp` or thread termination).

The interface described here tries to keep both similar. There is a major difference, however.

- In the case where an exception is thrown, the stack is unwound while the exception propagates, but it is expected that the personality routine for each stack frame knows whether it wants to catch the exception or pass it through. This choice is thus delegated to the personality routine, which is expected to act properly for any type of exception, whether "native" or "foreign". Some guidelines for "acting properly" are given below.

- During "forced unwinding", on the other hand, an external agent is driving the unwinding. For instance, this can be the `longjmp` routine. This external agent, not each personality routine, knows when to stop unwinding. The fact that a personality routine is not given a choice about whether unwinding will proceed is indicated by the _UA_FORCE_UNWIND flag.

To accommodate these differences, two different routines are proposed.

`_Unwind_RaiseException` performs exception-style unwinding, under control of the personality routines. `_Unwind_ForcedUnwind`, on the other hand, performs unwinding, but gives an external agent the opportunity to intercept calls to the personality routine. This is done using a proxy personality routine, that intercepts calls to the personality routine, letting the external agent override the defaults of the stack frame's personality routine.

As a consequence, it is not necessary for each personality routine to know about any of the possible external agents that may cause an unwind. For instance, the C++ personality routine need deal only with C++ exceptions (and possibly disguising foreign exceptions), but it does not need to know anything specific about unwinding done on behalf of `longjmp` or pthreads cancellation.

6.1.1.2 The Unwind Process

The standard ABI exception handling / unwind process begins with the raising of an exception, in one of the forms mentioned above. This call specifies an exception object and an exception class.

The runtime framework then starts a two-phase process:

- In the *search* phase, the framework repeatedly calls the personality routine, with the `_UA_SEARCH_PHASE` flag as described below, first for the current `ip` and register state, and then unwinding a frame to a new `ip` at each step, until the personality routine reports either success (a handler found in the queried frame) or failure (no handler) in all frames. It does not actually restore the unwound state, and the personality routine must access the state through the API.
- If the search phase reports failure, e.g. because no handler was found, it will call `terminate()` rather than commence phase 2.

  If the search phase reports success, the framework restarts in the *cleanup* phase. Again, it repeatedly calls the personality routine, with the `_UA_CLEANUP_PHASE` flag as described below, first for the current `ip` and register state, and then unwinding a frame to a new `ip` at each step, until it gets to the frame with an identified handler. At that point, it restores the register state, and control is transferred to the user landing pad code.

Each of these two phases uses both the unwind library and the personality routines, since the validity of a given handler and the mechanism for transferring control to it are language-dependent, but the method of locating and restoring previous stack frames is language independent.

A two-phase exception-handling model is not strictly necessary to implement C++ language semantics, but it does provide some benefits. For example, the first phase allows an exception-handling mechanism to *dismiss* an exception before stack unwinding begins, which allows *resumptive* exception handling (correcting the exceptional condition and resuming execution at the point where it was raised). While C++ does not support resumptive exception handling, other languages do, and the two-phase model allows C++ to coexist with those languages on the stack.

Note that even with a two-phase model, we may execute each of the two phases more than once for a single exception, as if the exception was being thrown more than once. For instance, since it is not possible to determine if a given catch clause will rethrow or not without executing it, the exception propagation effectively stops at each catch clause, and if it needs to restart, restarts at phase 1. This process is not needed for destructors (cleanup code), so the phase 1 can safely process all destructor-only frames at once and stop at the next enclosing catch clause.

For example, if the first two frames unwound contain only cleanup code, and the third frame contains a C++ catch clause, the personality routine in phase 1 does not indicate that it found a handler for the first two frames. It must do so for the third frame, because it is unknown how the exception will propagate out of this third frame, e.g. by rethrowing the exception or throwing a new one in C++.

The API specified by the Itanium architecture psABI for implementing this framework is described in the following sections.

6.1.2 Data Structures

6.1.2.1 Reason Codes

The unwind interface uses reason codes in several contexts to identify the reasons for failures or other actions, defined as follows:

```
typedef enum {
    _URC_NO_REASON = 0,
    _URC_FOREIGN_EXCEPTION_CAUGHT = 1,
    _URC_FATAL_PHASE2_ERROR = 2,
    _URC_FATAL_PHASE1_ERROR = 3,
    _URC_NORMAL_STOP = 4,
    _URC_END_OF_STACK = 5,
    _URC_HANDLER_FOUND = 6,
    _URC_INSTALL_CONTEXT = 7,
    _URC_CONTINUE_UNWIND = 8
} _Unwind_Reason_Code;
```

The interpretation of these codes is described below.

6.1.2.2 Exception Header

The unwind interface uses a pointer to an exception header object as its representation of an exception being thrown. In general, the full representation of an exception object is language- and implementation-specific, but it will be prefixed by a header understood by the unwind interface, defined as follows:

```
typedef void (*_Unwind_Exception_Cleanup_Fn)
    (_Unwind_Reason_Code reason,
     struct _Unwind_Exception *exc);

struct _Unwind_Exception {
    uint64      exception_class;
    _Unwind_Exception_Cleanup_Fn exception_cleanup;
    uint64      private_1;
    uint64      private_2;
};
```

An `_Unwind_Exception` object must be double-word aligned. The first two fields are set by user code prior to raising the exception, and the latter two should never be touched except by the runtime.

The `exception_class` field is a language- and implementation-specific identifier of the kind of exception. It allows a personality routine to distinguish between native and foreign exceptions, for example.

The `exception_cleanup` routine is called whenever an exception object needs to be destroyed by a different runtime than the runtime which created the exception object, for instance if a Java exception is caught by a C++ *catch* handler. In such a case, a reason code (see above) indicates why the exception object needs to be deleted:

- `_URC_FOREIGN_EXCEPTION_CAUGHT` = 1: This indicates that a different runtime caught this exception. Nested foreign exceptions, or rethrowing a foreign exception, result in undefined behavior.
- `_URC_FATAL_PHASE1_ERROR` = 3: The personality routine encountered an error during phase 1, other than the specific error codes defined.
- `_URC_FATAL_PHASE2_ERROR` = 2: The personality routine encountered an error during phase 2, for instance a stack corruption.

Note: Normally, all errors should be reported during phase 1 by returning from `_Unwind_RaiseException`. However, landing pad code could cause stack corruption between phase 1 and phase 2. For a C++ exception, the runtime should call `terminate()` in that case.

The private unwinder state (`private_1` and `private_2`) in an exception object should be neither read by nor written to by personality routines or other parts of the language-specific runtime. It is used by the specific implementation of the unwinder on the host to store internal information, for instance to remember the final handler frame between unwinding phases.

