MIPS ABIs Described

Document Number: MD00305
Revision 1.3
2002/12/02

MIPS Technologies, Inc
1225 Charleston Road
Mountain View, CA 94043-1353

Table of Contents

MIPS ABIs Described

1. Introduction and scope

This document describes standards (‘‘ABIs’’) to which compilation systems should adhere to achieve the following goals, which are ordered approximately by how challenging they are:

- *Inter-calling* : a binary program built with one compiler should be able to call a subroutine defined in another (so long as address resolution problems are solved).

  The standards relevant to this are called the ‘‘calling conventions’’; they describe how subroutines pass parameters, return values, and co-operate to share the register set and stack resources. They’re discussed in §5 below.

- *Interlinkable* : object files built with one compiler can be linked successfully with those produced by another.

  The standard relevant to this is the object code definition, in particular the definition of symbols and relocation mechanisms, and is discussed in §7 below.

- *Runnable* : a binary produced with a compliant toolkit can be successfully executed on a compliant OS (in particular we’re interested in versions of the Linux OS). The program must first be compatible with the semantics of the library and/or system calls provided by the OS, of course; the ABI says nothing about that.

  This requires standards regulating the use of run-time linked system library code. Most of this stuff is defined by related standards such as the overarching [SVR4 ABI], to which all subsequent ABI manuals are footnotes. But because that is very long and abstract, and much of this ABI would not make sense otherwise, there’s a fair amount of background material in §3 which summarises how Linux runs applications.

- *Debuggable* : more conventions and standards are required before a program build with a toolkit can be successfully debugged. People quite often use ‘‘foreign’’ debuggers with code built with the GNU toolchain, for example. The issues involved are described in §6.1.

- *Profilable* : where available, code profilers have their own requirements - related to but not identical to those of debuggers.

Three ABIs

Three MIPS ABIs are described here, perhaps with some notes on common variants. This document sets out to describe; a later document will prescribe!

The three ABIs are:

o32 An ABI elaborated from the calling and linkage conventions first developed to go with the MIPS CPUs in the early 1980s. Those conventions were extended by SGI to comply with the requirements of the generic ‘‘System V’’ specification [SVR4 ABI], and in particular to support position-independent shared libraries.

o32 as described here attempts to record the practice of MIPS/Linux systems and tools in 2002. Thanks are particularly due to Kjeld Borch Egevang for his painstaking work in elucidating these practices, recorded in the precursor of this document, [MIPSABI2].

n64 An ABI designed from scratch to fit the ‘‘System V’’ specification by SGI, for use on 64-bit CPUs only (MIPS III or superset). n64 has 64-bit representations of C pointers and `long` integers; essential to exploit the large memory space of these CPUs, but a source of unwelcome data bloat in programs which don’t use it.

n32 Very closely related to n64 and readily supported by the same OS kernel, n32 has 32-bit pointer and `long` types, but otherwise has identical rules and syntax to n64.

All these historical ABIs have features which are troublesome for the MIPS architecture, and may not be well-matched to the ‘‘embedded’’ applications where the CPUs are to be found.

A summary of differences between o32 and n32/n64 can be found as Appendix B below.

Document Conventions

C language snippets will be set in `monospace` type.

Numbers will be in `monospace` type, and preceded by a radix indication. In particular `0b0101` is the *binary* number representing ‘‘5’’, and `0x1c` is the hexadecimal number representing ‘‘28’’.

Filenames will be in `small monospace` type.

Registers will be named in `small monospace` type.

MIPS instructions (written as in an assembler source file) will be in **`bold monospace`** type.

2. Register naming conventions and usage

A MIPS CPU offers 32 general purpose registers for your program to use[1]: `$0` to `$31`. Two, and only two, behave differently from the others:

`$0` always returns zero, no matter what you store in it.

`$31` is always used by the normal subroutine-calling instruction (**`jal`**) for the return address. Note that the call-by-register version (**`jalr`**) can use *any* register for the return address, though use of anything except `$31` would be eccentric.

In all other respects the general purpose registers are identical and can be used in any instruction (it is legal to use `$0` as the destination of instructions, though the result data will disappear without trace).

In the MIPS architecture the ‘‘program counter’’ is not a register, and it is probably better for you not to think of it that way - in a pipelined CPU there are multiple candidates for its value, which gets confusing. The return address of a **`jal`** is the next instruction *but one* in sequence:

```
...
jal printf
move $4, $6
xxx     # return here after call
```

That makes sense because the instruction immediately after the call is the call’s ‘‘delay slot’’ - the MIPS architecture rules say it must be executed before the branch target. By default most assemblers hide the delay slot from you, but it’s always there. The delay slot instruction of the call is rarely wasted, because it is typically used to set up a parameter.

The floating point math coprocessor (called *FPA* for floating point accelerator), if included, adds 32 floating point registers with their own conventions: see §2.2 below.

2.1. Conventional names and uses of general-purpose registers

Although the hardware makes few rules about the use of registers, their practical use is governed by a forest of conventions, and as part of those conventions the registers are referred to by conventional names - typically defined in a header file[2] and implemented by using the C preprocessor on assembler files.

MIPS hardware cares nothing for these conventions, but all the benefits of software standardisation are lost without them. Table 2.1 shows the register numbers, conventional names and a note on their use.

In a few cases the mapping of the conventional names differs between the o32 and n32/n64 ABIs. We’ll discuss them as we go.

More about register usage

- *AT* : this register is reserved for the synthetic instructions generated by the assembler. Where you must use it explicitly - such as when saving or restoring registers in an exception handler - there’s an assembler directive to stop the assembler from using it behind your back (but then some of the assembler’s macro instructions won’t be available.)

  The assembler directive’s existence is the reason why this name is traditionally used in upper case...

- *v0-v1* : used when returning non-floating-point values from a subroutine. If you need to return anything too big to fit in two registers, the compiler will allocate a memory buffer whose address will be passed as an invisible first argument.

  While a function is running `v0-v1` can be freely used as temporaries.

  In o32 only integer values are returned in these registers. Structure or array types (even if small enough to fit in the two registers) are always returned through a data area defined by the caller and whose address is an invisible first argument. (This rule appears to be a bug blessed by tradition.)

- *a0-a3/a7* : used to pass the first four (o32) or eight (n32/n64) non-FP parameters to a subroutine. But that’s a misleading simplification; see §5 below for a more complicated but correct description.

[1] The contents of this section and some others are adapted with permission from ‘‘See MIPS Run’’ - see [SMR]

[2] `/usr/include/asm/regdef.h` on most Linux/MIPS systems.