In addition to the above information, a typical runtime such as the C++ runtime will add language-specific information used to process the exception. This is expected to be a contiguous area of memory after the `_Unwind_Exception` object, but this is not required as long as the matching personality routines know how to deal with it, and the `exception_cleanup` routine de-allocates it properly.

6.1.2.3 Unwind Context

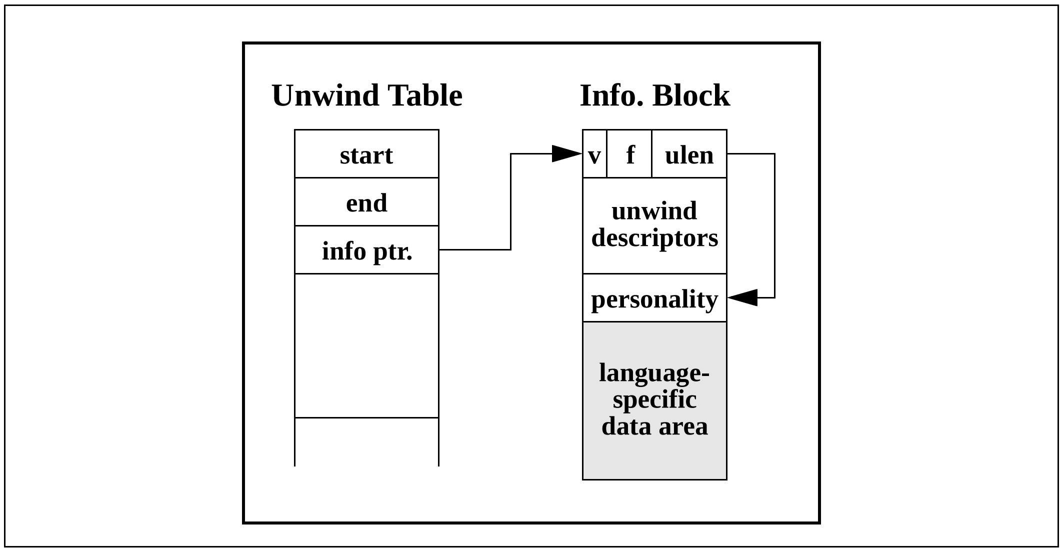
The `_Unwind_Context` type is an opaque type used to refer to a system-specific data structure used by the system unwinder. This context is created and destroyed by the system, and passed to the personality routine during unwinding.

```
struct _Unwind_Context
```

6.1.2.4 Personality Routine

As documented in Chapter 11 of the *Itanium™ Software Conventions and Runtime Architecture Guide*, the unwind tables consists of three fields as illustrated in Figure 6-1; each field is a 64-bit doubleword. The first two fields define the starting and ending addresses of the procedure, respectively, and the third field points to a variable-size information block containing the unwind descriptor list and language-specific data area. The ending address is the address of the first bundle beyond the end of the procedure. These values are all segment-relative offsets, not absolute addresses, so they do not require run-time relocations. The unwind table is sorted by the procedure start address. The shaded area in the figure represents the language-specific data area.

Figure 6-1. Unwind Table



The personality routine identifier is accessed by adding the size of the unwind descriptor area (*ulen),* which is a count of doublewords, not bytes), plus the size of the header doubleword, to the information block pointer. This identifier contains the 64-bit `gp`-relative offset of a doubleword in the linkage table that contains a function pointer, which in turn points to the function descriptor of the personality routine. The function pointer itself must be in the data segment because it may need relocation. The dispatcher should call this routine during the first unwind only if the EHANDLER bit is set, and during the second unwind only if the UHANDLER bit is set. The language-specific data immediately follows the personality routine identifier, so the address of this area must be made available to the personality routine.

6.1.3 Throwing an Exception

6.1.3.1 _Unwind_RaiseException

```
_Unwind_Reason_Code _Unwind_RaiseException
      ( struct _Unwind_Exception *exception_object );
```

Raise an exception, passing along the given exception object, which should have its `exception_class` and `exception_cleanup` fields set. The exception object has been allocated by the language-specific runtime, and has a language-specific format, except that it must contain an `_Unwind_Exception` struct (see *Exception Header* above). `_Unwind_RaiseException` does not return, unless an error condition is found (such as no handler for the exception, bad stack format, etc.). In such a case, an `_Unwind_Reason_Code` value is returned. Possibilities are:

- `_URC_END_OF_STACK`: The unwinder encountered the end of the stack during phase 1, without finding a handler. The unwind runtime will not have modified the stack. The C++ runtime will normally call `uncaught_exception()` in this case.
- `_URC_FATAL_PHASE1_ERROR`: The unwinder encountered an unexpected error during phase 1, e.g. stack corruption. The unwind runtime will not have modified the stack. The C++ runtime will normally call `terminate()` in this case.

If the unwinder encounters an unexpected error during phase 2, it should return _URC_FATAL_PHASE2_ERROR to its caller. In C++, this will usually be `_cxa_throw`, which will call `terminiate()`.

Note: The unwind runtime will likely have modified the stack (e.g. popped frames from it) or register context, or landing pad code may have corrupted them. As a result, the caller of `_Unwind_RaiseException` could make no assumptions about the state of its stack or registers.

6.1.3.2 _Unwind_ForcedUnwind

```
typedef _Unwind_Reason_Code (*_Unwind_Stop_Fn)
    (int version,
     _Unwind_Action actions,
     uint64 exceptionClass,
     struct _Unwind_Exception *exceptionObject,
     struct _Unwind_Context *context,
     void *stop_parameter );

_Unwind_Reason_Code _Unwind_ForcedUnwind
      ( struct _Unwind_Exception *exception_object,
    _Unwind_Stop_Fn stop,
    void *stop_parameter );
```

Raise an exception for forced unwinding, passing along the given exception object, which should have its `exception_class` and `exception_cleanup` fields set. The exception object has been allocated by the language-specific runtime, and has a language-specific format, except that it must contain an `_Unwind_Exception` struct (see *Exception Header* above).

Forced unwinding is a single-phase process (phase 2 of the normal exception-handling process). The `stop` and `stop_parameter` parameters control the termination of the unwind process, instead of the usual personality routine query. The `stop` function parameter is called for each unwind frame, with the parameters described for the usual personality routine below, plus an additional `stop_parameter`.