Register Nos	*name*	*use*		
`$0`	`zero`	always zero		
`$1`	`AT`	assembler temporary		
`$2-$3`	`v0-v1`	return value from function		
`$4-$7`	`a0-a3`	arguments		
	o32		*n32/n64*	
	name	*use*	*name*	*use*
`$8-$11`	`t0-t3`	temporaries	`a4-a7`	more arguments
`$12-$15` `$24-$25`	`t4-t7` `t8-t9`		`t0-t3` `t8-t9`	temporaries
`$16-$23`	`s0-s7`	saved registers		
`$26-$27`	`k0-k1`	reserved for interrupt/trap handler		
`$28`	`gp`	global pointer		
`$29`	`sp`	stack pointer		
`$30`	`s8/fp`	frame pointer if needed (additional saved register if not)		
`$31`	`ra`	Return address for subroutine		

Table 2.1: Conventional names of registers with usage mnemonics

Argument registers which are unused or whose value is no longer needed can be freely used as temporaries.

The odd-looking reallocation of temporary register names and numbers for n32/n64 vs o32 is some kind of effort to reduce the amount of effort spent porting assembler modules; I doubt if it helped!

- *t0-t9* : by convention, subroutines may use these registers without doing anything to preserve their previous contents. This makes them a good choice for ‘‘temporaries’’ when evaluating expressions - but the compiler/programmer must remember that values stored in them may be destroyed by a subroutine call.
- *s0-s8* : by convention, subroutines must guarantee that the values of these registers on exit are the same as they were on entry - either by not using them, or by saving them on the stack and restoring before exit.

  This makes them eminently suitable for use as ‘‘register variables’’ or for storing any value which must be preserved over a subroutine call.
- *k0-k1* : reserved for use by an OS’ trap/interrupt handlers, which will use them and not restore their original value; so they are of little use to anyone else. Not used at all by application code.
- *gp* : has two quite different roles. In position-independent (PIC) code - typically used only for application and library code in a large OS - it is used in the double-indirection used to reach variables and functions whose location is not known until the program and its libraries are loaded.

  In non-PIC code, typically used for all non-Linux embedded applications, it’s sometimes used to provide efficient access to C `static`/`extern` data.

  We’ll consider the two uses separately.
- *gp in PIC code* : In position-independent code `gp` acts as a pointer to the GOT (‘‘global offset table’’), as described in §3.2.

  The GOT pointer is loaded by code in the prologue of every function which makes a reference through the GOT.

  In the n32/n64 ABIs the value in `gp` is defined to survive a function call; so functions using the register must save it on entry and restore its old value before exit.

  In o32 PIC code, a function call may overwrite the value in `gp` and the compiler must ensure it’s reloaded after any such call. This isn’t very efficient...
- *gp in non-PIC code* : (if used) `gp` is initialised to point to a load-time-determined location in the midst of your static data. This means that loads and stores to data lying within 32Kbytes either side of the `gp` value can be performed in a single instruction using `gp` as the base register. Note that the pointer in `gp` is a constant; no application code ever writes to the register once it has been initialised.

Without the global pointer, loading data from a static memory area takes two instructions: one to load the most significant bits of the 32-bit constant address computed by the compiler and loader, and one to do the data load.

To use `gp` a compiler must know at compile time that a datum will end up linked within a 64Kbyte range of memory locations. In practice it can't know, only guess. The usual practice is to put ''small'' global data items (8 bytes and less in size) in the `gp` area, and to get the linker to complain if it still gets too big. The compiler *-Gnn* flag can be used to adjust the threshold of what is considered ''small''.

Not all compilation systems and not all run-time systems support `gp`.

- *sp* : (stack pointer). It takes explicit instructions to raise and lower the stack pointer, so MIPS code usually adjusts the stack only on subroutine entry and exit; and it is the responsibility of the subroutine being called to do this. `sp` is normally adjusted, on entry, to the lowest point that the stack will need to reach at any point in the subroutine. Now the compiler can access stack variables by a constant offset from `sp`.
- *fp* : (also known as `s8`). A subroutine will use a ''frame pointer'' to keep track of the stack if it wants to do things which involve extending the stack by an amount which is determined at run-time. Some languages - including C++ - may do this implicitly; assembler programmers are always welcome to experiment; and C programs which use some efficient implementations of the `alloca()` library routine will find themselves doing so.

  If the stack bottom can't be computed at compile time, you can't access stack variables from `sp`, so `fp` is initialized by the function prologue to a constant position relative to the function's stack frame. Cunning use of register conventions means that this behavior is local to the function, and doesn't affect either the calling code, or any nested function calls.
- *ra* : (return address). On entry to any subroutine, ra holds the address to which control should be returned - so a subroutine typically ends with the instruction: **`jr ra`**.

  Subroutines which themselves call subroutines must first save `ra`, usually on the stack.

2.2. Floating Point register conventions

There are a corresponding set of standard uses for floating point registers too.

MIPS floating point

MIPS CPUs which have FPA hardware have 32 floating point registers, whose assembler names are `$f0` - `$f31`. Even 32-bit MIPS CPUs support the 64-bit IEEE double-precision format.

The o32 ABI generates code compatible with ''traditional'' 32-bit MIPS I and MIPS II CPUs, which do arithmetic only in the 16 even-numbered registers `$f0` - `$f30`. The odd-numbered registers are referred to in move and load/store instructions; but the assembler provides synthetic ''macro'' instructions for move and load/store double, so you will probably never see the odd-numbered registers when writing o32 code.

The n32/n64 ABIs run only on 64-bit MIPS CPUs, and exploit their ability to have 32 independent 64-bit registers. You can run o32 code on 64-bit CPUs, but only by setting the FPA into a compatibility mode where the odd-numbered registers disappear.

The MIPS32 specification gives a third option, where you get 32 registers which work in pairs for double precision (as per MIPS I) but also work independently to provide 32 registers usable for single-precision calculation. So far, no FPU of this model has been built, but it is compatible with o32 - somewhat over-engineered, but does everything required.

FP register software use and calling conventions

Like the general-purpose registers, the MIPS calling conventions add a whole bunch of rules about register use which are nothing to do with the hardware; they tell you which FP registers are used for passing arguments, which ones' values are expected to be preserved over function calls and so on.

The division of functions is much as for the integer registers, less the special cases.

It may be worth stressing that the role of the odd-numbered registers is not affected by the CPU's ''endianness''.

Figure 2.1 shows these for each ABI. Note that the o32 ABI assumes that the CPU either has a MIPS II or earlier FP unit or that the CPU has the `SR[FR]` compatibility bit cleared to zero; in either case only 16 registers are usable for arithmetic, so there are no odd-numbered registers in the table.

n32/n64 are usable only when all 32 floating point registers are exposed.

There are some conventional names defined for the floating point registers and reflecting these roles; but they don't seem to be much used, and are not described here.

role	*o32*	*n32*	*n64*
function return values	`$f0, $f2`		
argument registers	`$f12, $f14`	`$f12-$f19`	
saved over function call (suitable for register variables)	evens `$f20–$f30`		`$f24–$f31`
temporaries (not saved over function call, or "caller−saved')	evens `$f4–$f10, $f16, $f18`	evens `$f4–$f10, $f16, $f18`, all odds `$f1–$f31`†	`$f1, $f3–$f11, $f20–$f23`

Table 2.2: Floating point register usage conventions

† This strange difference between n32 and n64 (it's the *only* difference, apart from the mapping of the pointer and `long` data types) is evidently due to an attempt to increase the portability of assembler programs from o32 to n32. It seems unlikely that this was a good idea.

3. Background material

This section is not part of the ABI definition. But some background and some common definitions here will allow us to keep the real definition shorter and more comprehensible.

3.1. Virtual memory layout

In particular, it's useful to draw some simple pictures of the various pieces of code and data which might make up an ABI-compliant application. For ''static'' applications this is almost too simple to be required; but for Linux applications the pictures contain a fair amount of information.

Names used for memory regions and object code chunks

Some definitions:

- *Module* : A compilation unit (the assembler is seen as just another compiler...), and also used for an object file generated from one compilation unit.
- *Program* : all the addressable data and code associated with an application. Strictly speaking, that associated with an instance of an application; in Linux there may be many copies of the shell running, and they're distinct programs in this sense.

  For Linux applications, this excludes the kernel and other parts of the memory map which are not accessible to the application.
- *Link unit* : a part of a program which has been bound together so that its components are at fixed offsets from each other.
- *Segment*[3] : a part of a program which is contiguous in the memory image of the running program, and which is distinguished for link/build purposes. By ancient convention segment names begin with a dot, and are called things like `.text` and `.bss`.

  When several modules are being combined into a single link unit during the build process, *sections* of the same name in different modules are brought together and various sections concatenated to make a segment.
- *_main*[4] : C programmers think execution begins with `main()`; but in reality there's always a more primitive, machine-dependent startup routine supplied by your build environment. This does things like initialising the `sp` register to mark the stack region, and zero-ing the memory region which contains the ''uninitialised'' C variables. If you write C++, this will also arrange for initialisor routines to be run.

3.1.1. Simple standalone application

Figure 3.1 depicts some important features of a simple MIPS application.

[3] Used in a sense which harmonises with its more technical use in the formal descriptions of object files.

[4] This is probably the wrong name for Linux.

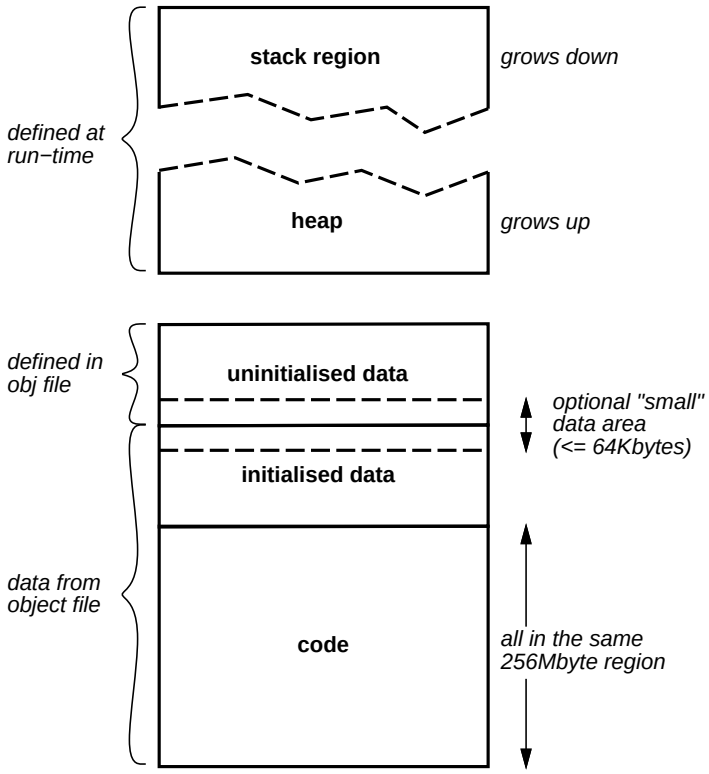

Figure 3.1 Memory map of typical statically-linked application

What's to see here?

- *Code, initialised and uninitialised data* : form a single link unit; their relative positions are fixed when the software is built. In fact, their *absolute* locations in program memory are also typically fixed at build-time; this code is position-dependent.
- *Stack* : is assigned by the start-up program in accordance with OS and toolchain conventions. It grows down, so is typically placed at the top of the program's memory space.
- *Heap* : an old-fashioned word for data space allocated by the program through C `setbrk()` or (slightly higher level) `malloc()` calls. The heap usually starts at the lowest suitably aligned location available after allowing for the linked code and data.
- *Small data area* : a MIPS special. It takes two MIPS instructions to load from or store to a C location declared at module level or as `static`. Where the ''small'' data area is used, the `gp` register is set to point to the middle of it by `__main()`. Now you can load and store to variables in that area with a single instruction.

  You can't expect many programs entire data to fit within the 64Kbyte address range limit imposed by the MIPS load/store instruction's 16-bit offset; so during compilation and build only data items below a certain size are considered as candidates for this area - that's why it's called ''small'' data.

  The small data area, if provided, overlaps both the initialised and uninitialised segments (and is implemented as a pair of sub-segments).
- *Common segment names* :

`.text`	all the code
`.data`	initialised data possibly excluding...
`.sdata`	initialised data for the ''small'' area
`.bss`	uninitialised data possibly excluding
`.sbss`	uninitialised data for the ''small'' area.

A word of warning: in many embedded OS' all the software runs in the same address space (perhaps because the OS doesn't use the MIPS memory management facilities at all). Such systems have complicated memory maps which are deeply OS dependent.

3.1.2. Linux application

Several things make Linux applications more interesting. These applications are built without their library functions; the library routines are linked in as the program is loaded into memory. The library routine may have been updated since the application was built, and it should still work[5]. The result is a much more complicated memory map with a number (perhaps quite a large number) of separately linked pieces, sketched in Figure 3.2.