When the `stop` function identifies the destination frame, it transfers control (according to its own, unspecified, conventions) to the user code as appropriate without returning, normally after calling `_Unwind_DeleteException`. If not, it should return an `_Unwind_Reason_Code` value as follows:

- `_URC_NO_REASON`: This is not the destination frame. The unwind runtime will call the frame's personality routine with the `_UA_FORCE_UNWIND` and `_UA_CLEANUP_PHASE` flags set in `actions`, and then unwind to the next frame and call the `stop` function again.
- `_URC_END_OF_STACK`: In order to allow `_Unwind_ForcedUnwind` to perform special processing when it reaches the end of the stack, the unwind runtime will call it after the last frame is rejected, with a NULL stack pointer in the context, and the `stop` function must catch this condition (i.e. by noticing the NULL stack pointer). It may return this reason code if it cannot handle end-of-stack.
- `_URC_FATAL_PHASE2_ERROR`: The `stop` function may return this code for other fatal conditions, e.g. stack corruption.

If the `stop` function returns any reason code other than `_URC_NO_REASON`, the stack state is indeterminate from the point of view of the caller of `_Unwind_ForcedUnwind`. Rather than attempt to return, therefore, the unwind library should use the `exception_cleanup` entry in the exception, and then call `abort()`.

Note: Example: `longjmp_unwind()`

The expected implementation of `longjmp_unwind()` is as follows. The `setjmp()` routine will have saved the state to be restored in its customary place, including the frame pointer. The `longjmp_unwind()` routine will call `_Unwind_ForcedUnwind` with a `stop` function that compares the frame pointer in the context record with the saved frame pointer. If equal, it will restore the `setjmp()` state as customary, and otherwise it will return `_URC_NO_REASON` or `_URC_END_OF_STACK`.

Note: If a future requirement for two-phase forced unwinding were identified, an alternate routine could be defined to request it, and an `actions` parameter flag defined to support it.

6.1.3.3 _Unwind_Resume

```
void _Unwind_Resume (struct _Unwind_Exception *exception_object);
```

Resume propagation of an existing exception e.g. after executing cleanup code in a partially unwound stack. A call to this routine is inserted at the end of a landing pad that performed cleanup, but did not resume normal execution. It causes unwinding to proceed further.

Note: `_Unwind_Resume` should not be used to implement rethrowing. To the unwinding runtime, the catch code that rethrows was a handler, and the previous unwinding session was terminated before entering it. Rethrowing is implemented by calling `_Unwind_RaiseException` again with the same exception object.

Note: This is the only routine in the unwind library which is expected to be called directly by generated code: it will be called at the end of a landing pad in a “landing-pad” model.

6.1.4 Exception Object Management

6.1.4.1 _Unwind_DeleteException

```
void _Unwind_DeleteException
      (struct _Unwind_Exception *exception_object);
```

Deletes the given exception object. If a given runtime resumes normal execution after catching a foreign exception, it will not know how to delete that exception. Such an exception will be deleted by calling `_Unwind_DeleteException`. This is a convenience function that calls the function pointed to by the `exception_cleanup` field of the exception header.

6.1.5 Context Management

These functions are used for communicating information about the unwind context (i.e. the unwind descriptors and the user register state) between the unwind library and the personality routine and landing pad. They include routines to read or set the context record images of registers in the stack frame corresponding to a given unwind context, and to identify the location of the current unwind descriptors and unwind frame.

6.1.5.1 _Unwind_GetGR

```
uint64 _Unwind_GetGR
    (struct _Unwind_Context *context, int index);
```

This function returns the 64-bit value of the given general register. The register is identified by its index: 0 to 31 are for the fixed registers, and 32 to 127 are for the stacked registers.

During the two phases of unwinding, only GR1 has a guaranteed value, which is the Global Pointer (`gp`) of the frame referenced by the unwind context. If the register has its NAT bit set, the behavior is unspecified.

6.1.5.2 _Unwind_SetGR

```
void _Unwind_SetGR
  (struct _Unwind_Context *context,
   int index,
   uint64 new_value);
```

This function sets the 64-bit value of the given register, identified by its index as for `_Unwind_GetGR`. The NAT bit of the given register is reset.

The behavior is guaranteed only if the function is called during phase 2 of unwinding, and applied to an unwind context representing a handler frame, for which the personality routine will return `_URC_INSTALL_CONTEXT`. In that case, only registers GR15, GR16, GR17, GR18 should be used. These scratch registers are reserved for passing arguments between the personality routine and the landing pads.

6.1.5.3 _Unwind_GetIP

```
uint64 _Unwind_GetIP
    (struct _Unwind_Context *context);
```

This function returns the 64-bit value of the instruction pointer (`ip`).

During unwinding, the value is guaranteed to be the address of the bundle immediately following the call site in the function identified by the unwind context. This value may be outside of the procedure fragment for a function call that is known to not return (such as `_Unwind_Resume`).

6.1.5.4 _Unwind_SetIP

```
void _Unwind_SetIP
    (struct _Unwind_Context *context,
     uint64 new_value);
```

This function sets the value of the instruction pointer (`ip`) for the routine identified by the unwind context.

The behavior is guaranteed only when this function is called for an unwind context representing a handler frame, for which the personality routine will return `_URC_INSTALL_CONTEXT`. In this case, control will be transferred to the given address, which should be the address of a landing pad.

6.1.5.5 _Unwind_GetLanguageSpecificData

```
uint64 _Unwind_GetLanguageSpecificData
    (struct _Unwind_Context *context);
```

This routine returns the address of the language-specific data area for the current stack frame.

Note: This routine is not strictly required: it could be accessed through `_Unwind_GetIP` using the documented format of the `UnwindInfoBlock`, but since this work has been done for finding the personality routine in the first place, it makes sense to cache the result in the context. We could also pass it as an argument to the personality routine.

6.1.5.6 _Unwind_GetRegionStart

```
uint64 _Unwind_GetRegionStart
    (struct _Unwind_Context *context);
```

This routine returns the address of the beginning of the procedure or code fragment described by the current unwind descriptor block.

This information is required to access any data stored relative to the beginning of the procedure fragment. For instance, a call site table might be stored relative to the beginning of the procedure fragment that contains the calls. During unwinding, the function returns the start of the procedure fragment containing the call site in the current stack frame.

6.1.6 Personality Routine

```
_Unwind_Reason_Code (*__personality_routine)
    (int version,
     _Unwind_Action actions,
     uint64 exceptionClass,
     struct _Unwind_Exception *exceptionObject,
     struct _Unwind_Context *context);
```

The personality routine is the function in the C++ (or other language) runtime library which serves as an interface between the system unwind library and language-specific exception handling semantics. It is specific to the code fragment described by an unwind info block, and it is always referenced via the pointer in the unwind info block, and hence it has no psABI-specified name.

6.1.6.1 Parameters

The personality routine parameters are as follows:

version	Version number of the unwinding runtime, used to detect a mis-match between the unwinder conventions and the personality routine, or to provide backward compatibility. For the conventions described in this document, `version` will be 1.
actions	Indicates what processing the personality routine is expected to perform, as a bit mask. The possible actions are described below.
exceptionClass	An 8-byte identifier specifying the type of the thrown exception. By convention, the high 4 bytes indicate the vendor (for instance HP\0\0), and the low 4 bytes indicate the language. For the C++ ABI described in this document, the low four bytes are C++\0.