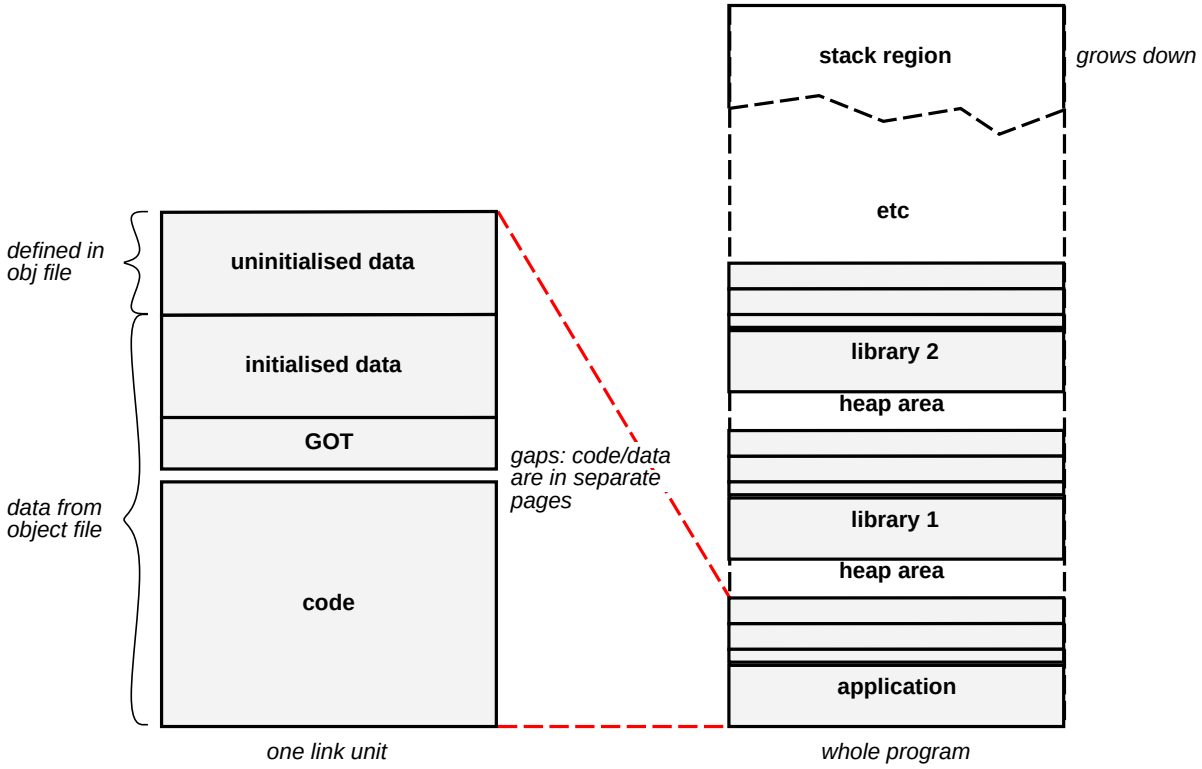

Figure 3.2 Memory map for Linux application and its libraries

In the memory map of Figure 3.2 all the link units except the base application are *shared libraries* of some kind (either built-in shared libraries or dynamically loaded by explicit programming). They are loaded into program memory working upward, first-come first-served. While the base application runs at program addresses which were known at build time, the libraries must be able to run at arbitrary memory addresses[6].

The requirement that library modules should link in just anywhere and still work (''position-independent code'', always ''PIC'' in Linux discussions) forces considerable changes to the way code is generated. The MIPS architecture's preferred subroutine call instruction is **`jal`**, and that instruction is not PC-relative; it encodes (most of) the absolute virtual address of the subroutine entry point. Moreover, Linux standards (and UNIX®standards before them) require that the shared libraries should also be able to share `extern` data. But this is a big subject for the next section.

[5] A version system based on conventional names should make sure that the library you pick up provides a compatible interface.

[6] In practice code and data segments start at the beginning of a virtual memory page, commonly 4Kbytes. Some OS versions may align link units to a small multiple of a page size.

3.2. PIC code and the Global Offset Table

An application's binary code is built before you know where data or subroutines in other link units will reside in the program's memory map; both the absolute and relative position of each link unit depends on what versions of what libraries get loaded in what order.

It isn't possible for the run-time loader to fix up these addresses in the code itself, because the code itself must be shared between different instances of the application program (and each instance may have a different library layout). So there's no hope that an application's or library's binary will contain the address information needed to reference functions and data in a different link unit.

The Global Offset Table (GOT)

Instead, the compiler generates code which makes every function call and every reference to `static`/`extern` data indirect, via a table of pointers. The table of pointers is the *Global Offset Table* or "GOT"; it is in a data segment and separate copies are kept for each instance of the application, so it can be and is fixed up by the loader. There's a sketch of how you might see it in Figure 3.3 below. The GOT contains an entry for each function or data item that is accessed by any code in the link unit (the loader finds each item by its name, so we often say there's an entry for each *symbol*). The table offset for a particular symbol *is* known at build time, and is a constant in the binary code.

For MIPS code, the `gp` register is maintained as a pointer to the GOT of the link unit[7].

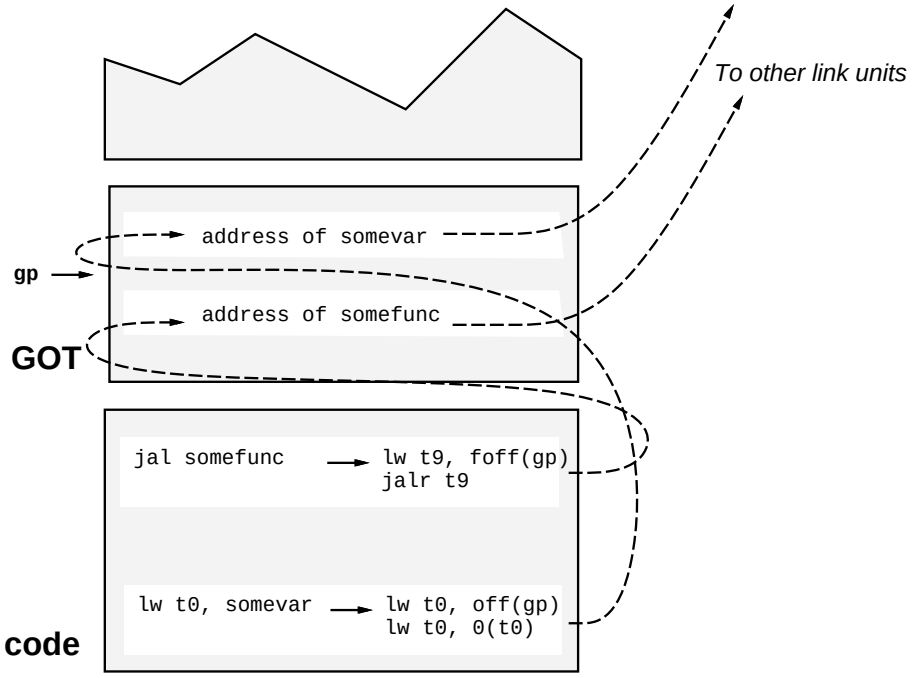

Figure 3.3 The GOT in action

The `gp` register is set to point to the GOT by code included as part of the prologue of each function (at least, each function which makes any use of the GOT.) This is suboptimal, since for intra-link-unit calls it will already hold the right value... but the compiler can't (in general) distinguish intra- and inter-link-unit calls.