Note: This is not a null-terminated string. Some implementations may use no null bytes.

exceptionObject	The pointer to a memory location recording the necessary information for processing the exception according to the semantics of a given language (see the *Exception Header* section above).
context	Unwinder state information for use by the personality routine. This is an opaque handle used by the personality routine in particular to access the frame's registers (see the *Unwind Context* section above).
return value	The return value from the personality routine indicates how further unwind should happen, as well as possible error conditions. See the following section.

6.1.6.2 Personality Routine Actions

The `actions` argument to the personality routine is a bitwise `OR` of one or more of the following constants:

```
typedef int _Unwind_Action;
const _Unwind_Action _UA_SEARCH_PHASE = 1;
const _Unwind_Action _UA_CLEANUP_PHASE = 2;
const _Unwind_Action _UA_HANDLER_FRAME = 4;
const _Unwind_Action _UA_FORCE_UNWIND = 8;
```

_UA_SEARCH_PHASE	Indicates that the personality routine should check if the current frame contains a handler, and if so return `_URC_HANDLER_FOUND`, or otherwise return `_URC_CONTINUE_UNWIND`. `_UA_SEARCH_PHASE` cannot be set at the same time as `_UA_CLEANUP_PHASE`.
_UA_CLEANUP_PHASE	Indicates that the personality routine should perform cleanup for the current frame. The personality routine can perform this cleanup itself, by calling nested procedures, and return `_URC_CONTINUE_UNWIND`. Alternatively, it can setup the registers (including the `ip`) for transferring control to a "landing pad", and return `_URC_INSTALL_CONTEXT`.
_UA_HANDLER_FRAME	During phase 2, indicates to the personality routine that the current frame is the one which was flagged as the handler frame during phase 1. The personality routine is not allowed to change its mind between phase 1 and phase 2, i.e. it must handle the exception in this frame in phase 2.
_UA_FORCE_UNWIND	During phase 2, indicates that no language is allowed to "catch" the exception. This flag is set while unwinding the stack for `longjmp` or during thread cancellation. User-defined code in a catch clause may still be executed, but the catch clause must resume unwinding with a call to `_Unwind_Resume` when finished.

6.1.6.3 Transferring Control to a Landing Pad

If the personality routine determines that it should transfer control to a landing pad (in phase 2), it may set up registers (including `ip`) with suitable values for entering the landing pad (e.g. with landing pad parameters), by calling the context management routines above. It then returns `_URC_INSTALL_CONTEXT`.

Prior to executing code in the landing pad, the unwind library restores registers not altered by the personality routine, using the context record, to their state in that frame before the call that threw the exception, as follows. All registers specified as callee-saved by the base ABI are restored, as well as scratch registers `r15`, `r16`, `r17` and `r18` (see below). Except for those exceptions, scratch (or caller-saved) registers are not preserved, and their contents are undefined on transfer. The accessibility of registers in the frame will be restored to that at the point of call, i.e. the same logical registers will be accessible, but their mappings to physical registers may change. Further, the state of stacked registers beyond the current frame is unspecified, i.e. they may be either in physical registers or on the register stack.

The landing pad can either resume normal execution (as, for instance, at the end of a C++ catch), or resume unwinding by calling `_Unwind_Resume` and passing it the `exceptionObject` argument received by the personality routine. `_Unwind_Resume` will never return.

`_Unwind_Resume` should be called if and only if the personality routine did not return `_Unwind_HANDLER_FOUND` during phase 1. As a result, the unwinder can allocate resources (for instance memory) and keep track of them in the exception object reserved words. It should then free these resources before transferring control to the last (handler) landing pad. It does not need to free the resources before entering non-handler landing-pads, since `_Unwind_Resume` will ultimately be called.

The landing pad may receive arguments from the runtime, typically passed in registers set using `_Unwind_SetGR` by the personality routine. For a landing pad that can call to `_Unwind_Resume`, one argument must be the `exceptionObject` pointer, which must be preserved to be passed to `_Unwind_Resume`.

The landing pad may receive other arguments, for instance a *switch value* indicating the type of the exception. Four scratch registers are reserved for this use (`r15`, `r16`, `r17` and `r18`.)

6.1.6.4 Rules for Correct Inter-language Operation

The following rules must be observed for correct operation between languages and/or runtimes from different vendors:

An exception which has an unknown class must not be altered by the personality routine. The semantics of foreign exception processing depend on the language of the stack frame being unwound. This covers in particular how exceptions from a foreign language are mapped to the native language in that frame.

If a runtime resumes normal execution, and the caught exception was created by another runtime, it should call `_Unwind_DeleteException`. This is true even if it understands the exception object format (such as would be the case between different C++ runtimes).

A runtime is not allowed to catch an exception if the `_UA_FORCE_UNWIND` flag was passed to the personality routine.

Note: Example: Foreign exceptions in C++. In C++, foreign exceptions can be caught by a `catch(...)` statement. They can also be caught as if they were of a `__foreign_exception` class, defined in `<exception>`. The `__foreign_exception` may have subclasses, such as `__java_exception` and `__ada_exception`, if the runtime is capable of identifying some of the foreign languages.

The behavior is undefined in the following cases:

- A `__foreign_exception` catch argument is accessed in any way (including taking its address).
- A `__foreign_exception` is active at the same time as another exception (either there is a nested exception while catching the foreign exception, or the foreign exception was itself nested).

- `uncaught_exception`(), `set_terminate`(), `set_unexpected`(), `terminate`(), or `unexpected`() is called at a time a foreign exception exists (for example, calling `set_terminate`() during unwinding of a foreign exception).

All these cases might involve accessing C++ specific content of the thrown exception, for instance to chain active exceptions.

Otherwise, a catch block catching a foreign exception is allowed:

- to resume normal execution, thereby stopping propagation of the foreign exception and deleting it, or
- to rethrow the foreign exception. In that case, the original exception object must be unaltered by the C++ runtime.

A catch-all block may be executed during forced unwinding. For instance, a `longjmp` may execute code in a `catch(...)` during stack unwinding. However, if this happens, unwinding will proceed at the end of the catch-all block, whether or not there is an explicit rethrow.

Setting the low 4 bytes of exception class to C++\0 is reserved for use by C++ runtimes compatible with the common C++ ABI.

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

7.2.3.1 Controlling Section Attributes

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.

7.2.3.2 Pragma for Control Flow Properties of Procedure Calls

```
/usr/include/setjmp.h:#pragma unknown_control_flow(setjmp)
/usr/include/setjmp.h:#pragma unknown_control_flow(_setjmp)
/usr/include/setjmp.h:#pragma unknown_control_flow(sigsetjmp)
/usr/include/ucontext.h:#pragma unknown_control_flow(getcontext)
/usr/include/unistd.h:#pragma unknown_control_flow(vfork)
/usr/include/sys/systm.h:#pragma unknown_control_flow(setjmp)
/usr/include/sys/systm.h:#pragma unknown_control_flow(on_fault)
/usr/include/sys/systm.h:#pragma unknown_control_flow(on_data_trap)
```

Pragma `unkown_control_flow` specifies a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`. Since such routines render standard flow graph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

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 Itanium architecture programs which will run in 32 bit address space. The Itanium architecture 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™ 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 Itanium architecture 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 Itanium architecture 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 Itanium architecture 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 as follows:

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

Right shift would sign extend integer data types.

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 *Intel® IA-64 Architecture Software Developer's Manual* 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 Itanium architecture.

7.4 Synchronization Primitives

The intrinsics described here provide a variety of primitive synchronization operations. Besides performing the particular synchronization operation, each of these intrinsics has two key properties:

- The function performed is guaranteed to be atomic (typically achieved by implementing the operation using a sequence of load-linked/store-conditional instructions in a loop on MIPS).
- Associated with each instrinsic are certain memory barrier properties that restrict the movement of memory references to visible data across the intrinsic operation (by either the compiler or the processor).

A visible memory reference is a reference to a data object potentially accessible by another thread executing in the same shared address space. A visible data object may be one of the following:

- C/C++ global data
- Fortran COMMON data
- Data declared extern
- Volatile data
- Static data (either file-scope or function-scope)
- Data accessible via function parameters
- Automatic data (local-scope) that has had its address taken and assigned to some object which is visible (recursively)

The memory barrier semantics of an intrinsic may be one of the following three types:

acquire barrier	Disallows the movement of memory references to visible data from after the intrinsic (in program order) to before the intrinsic (this behavior is desirable at lock-acquire operations, hence the name).
release barrier	Disallows the movement of memory references to visible data from before the intrinsic (in program order) to after the intrinsic (this behavior is desirable at lock-release operations, hence the name).
full barrier	disallows the movement of memory references to visible data past the intrinsic (in either direction), and is thus both an acquire and a release barrier. A barrier only restricts the movement of memory references to visible data across the intrinsic operation: between synchronization operations (or in their absence), memory references to visible data may be freely reordered subject to the usual data-dependence constraints.

Caution: Conditional execution of a synchronization intrinsic (such as within an if or a while statement) does not prevent the movement of memory references to visible data past the overall if or while construct.

7.4.1 Atomic Fetch-and-op Operations

```
"type __sync_fetch_and_add (type* ptr, type value, ...)"
"type __sync_fetch_and_sub (type* ptr, type value, ...)"
"type __sync_fetch_and_or  (type* ptr, type value, ...)"
"type __sync_fetch_and_and (type* ptr, type value, ...)"
"type __sync_fetch_and_xor (type* ptr, type value, ...)"
"type __sync_fetch_and_nand(type* ptr, type value, ...)"
```

Where type may be one of int, long, long long, unsigned int, unsigned long, or unsigned long long. The ellipsis (...) refers to an optional list of variables protected by the memory barrier.

Behavior:

- Atomically performs the specified operation with the given value on *ptr, and returns the old value of *ptr, as in the following example:

```
{ tmp = *ptr; *ptr <op>= value; return tmp; }
```

- Full barrier.

7.4.2 Atomic Op-and-fetch Operations

```
"type __sync_add_and_fetch (type* ptr, type value, ...)"
"type __sync_sub_and_fetch (type* ptr, type value, ...)"
"type __sync_or_and_fetch  (type* ptr, type value, ...)"
"type __sync_and_and_fetch (type* ptr, type value, ...)"
"type __sync_xor_and_fetch (type* ptr, type value, ...)"
"type __sync_nand_and_fetch(type* ptr, type value, ...)"
```

Where type may be one of int, long, long long, unsigned int, unsigned long, or unsigned long long. The ellipsis (...) refers to an optional list of variables protected by the memory barrier.

Behavior:

- Atomically performs the specified operation with the given value on *ptr, and returns the new value of *ptr. (i.e.)

  ```
  { *ptr <op>= value; return *ptr; }
  ```

- Full barrier.

7.4.3 Atomic Compare-and-swap Operation

```
"int __sync_bool_compare_and_swap (type* ptr, type oldvalue, type newvalue,
...)"
"type __sync_val_compare_and_swap (type* ptr, type oldvalue, type newvalue,
...)"
```

Where type may be one of int, long, long long, unsigned int, unsigned long, unsigned long long. The ellipsis (...) refers to an optional list of variables protected by the memory barrier.

Behavior:

- Atomically do the following: compare *ptr to oldvalue. If equal, store the new value. The _sync_bool_compare_and_swap version returns 1 if successful, or 0 if *ptr does not match oldvalue. I.e., the __sync_bool_compare_and_swap version does the following:

  ```
  if (*ptr != oldvalue) return 0;
  else {
          *ptr = newvalue;
          return 1;
  }
  ```

  The __sync_val_compare_and_swap version returns *ptr. (Note that doing this atomically requires looping on an architecture with an LL/SC implementation like MIPS.)

- Full barrier.

7.4.4 Atomic Synchronize Operation

"__sync_synchronize (...)"

The ellipsis (...) refers to an optional list of variables protected by the memory barrier.

Behavior:

- Full barrier

7.4.5 Atomic Lock-test-and-set Operation

```
"type __sync_lock_test_and_set (type* ptr, type value, ...)"
```

Where type may be one of int, long, long long, unsigned int, unsigned long, or unsigned long long. The ellipsis (...) refers to an optional list of variables protected by the memory barrier.

Behavior:

- Atomically store the supplied value in *ptr and return the old value of *ptr. (i.e.)
  ```
  { tmp = *ptr; *ptr = value; return tmp; }
  ```
- Acquire barrier.

7.4.6 Atomic Lock_release Operation

```
"void __sync_lock_release (type* ptr, ...)"
```

Where type may be one of int, long, long long, unsigned int, unsigned long, or unsigned long long. The ellipsis (...) refers to an optional list of variables protected by the memory barrier.

Behavior:

- Set *ptr to 0. (i.e.) { *ptr = 0 }
- Release barrier.

7.5 Thread-Local Storage

This section describes the use and implementation of thread-local storage in the *Itanium™ Conventions and Runtime Architecture Guide*.