In the o32 ABI, the calling code must be aware that a function call might overwrite the value in the `gp` register, and the caller must preserve or recalculate the value after the call if required; in the n32/n64 ABIs, `gp` is defined as a register whose value must be preserved, so any function which uses it has to save the old value of `gp` and restore it before exit. This is undoubtedly a better idea...

[7] There can be more than one GOT in a link unit in some circumstances; see §3.2 below.

Loading a PIC application and its libraries

The program loader is the Linux application which really runs when you load any binary which uses shared libraries - it's the ''interpreter'' for these files[8]. The program loader maps the application code and data and any libraries it needs into the program's address space. The build system leaves a list of required library names in the application's object file, and the program loader finds the library files via a series of search path mechanisms - for details see [LoaderHowTo]. Conventions about file names (if followed correctly) make sure the program finds the ''right'' library.

The program loader maintains symbol tables for the data items and subroutine entry points which are exported by the application and each library, so it can tie up references between separate link units.

Lazy loading/binding of libraries

While you don't have to actually read in all the code of the libraries required by an application (the ordinary virtual memory paging system takes care of that), the process of binding in a link unit, fixing up its GOT and getting it ready for use is relatively time-consuming. This penalty is paid even for libraries which provide facilities which the application rarely uses. That can slow the application startup.

So as an optimisation Linux defers loading and fixing up libraries until they are first used. By the nature of the PIC code the unresolved references are all in the GOT. Where the first reference to the new library is a function call this is relatively straightforward; the GOT entry for an unresolved subroutine reference is set to point to a function in the run-time loader which then loads the library, patches the GOT so that future calls will go direct, and calls the library function.

Allowing libraries to be ''demand-loaded'' after a program trips over a missing data reference is more difficult. It could certainly be done by co-operation between the run-time loader and the virtual memory exception handlers in the kernel - but this isn't done in current Linux/MIPS kernels (at least up to 2.5). There are other more subtle issues (for example, where the same symbol is provided by two different libraries) which make lazy loading problematic. So the build system is charged with identifying which libraries are safe to lazy-load, and to identify them in the application binary. The loader can then load unsafe libraries at startup.

Dynamic (explicit) loading of libraries - dlopen()

It's also possible to get software to pick its own shared library and then build an explicit software-visible table of calls to it. This mechanism (which is reminiscent of Microsoft Windows ''DLL''s) fits naturally onto the object/class concepts of C++, and libraries loaded like this are referred to as ''dynamic shared objects''.

You don't have to build a Linux shared library in a special way to make it fit for `dlopen()` - any library will do.

At the lowest level you call `dlopen()` to grab the library and `dlsym()` calls to obtain pointers to named data items or functions in the dynamic shared object. But because dynamic libraries are just shared libraries, you get some unexpected ''bonus'' semantics.

Firstly, the explicitly-loaded library will gain access to any public symbols in the application (or its pre-loaded libraries). Perhaps more unexpectedly, a straightforward `extern` function pointer reference in the application can bind to a symbol from a library which wasn't mentioned at all at build time, but only brought in with `dlopen()`.

It's much more complicated than that, of course; there has been substantial theological debate in the Linux mailing lists... everyday programmers beware.

PIC/GOT problems

There are some complications which are worth mentioning here.

- *What to do when your GOT overflows*: On MIPS, GOT pointer loads are usually compiled to a single load relative to the `gp` register; but this can only span a table 64Kbytes in size (16K pointer entries, or only 8K in n64). Large applications and libraries can use more symbols than that.

  There are two approaches. One is to just let the GOT grow above 64Kbytes, and require the compiler to generate code which can load/store arbitrary entries in it. This generally uses the *gcc -PIC* option - it's trouble-free and portable but generates truly awful code.

[8] On Linux it's usually `/lib/ld-linux.so`.

Some compilers support an option where you can generate one GOT to each *module* in a link unit (*gcc -multigot*). Done properly this is no trouble, but the dynamic loader has to know about it.

- *Ameliorating the overheads of PIC code* : nothing in the ABI *obliges* the compiler to go through the GOT when accessing data or calling subroutines which are in the same link unit; neither is it strictly necessary for a function to reset the `gp` register on an intra-link-unit call.

  However, there are several reasons why this hasn't been done:

  1. Even within the link unit only relative addresses are known; the MIPS architecture lacks efficient PC-relative call and load instructions.
  2. The compiler doesn't know which references are in the link unit. While it's possible to get the linker to do some instruction re-writing to simplify intra-link-unit calls and references, it's bad practice...
  3. The PIC calling convention for MIPS requires that on entry to a function the `t9` register holds the address of the function's entry point. Since calls made through the GOT mean the address may be in some register, this seems unproblematic - but this requirement is burdensome to any possible future intra-link-unit (or even intra-module) call mechanism.

4. How data types map onto memory (and endianness)

For the purposes of this document memory is taken as an array of unsigned 8-bit quantities, whose index is the virtual address. For all known MIPS architecture CPUs this corresponds to a C definition `unsigned char []`.

Like all the modern computers I know of, MIPS uses 2s-complement representation for signed integers - so in any data size ‘‘-1’’ is represented by binary all-ones. The overwhelming advantage of 2s-complement numbers is that the basic arithmetic operations (add, subtract, multiply, divide) have the same implementation for signed and unsigned data types[9].

C integer data types come in `signed` and `unsigned` versions, which are always the same size and alignment. When you don’t specify which you typically get a `signed int`, `long` or `long long` but often an *unsigned* `char`[10].

Sizes of basic types

Table 4.1 lists fundamental C data types and how they’re implemented for MIPS architecture CPUs. We’ll come back to the `long` and pointer types a bit later - their size changes according to which ABI you use.

C type	*MIPS asm name*	*size (bytes)*
`char`	**`byte`**	**1**
`short`	**`half`**	2
`int`	**`word`**	4
`long long`†	**`dword`**	8
`float`	**`word`**‡	4
`double`	**`dword`**‡	8

Table 4.1: Data types and memory representations

Size of “long” and pointer types

Although these vary according to the type, in practice they’re always stored the same as something else... For the n64 ABI `long` is implemented just like the `long long` shown above, while for o32 and n32 `long` is implemented just like an `int`.

And then in all three ABIs a pointer is always implemented as an `unsigned long`; the MIPS architecture always boasts a simple ‘‘flat’’ address space.

Alignment requirements

All these primitive data types can only be directly handled by standard MIPS instructions if they are *naturally aligned*: that is, a 2-byte datum starts at an address which is even (zero modulo 2), a 4-byte datum starts at an address which is zero modulo 4, and an 8-byte datum starts at an address which is zero modulo 8[11].