The compiler tool chain provides direct support for the declaration of thread-local data (also referred to as thread-specific or thread-private data). The programmer may declare variables to be thread local, and the compiler will automatically arrange for those variables to be allocated on a per-thread basis.

The built-in support for this feature serves three purposes:

- It provides a foundation upon which the POSIX interfaces for allocating thread-specific data are built.
- It offers a more convenient and more efficient mechanism for direct use by applications and libraries.
- It allows compilers to allocate thread-local storage as necessary when performing loop-parallelizing optimizations.

7.5.1 C/C++ Programming Interface

A programmer declares a variable to be thread local using the __thread keyword, as in the following examples:

```
__thread int i;
__thread char *p;
__thread struct state s;
```

During loop optimizations, the compiler may choose to create thread-local temporaries as needed.

Applicability. The __thread keyword may be applied to any global, file-scoped static, or function-scoped static variable (it has no effect on automatic variables, which are always thread-local).

Initialization. In C++, a thread-local variable may not be initialized if the initialization would require a static constructor. Otherwise, a thread-local variable may be initialized to any value that would be legal for an ordinary static variable.

No variable, thread-local or otherwise, may be initialized to the address of a thread-local variable.

Binding. Thread-Local variables may be declared and referenced externally, and they are subject to the same pre-emption rules as normal symbols.

Tentative definitions. In ANSI C and C++, a thread-local variable declared without an initializer and without the extern keyword is treated as a definition. In K&R C, the treatment is unspecified (i.e., it may be treated by the implementation as a definition or a tentative definition).

Dynamic loading restrictions. A shared library, *x,* that contains thread-local storage may be loaded dynamically, via dlopen(), provided that every translation unit containing a reference to a thread-local variable defined in *x* has been compiled with the dynamic thread-local storage model.

While the static thread-local storage model generates faster code, code compiled with this model cannot reference thread-local variables in dynamically-loaded libraries. The dynamic thread-local storage model is able to reference all thread-local storage. Both thread-local storage models are described in this document.

Address-of operator. The address-of operator (&), when applied to a thread-local variable, is evaluated at run-time, and returns the address of the current thread's instance of that variable. The address obtained by this operator may be used freely by any thread in the process as long as the thread that evaluated the address remains in existence. When a thread terminates, any pointers to thread-local variables in that thread become invalid.

When the dlsym() is used to obtain the address of a thread-local variable, the address returned will be the address of the instance of that variable in the thread that called dlsym().

7.5.2 Compile-time Allocation of Thread-Local Storage

The compiler allocates thread-local storage based on how it is declared:

- If a thread-local variable is not initialized (or is initialized to zero), it is allocated in the .tbss section.
- If a thread-local variable is initialized to a non-zero value, it is allocated in the .tdata section, and the initialization value is placed into the section's initialization image. The initialization may require relocation.
- If a thread-local variable is a tentative definition, it is declared as a "TLS Common" symbol, using the SHN_TLS_COMMON section index in the symbol table entry.

The section attributes for .tbss and .tdata are listed in Table 7-1.

Table 7-1. Section Table Entries for .tbss and .tdata

sh_name	tbss	tdata
sh_type	SHT_NOBITS	SHT_PROGBITS
sh_flags	SHF_ALLOC + SHF_WRITE + SHF_TLS	SHF_ALLOC + SHF_WRITE + SHF_TLS
sh_addr	virtual address of section	virtual address of section
sh_offset	0	file offset of initialization image
sh_size	size of section	size of section
sh_link	SHN_UNDEF	SHN_UNDEF
sh_info	0	0
sh_addralign	alignment of section	alignment of section
sh_entsize	0	0

Thread-Local storage symbols must have the symbol type STT_TLS (6). In relocatable object files, the st_value field of an STT_TLS symbol contains a section-relative offset for defined symbols, or zero for undefined symbols.

7.5.3 Linker Treatment of Thread-Local Storage Sections

The linker processes “TLS Common” symbols as it processes ordinary common symbols, except that the resulting allocations are made in the .tbss section.

The linker collects the .tdata sections (i.e., all sections of type SHT_PROGBITS with the SHF_TLS flag set) into a combined .tdata section that may be allocated in any program segment, except that it must be in a writable segment if it contains any dynamic relocations.

The linker collects the .tbss sections (i.e., all sections of type SHT_NOBITS with the SHF_TLS flag set) into a combined .tbss section that is allocated immediately following the .tdata section, subject to padding for proper alignment.

The combined .tdata and .tbss sections together form a *TLS template* that is used to allocate thread-local storage whenever a new thread is created. The initialized portion of this template is called the *TLS initialization image*. All relocations generated as a result of initialized thread-local variables are applied to this template, so that the relocated values can be used when a new thread requires the initial values.

All symbols defined in a thread-local storage section are assigned offsets relative to the beginning of the TLS template. The actual virtual address associated with these symbols is irrelevant, since the address refers only to the template, and not to the per-thread copy of each data item.

In executable and shared object files, the st_value field of an STT_TLS symbol contains the assigned offset for defined symbols, or zero for undefined symbols.

Several relocations are defined to support access to thread-local storage, and the linker must process these as described in “Code Sequences for Accessing Thread-Local Variables,” below. Symbols of type STT_TLS may be referenced by only these TLS relocations, and TLS relocations may reference only symbols of type STT_TLS.

Although the .tbss section must be allocated following the .tdata section, so that symbols in .tbss receive proper template-relative offsets, it does not need to be physically allocated in the output file—that is, the address space that would be occupied by the uninitialized portion of the thread-local storage template may be overlayed by other data.

In the output file, the linker creates a new program header table entry to describe the TLS template; the fields of this entry are described in Table 7-2. The memory described by this program header table entry must be part of a loadable segment described by a PT_LOAD entry.

Table 7-2. Program Header Table Entry for Thread-Local Storage

Field	Value
p_type	PT_TLS (7)
p_offset	File offset of the TLS initialization image
p_vaddr	Virtual memory address of the TLS initialization image
p_paddr	Reserved
p_filesz	Size of the TLS initialization image
p_memsz	Total size of the TLS template
p_flags	PF_R
p_align	Alignment of the TLS template

The flag DF_STATIC_TLS (0x10) in the DT_FLAGS dynamic table entry is used to indicate that an executable or shared object file contains code using the static TLS model. The linker must set this flag when the static TLS model is used so that the dynamic loader can easily reject attempts to load such a file dynamically.