4.1. Memory layout of basic types and how it changes with endianness

Table 4.2 shows how each basic type is laid out in our byte-addressed memory; the arrangement is different for big-endian and little-endian software.

---

[9] At least, until the result has greater precision than the operands.

[10] This is an ANSI C feature. In early C `char` was also signed by default. Most compilers allow you to change the default for `char` with a command line flag.

† The `long long` data type is familiar to GNU C users, and widely used elsewhere.

‡ The assembler does not distinguish storage definitions for integer and floating-point data types.

[11] For MIPS32 CPUs using only 32-bit registers and data paths, 8-byte data types are not handled by any machine instruction and the 8-byte alignment restriction is not strictly necessary. However, it is still imposed in all known ABIs.

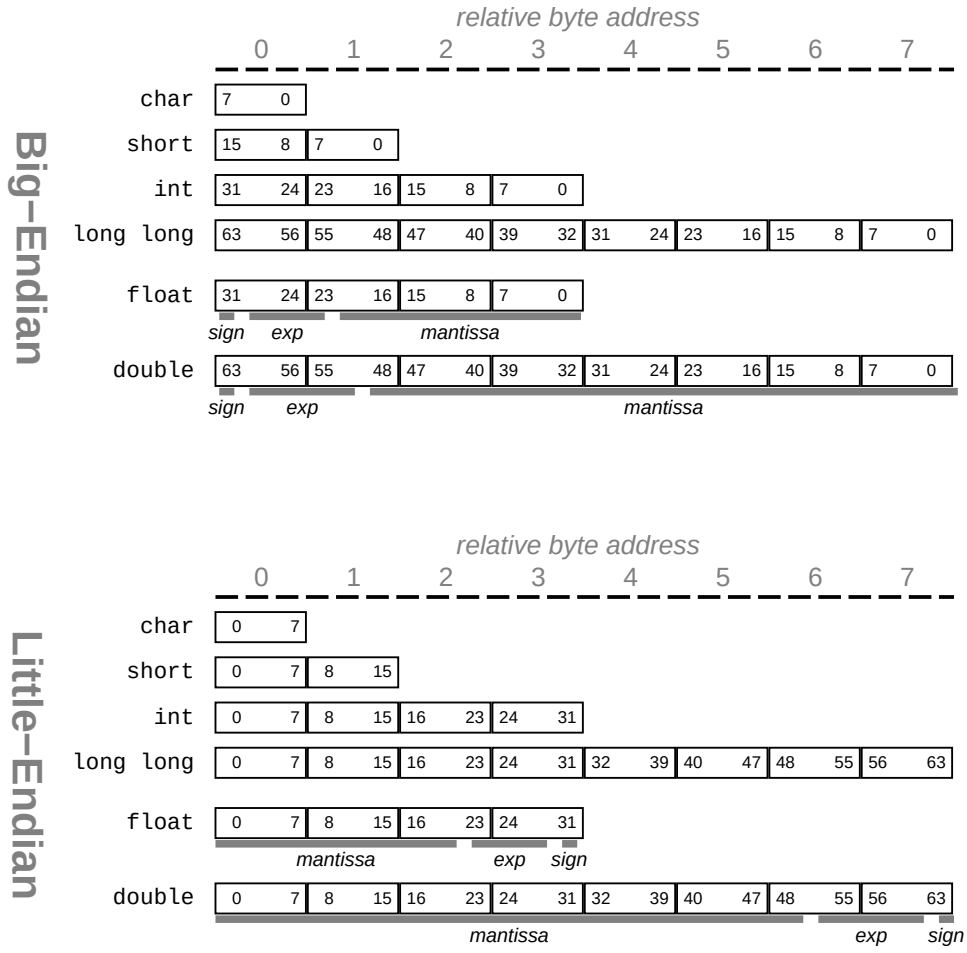

Table 4.2: C data types in memory

In Table 4.2 I've given in to the temptation to reverse the bit-numbering within each byte for the big-endian layouts. For memory addressing purposes this is meaningless; bytes are indivisible 8-bit objects. However, reversing the bit numbers as above makes the bitwise depiction of the fields of floating point numbers easier to absorb (and prettier).

Each of these data types is naturally aligned, as described above.

''Endianness'' can be a troubling subject. If you are uneasy about it, read it up in [SMR].

4.2. Memory layout of structure and array types and alignment

Complex types are built by concatenating simple types, but inserting unused (''padding'') bytes between items so as to respect the alignment rules[12].

It's worth giving a couple of examples. Here's the byte offsets of data items in a `struct mixed`:

[12] Some compiler systems provide mechanisms to alter the alignment rules for particular data definitions. This allows you to model more possible data patterns with C data declarations, and the compiler will generate appropriate code (with some loss of efficiency) to handle the resulting unaligned basic data types. But such features are outside the scope of this document.

```
struct mixed {
    char c;     /* byte 0 */
                /* bytes 1-14 are ‘‘padding’’ */
    double d;   /* bytes 8-15 */
    short s;    /* bytes 16-17 */
};
```

It’s worth stressing that the byte offsets of the fields of constructed data types (*other than those using C bitfields*, see §4.2 below) are unaffected by endianness.

Constructed data types are aligned in memory to the largest alignment boundary required by an data type defined inside them. So a `struct mixed` will start on an 8-byte boundary; and that means that if you build an array of these structures you will need padding between each array element. C compilers provide for this by ‘‘tail padding’’ the structure to make it usable for an array, so `sizeof(struct mixed) == 24` and the structure should really be annotated:

```
struct mixed {
    char c;     /* byte 0 */
                /* bytes 1-14 are ‘‘padding’’ */
    double d;   /* bytes 8-15 */
    short s;    /* bytes 16-17 */
                /* bytes 18-23 are ‘‘tail padding’’ */
};
```

Just to remind you: the size and alignment requirement of pointer and `long` data types can be 4 or 8, depending on the ABI.

Bit fields in structures

C allows you to define structures which pack several short ‘‘bit field’’ members into one or more locations of a standard integer type. This is a useful feature for emulation, hardware interfacing, and perhaps for defining dense data structures, but is fairly incomplete. Bitfield definitions are nominally CPU-dependent (but so is everything) but also genuinely endianness-dependent.

One can, for example, define a data structure which permits access to the various fields of a MIPS single-precision floating point number:

```
#if BYTE_ORDER == BIG_ENDIAN

struct ifloat {
    unsigned int sign:1;
    unsigned int bexp:8;
    unsigned int mant:23;
};

#else /* little-endian */

struct ifloat {
    unsigned int mant:23;
    unsigned int bexp:8;
    unsigned int sign:1;
};

#endif
```

In this case (as you’d hope and expect) the three fields are packed into one 32-bit `int` storage unit. How do the two cases differ? Well, for both endianness’ the bitfields are allocated with the first-defined field occupying the lowest byte-addressed part of the `int`. For big-endian, that means the high-order bits are occupied first; for little-endian, it’s the low-order bits.

Does this make sense? Certainly some; if you tried to implement bitfields in a less endianness-dependent way, then in the following example `struct fourbytes` would have a different memory layout from `struct fouroctets`- and that doesn't seem reasonable:

```
struct fourbytes {
  signed char a; signed char b; signed char c; signed char d;
}

struct fouroctets {
  int a:8; int b:8; int c:8; int d:8;
}
```

A field can only be packed inside one storage unit of its defined type; if we try to define a structure for a MIPS double-precision floating point number, the mantissa field contains part of two 32-bit `int` storage units and can't be defined in one go. The best we can do is something like this:

```
struct ieee754dp_konst {
    unsigned    sign:1;
    unsigned    bexp:11;
    unsigned    manthi:20;  /* cannot get 52 bits into... */
    unsigned    mantlo:32;  /* .. a regular C bitfield    */
};
```

You're permitted to leave out the name of the field definition, so you don't have to invent names for fields which are just there for padding.

Although ANSI doesn't require it, many compilers permit you to use bitfields of type other than `int`. We could have used an `unsigned long long` bit field and defined the double-precision floating point register in one go.

The full alignment rules for bit-fields are complicated:

- As we said above, a bit-field must reside entirely in a storage unit that is appropriate for its declared type. Thus a bit-field never crosses its unit boundary.
- Bit-fields can share a storage unit with other struct/union members, including members that are not bit-fields (to pack together, the adjacent structure member must be of a smaller integer type).
- Structures generally inherit their own alignment requirement from the alignment requirement of their most demanding type. Named bit-fields will cause the structure to be aligned (at least) as well as the type requires.

  Unnamed fields - regardless of their defined type - only force the storage unit or overall structure alignment to that of the smallest integer type which can accommodate that many bits.
- You might want to be able to force subsequent structure members to occupy a new storage unit. In some compilers you can do that with an *unnamed zero-width* field. Zero-width fields are otherwise illegal (or at least pointless).

You now know everything you need to map C data declarations to memory in a manner compatible with the various ABIs.

5. Calling conventions

The calling convention describes how arguments are passed to functions, and how values are returned. It's also a convenient place to describe the stackframe structure which builds up to represent the current function nest.

ANSI C permits pretty much any value - structures and arrays as well as scalars - to be passed as arguments or returned by a function.

Stack maintenance and alignment

When the stack is adjusted by functions to make space for local variables, register saves and argument passing it is always adjusted by a multiple of 8 bytes (o32) or 16 bytes (n32/n64), so that the stack base is aligned to the greatest extent required by any variable.

Registers and the argument structure

For efficiency, we want (most of the time) to pass arguments in registers, and avoid data loads/stores. But C permits pretty much any non-array data type - no matter how large or complex - to be passed as an argument[13]. It isn't going to be ''obvious'' how such arguments should be passed. To make sure we handle corner cases correctly, the set of arguments passed to a function is mapped as it would be to a memory-based *argument structure*, and then as much of that structure as will fit is pasted into the available registers. For any arguments left over after all available argument registers have been used up, we put a copy of that part of the argument structure onto the stack.

Here are the rules:

1. Each argument is aligned to the start of a new *argument slot* within the argument structure; these slots are 4 bytes on o32 and 8 bytes on n32/n64 - chosen to match the size of the general-purpose registers.

   If the next slot doesn't have the correct alignment for a value (for example, a `double` on o32 requires 8-byte alignment), it is skipped to find a slot which is correctly aligned. Skipped slots remain unused.

   Large arguments may spill over into more than one slot.
2. Integer values are first converted to the type of the argument, if there's a function prototype, using standard C rules. Where there's no function prototype the rules (derived from old K&R C) are that integer and floating point values are coerced to `signed int` and `double` respectively.
3. Integers smaller than `int` are expanded to `int` by zero- or sign-extending them in accordance with C rules.

   Then on n32/n64, where the argument slot is bigger than the an `int`, the value is expanded up to the size of a register by sign-extension, in accordance with the way the MIPS architecture represents 32-bit values in 64-bit registers.
4. Non-integer arguments smaller than a register-sized slot are aligned to the lowest addressed part of the slot.
5. In o32, `float` arguments are 8-byte aligned and occupy two slots (even though there's nothing useful in the second four bytes). Note that you need function prototypes to be passing `float`rather `double` arguments
6. The argument registers are rigidly identified with a particular slot in the argument structure. If for alignment or other reasons a slot cannot be used, then the corresponding register won't be used to pass an argument.

From this point on o32 is different from n32/n64.

	o32	*n32/n64*
7.	The caller will always build an argument data structure, even though it may remain unused in whole or part. Moreover, the data structure is always a minimum of 16 bytes (four register-sized slots) in size.	The caller need not provide any data structure unless the arguments occupy more space than can be mapped onto registers, and the remaining arguments have to be passed on the stack.

[13] In practice, C programmers almost always prefer to pass pointers to data structures, but we can't rely on that.

	o32	*n32/n64*
8	The first 4 × register-sized (ie 4 byte) slots of the structure are mapped to registers `a0-3`. point register pairs `$f12`/`$f13` and `$f14`/`$f15` - more often known as `fa0` and `fa1`.	The first 8 register-sized (8 byte) slots of the structure are mapped to registers `a0-7` (for integer arguments), and the same slots mapped to eight floating point registers `$f12-$f19`. Slots which are known to contain a properly aligned floating point value are passed in the floating point register; everything else is passed in the integer register.
9.	o32 does not assume the existence of function prototypes. For reasons to do with the implementation of functions with variable numbers of arguments, it is difficult to ensure that the caller and the called function always agree when to use a floating point rather than a general-purpose register for an argument. o32's rule is that up to two leading floating point arguments will be passed in FP registers, but if the first argument is not an FP a second FP argument will *not* be put in an FP register. In functions like `printf()` the first argument is a pointer, so floating point values will be passed in integer registers or on the stack.	n32/n64 do require function prototypes, so that the caller has information about the type of arguments and can determine whether to use a floating point register.

That's it. Armed with the information above you can describe the register and stack values to be passed for any possible set of C arguments.

Returning values from a function

In all the ABIs a simple scalar value is returned in a register; `v0` for integers, and `fv0` for floating point values. A second integer register is defined for returning larger values, and is used when returning a `long long` value in o32.

In n32/n64 a structure value will be returned in the registers if it fits (that is, if it's 16 bytes or less in size). A structured value will only be returned in floating point registers when it consists exactly of one or two floating point fields, and nothing else.

For all other structures or larger values which are not accommodated in the registers, the caller must provide a pointer to a memory buffer (usually on the stack, but that's not mandatory). The caller prepends a pointer to the memory buffer as an implicit first argument, followed by its explicit arguments. The called function should copy the return value to the supplied address.