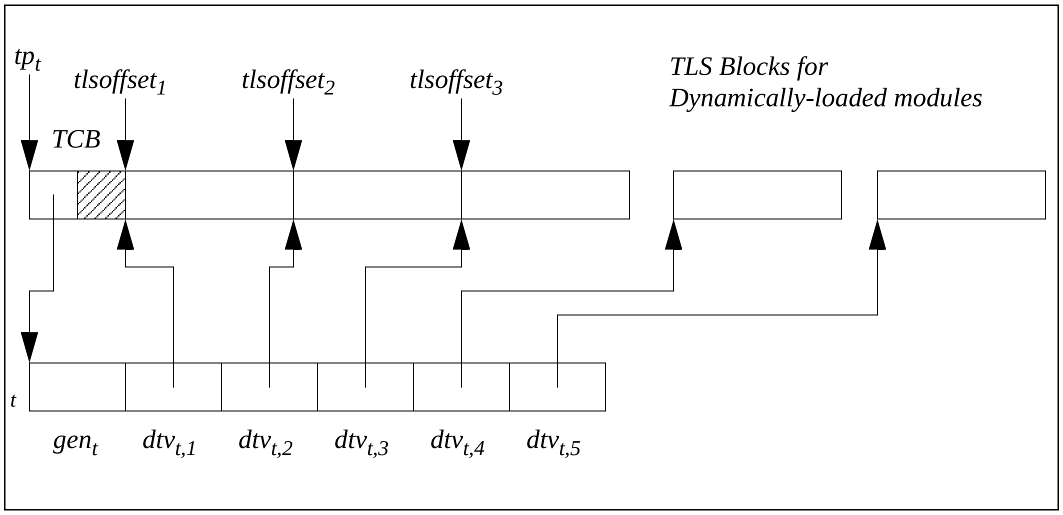
7.5.4 Runtime Allocation of Thread-Local Storage

Thread-Local storage must be created at three occasions during the lifetime of a program:

- At program startup.
- When a new thread is created.
- When a thread references a TLS block for the first time after a new library is loaded.

Figure 7-1 contains an illustration of the layout of the data structures described in this section.

Figure 7-1. Thread-Local Storage Data Structure Layout



Program startup. At program startup, the runtime system creates thread-local storage for the main thread.

First, the dynamic loader logically combines the TLS templates for all load modules in the startup set (including the a.out itself) into a single static template. Each load module's TLS template is assigned an offset within the combined template, $tlsoffset_m$, as follows:

- $tlsoffset_1$ = round(16, $align_1$)
- $tlsoffset_{m+1}$ = round($tlsoffset_m$ + $tlssize_m$, $align_{m+1}$)

where $tlssize_m$ and $align_m$ are the size of and the required alignment boundary, respectively, for the allocation template for load module m ($1 \le m \le M$, where M is the total number of load modules). The round(*offset*, *align*) function returns *offset* rounded up to the next multiple of *align*.

The first 16 bytes of the static allocation template are used by the thread library as a Thread Control Block (TCB). The doubleword at offset 0 is used as a pointer to the dynamic thread vector, dtv_t, described below under "Thread creation." The remaining 8 bytes are reserved for internal use by the thread library.

The dynamic loader also computes the total startup thread-local storage allocation size, $tlssize_S$ (equal to $tlsoffset_M$ + $tlssize_M$).

The dynamic loader then constructs a linked list of initialization records. Each record in this list describes the TLS initialization image for one load module, and contains the following four fields:

- Pointer to the TLS initialization image.
- Size of the TLS initialization image.
- The $tlsoffset_m$ for the load module.
- A flag indicating whether the load module uses the static TLS model.

The thread library allocates storage for the initial thread, initializes the storage, and creates a dynamic thread-local storage vector for the initial thread, as described under "Thread creation," below.

Thread creation. For the initial thread, and when a new thread is created, the thread library allocates a new thread-local storage block for each load module in the startup set. Depending on the implementation, it may allocate blocks separately or as a single contiguous block of length $tlssize_T$, which is the current total of the sizes of the TLS templates of all load modules (i.e, $tlssize_S$ plus $tlssize_m$ for each dynamically-loaded module), plus any padding required for alignment between TLS templates.

Each thread t has an associated thread pointer tp_t, which points to the thread's TCB. The thread pointer register, *tp* (GR 13), always contains the value of tp_t for the currently running thread.

The thread library then creates a vector of pointers, dtv_t, for the current thread t. The first element of each vector contains a generation number gen_t, which is used to determine when the vector needs to be extended. The use of this field is described under "Deferred allocation of TLS blocks," below.

Each remaining element in the vector, $dtv_{t,m}$, is a pointer to the block reserved for the thread-local storage belonging to load module m.

For dynamically-loaded modules, the thread library defers the allocation of thread-local storage blocks until an actual reference is made from the new thread. All references to TLS defined in a dynamically-loaded module must use the dynamic TLS model. For blocks whose allocation has been deferred, the pointer $dtv_{t,m}$ is set to an implementation-defined special value.

Implementation Note: The dynamic loader may, if it so chooses, group the TLS templates for the startup set of load modules such that they share a single element in the vector, $dtv_{t,1}$. This must not affect the offset calculations described above or the creation of the list of initialization records. For the following sections, however, the value of M, the total number of load modules, would start with the value 1.

The thread library then copies the initialization images to the corresponding locations within the new block of storage.

Dynamic loading. When a new library that contains thread-local storage is loaded, the dynamic loader extends the list of initialization records to include the new library's initialization template. The new load module is given an index $m = M + 1$, and the counter M is incremented by one. The allocation of new TLS blocks, however, is deferred until they are actually referenced.

Dynamic unloading. When a library that contains thread-local storage is unloaded, the implementation may choose to free the TLS blocks used for that library, or it may keep them allocated for reuse. The implementation must ensure that memory leaks do not occur as the result of repeated loading and unloading of the same library.

Deferred allocation of TLS blocks. In the dynamic TLS model, when a thread t needs to access a TLS block for load module m, the code must update the vector dtv_t and perform the initial allocation of the TLS block, if necessary. The thread library provides the following interface, which is part of the base ABI:

```
extern void *__tls_get_addr(size_t m, size_t offset);
```

This routine first checks the per-thread generation counter, gen_t, to determine whether the vector needs to be updated. If the vector dtv_t is out of date, the routine updates the vector, possibly reallocating it to make room for more entries. The routine then checks to see if the TLS block corresponding to $dtv_{t,m}$ has been allocated. If it has not been allocated, the routine allocates and initializes the block, using the information in the list of initialization records provided by the dynamic loader, and sets the pointer $dtv_{t,m}$ to point to the newly-allocated block. The routine then returns a pointer to the given offset within the block.

The methods used to determine whether the vector is out of date, and whether a particular TLS block has been allocated are implementation dependent.

7.5.5 Code Sequences for Accessing Thread-Local Variables

The compiler generates code using either the static thread-local storage model, or the dynamic thread-local storage model, depending on a compile-time switch.