5.1. Calling conventions extended for Linux ("MIPS ABI") PIC code.

In PIC code functions are not called directly; instead the compiler/assembler generate code which loads the function address from the GOT table (see §3.2 above). The disassembled code looks something like this:

```
/* (caller) */
        lw t9, <function symbol offset in GOT>(gp)
        # nop
        jalr t9
        # nop
        ...

/* function */
        /* _gp_disp is magic symbol for offset between start of
           function and gp pointer into GOT */
        li gp, _gp_disp
        addu gp, gp, t9
        ...
```

It's mandatory that the `t9` register should be used to compute the function address; the function itself depends on it to recalculate the GOT base register `gp`[14]. **`_gp_disp`** is calculated so as to place `gp` 32Kbytes on from the start of the GOT, to maximise the amount of the table which is in reach of a MIPS load instruction (which has a ±32K offset range).

[14] It's not obvious that a function can compute the GOT base address with only its own address for input; but a glance back at the memory map Figure 3.2 should remind you that the code, data and GOT of a link unit are loaded together in memory, with their offsets from one another fixed at link time.

6. Further constraints required by other tools

It's possible to build an application fully compliant with the ABI as described up to this point - it should load and run correctly on a compliant OS - but it may be very ill-behaved in relation to other development tools. For example, the debugger may be unable to reconstruct a stack backtrace after a breakpoint. In this section we aim to identify known constraints of significant tools.

6.1. Meeting debugger assumptions

The *gdb* debugger obtains information about the program under test, in the first place, from the object file. When a program is compiled with the *-g* option, the compiler notes lots of extra information about the program and provides it as a special debug section of the object file. To reach the final executable object file the debug information has to pass through the assembler and linker, but neither of them knows much about the data format or what it means: for most purposes, you should visualise that the compiler speaks straight to the debugger.

Various debug formats are in use: for o32 use ''stabs'' [STABS] is popular, but n32/n64 use DWARF2.

The ''stabs'' format doesn't encode much information about functions. Some things which are missing (and what the debugger does about it):

- *Function end marker* : the debugger assumes a function continues until the next function entry point symbol in the text segment, or the end of the segment.
- *Size of a functions stack region* : the debugger infers the size of the stack by reading the first few instructions from the function entry point looking for a **`subui sp, sp, nn`** instruction used to lower the `sp` register.
- *Layout of a functions stack* : the debugger requires that functions lay out their stack in a particular order:

7. ELF object code

Programs ready to be executed, shared libraries ([SVR4 ABI] calls them ‘‘shared objects’’) are kept as ELF files. Intermediate compilation and link files are kept as ELF too, but that’s not of any further interest in the ABI.

The ELF format is a binary file format obtained by mapping C language structures into memory, and then dumping the structure to disk byte-by-byte from its memory image. Everyone who ever taught you programming will have told you not to do this - and they were right - but:

- ELF is designed to hold large, raw images which will be paged into virtual memory when the application is run. Such images are useful only in the context of the appropriate CPU architecture, OS and ABI rules. They can’t be portable, for most purposes.
- The key ELF data structures which are most often used for navigation of the file are constructed from a small range of ‘‘machine-independent’’ data types with explicit alignment rules, so that object code tools - which may be run on a host machine of quite different architecture from the target - can still make sense of them.
- The machine-dependent ELF data structures are otherwise defined as being laid out as per the rules in this ABI document for the target CPU; so at least the definition is unambiguous.

For truly authoritative information, consult [SVR4 ABI].

This document will proceed by reference to a well-defined[15] snapshot of Linux/MIPS header files and other source code; it will only define bits where wanting to assert that the snapshot is wrong. Otherwise, it will only explain what is in the source files.

What’s in an ELF file?

We’ll proceed by reference to an example. For each chunk of the file we’ll show the name used in [SVR4 ABI], the header file and structure where it’s defined, and (to keep us rooted) what’s to be found in the executable for `/bin/ls` on a typical Linux/MIPS system.

To be supplied

Table 7.1: Structures, fields and examples in an executable ELF file

[15] Exact version numbers will be added, I promise.

8. Debug information in object code.

To do its job, the debugger needs intimate information about a program; it needs to know where all variables (even local ones) are in memory, and what C line corresponds to each instruction that was generated.

When given the *-g* flag, the compiler generates that information. The assembler just wraps it up and passes it through, the linker does nothing except to collect it into sections; for most purposes you can assume that the compiler alone generates debug information and the debugger alone consumes it.

As we saw above, ELF files can contain arbitrary sections. Existing Linux/MIPS toolchains generate debug information in a style called ‘‘stabs’’, which was invented at the University of California at Berkeley and popularised by Sun: see [STABS] for more information.

9. References

SMR	‘‘See MIPS Run’’
LoaderHowTo	the Linux Documentation Project’s ‘‘Program Library HOWTO’’, available as a web page.
SVR4 ABI	The ‘‘System V Application Binary Interface’’ Edition 4.1, written by AT&T and the Santa Cruz Operation Inc.
MIPSABI2	"MIPS® Linux Application Binary Interface (ABI) Specification", MIPS Technologies document MD00245.
STABS	‘‘The ’stabs’ debug format’’ - GNU tools manual written by team members at Cygnus Support. Available as web (HTML) or printable format from many places.
DWARF	

The web site

For internal MIPS Technologies[16] use refer to `http://ukwww.algor.co.uk/abi/` for updates, references and other web materials relevant to this manual. The external references and links to published MIPS documentation will soon be made available on a publicly-accessible web site, too.

Locating compliant source code for Linux/MIPS

[This section will eventually have a list of CVS revisions and archives describing the versions used for reference].

[16] Outsiders please ask your favourite MTI contact.

Appendix A: Revision History

Revision	Date	Description
1.2d	4th October 2002	First visibility of incomplete document outside MTI.

Appendix B: Differences between o32 and n32/n64 ABIs

	o32	*n32/n64*
Registers saved and restored as	32-bit	64-bit
Argument structure	4-byte slots 8-byte alignment at least 4 slots long	8-byte slots 16-byte alignment no minimum size
Argument registers	4 integer, 2 FP	8 integer, 8 FP
Arguments in FP registers?	Leading FP arguments only (doesn't need correct function prototypes)	Any FP argument which occupies a whole slot (except that `varargs` arguments are always passed in integer registers). Horribly fragile without prototypes, which are assumed available.
Return values	Only scalars are ever returned in registers; `v1` is used only for `long long` data.	Any structure of up to 16 bytes in size will be returned in the registers. A structure consisting of one or two FP values (and nothing else) will be returned in the two FP return-value registers (for compatibility with the Fortran `complex` type.)
long long type	implemented with register pairs and library calls	hardware type
gp register in PIC code	not preserved over calls	preserved over calls