Static thread-local storage model. In the static thread-local storage model, the tp-relative offset for a given variable *x* (i.e., its offset relative to the beginning of the TCB), is stored in a linkage table entry for *x*. The generated code to access *x* obtains the offset from the linkage table entry, adds this offset to the value of *tp*, and uses the resulting virtual address to load or store the variable. An example code sequence that forms the address of a thread-local variable *x* is shown in Example 1.

Example 1. Static Thread-Local Storage Model

```
        addl t1  = @ltoff(@tprel(x)), gp// find linkage tbl entry
        ;;
        ld8  t2  = [t1]              // load tp-relative offset
        ;;
        add  loc0 = t2, tp           // form address of x
```

The @ltoff(@tprel(x)) operator translates to the R_IA_64_LTOFF_TPREL22 relocation, which requests the linker to allocate a linkage table entry to hold the tp-relative offset for the variable *x*. The linker processes this relocation by substituting the gp-relative offset for the new linkage table entry.

The tp-relative offset for *x* is given by $tlsoffset_m$, where *m* is the load module containing the definition of *x*, plus the symbol value of *x*, which is its offset relative to the beginning of the load module's allocation template. Since $tlsoffset_m$ is not calculated until load time, the linker attaches an R_IA_64_TPREL64MSB/LSB dynamic relocation to the linkage table entry.

Static model with linker-assigned offsets. For references to TLS known to be in the main program (e.g., when building a statically-bound program, or when building a main program and the referenced symbol is protected), the linker can calculate the tp-relative offsets statically, without the need for dynamic relocations, and the extra reference to the linkage table. Example 2 shows the code that can be generated for this case.

Example 2. Static Model with Linker-assigned Offsets

```
        mov  r2 = tp                 // put tp where addl
        ;;                           //   use it
        addl loc0 = @tprel(x), r2  // form address of x
```

The first instruction of this sequence can be scheduled early in the code, and the copy of tp in register r2 can be used by several thread-local storage references.

The @tprel(x) operator translates to the R_IA_64_TPREL22 relocation, which requests the linker to relocate the instruction with the static tp-relative offset for the variable *x*.

A compiler may support compilation models where an assertion has been made that the tp-relative offset is smaller than 2^{13}, or larger than 2^{21}, allowing the use of the short add instruction, or requiring the use of the move long immediate instruction. The R_IA_64_TPREL14 and R_IA_64_TPREL64I relocations are also provided to support these instructions.

Dynamic thread-local storage model. In the dynamic thread-local storage model, a variable *x*, defined in load module *m*, is referenced by obtaining the pointer $dtv_{t,m}$ for the current thread *t*, and adding to this pointer the dtv-relative offset for *x*. Because the referenced TLS block may not have been allocated yet, the code must perform runtime checks described in “Deferred allocation of TLS blocks,” above. An example code sequence that forms the address of a thread-local variable *x*, using the __tls_get_addr interface, is shown in Example 3.

Example 3. Dynamic Thread-Local Storage Model

```
mov  loc0 = gp              // save gp (if necessary)
addl t1 = @ltoff(@dtpmod(x)), gp// find LT entry 1
addl t2 = @ltoff(@dtprel(x)), gp// find LT entry 2
;;
ld8  out0 = [t1]            // load value of m
ld8  out1 = [t2]            // load dtv-rel. offset
br.callrp = __tls_get_addr  // compute addr. of x
;;                          // address of x in ret0
mov  gp = loc0              // restore gp
```

The @ltoff(@dtpmod(x)) operator translates to the R_IA_64_LTOFF_DTPMOD22 relocation, which requests the linker to allocate a linkage table entry to hold the load module index *m* for the variable *x*. The linker processes this relocation by substituting the gp-relative offset for the new linkage table entry. Since the load module index *m* is not calculated until load time, the linker attaches an R_IA_64_DTPMOD64MSB/LSB dynamic relocation to the linkage table entry.

The @ltoff(@dtprel(x)) operator translates to the R_IA_64_LTOFF_DTPREL22 relocation, which requests the linker to allocate a linkage table entry to hold the dtv-relative offset for the variable *x*. The linker processes this relocation by substituting the gp-relative offset for the new linkage table entry. The linker attaches an R_IA_64_DTPREL64MSB/LSB dynamic relocation to the linkage table entry.

Referencing protected symbols in the dynamic model. If a reference is made to a hidden or protected thread-local symbol using the dynamic model, the linker can calculate a static dtv-relative offset, saving a reference to the linkage table. An example code sequence that forms the address of a protected symbol *x* is shown in Example 4.

Example 4. Referencing a Protected Symbol in the Dynamic Mmodel

```
mov  loc0 = gp              // save gp (if necessary)
addl t1 = @ltoff(@dtpmod(x)), gp// find LT entry 1
addl out1 = @dtprel(x), r0 // load dtv-rel. offset
;;
ld8  out0 = [t1]            // load value of m
br.callrp = __tls_get_addr  // compute addr. of x
;;                          // address of x in ret0
mov  gp = loc0              // restore gp
```

The @dtprel(x) operator translates to the R_IA_64_DTPREL22 relocation, which requests the linker to relocate the instruction with the static dtv-relative offset for the variable *x*.

When a procedure references more than one protected symbol, the compiler should obtain the base address of the TLS block once, then use that base address to calculate the addresses of each symbol without a separate library call. An example code sequence that forms the addresses of two protected symbols *x* and *y* is shown in Example 5.

Example 5. Referencing Several Protected Symbols in the Dynamic Model

```
        mov  loc0 = gp               // save gp (if necessary)
        addl t1 = @ltoff(@dtpmod(x)), gp// find LT entry 1
        mov  out1 = r0               // use dtv-rel. offset = 0
        ;;
        ld8  out0 = [t1]             // load value of m
        br.callrp = __tls_get_addr   // compute base addr.
        ;;                           //  of TLS block in ret0
        mov  gp = loc0               // restore gp
        mov  r2 = ret0               // prepare for addl
        ;;
        addl loc1 = @dtprel(x), r2 // form address of x
        addl loc2 = @dtprel(y), r2 // form address of y
```

A compiler may support compilation models where an assertion has been made that the dtp-relative offset is smaller than 2^{13}, or larger than 2^{21}, allowing the use of the short add instruction, or requiring the use of the move long immediate instruction. The R_IA_64_DTPREL14 and R_IA_64_DTPREL64I relocations are also provided to support these instructions.

7.5.6 ELF Relocations for Thread-Local Storage

The new relocations required to support thread-local storage are listed in Table 4-7. All mnemonics have the prefix “R_IA_64_”.

7.5.7 TLS Variable References

TLS variable can be referenced using the following function:

```
    __tls_get_addr()
```