Native 64-bit Subprogram Interface for MIPS Architectures

This document describes the internal subprogram interface for native 64-bit programs, which was introduced with the MIPSpro version 6.0 compiler suite. We assume some familiarity with the current 32-bit interface conventions as specified in the MIPS ABI (i.e. the MIPS supplement to the SVR4 Application Binary Interface specification, which we will refer to as the *32-bit ABI*, as the *old 32-bit ABI*, or as the *o32 ABI*).

This document also describes a proposed new 32-bit subprogram interface, which is largely identical to the 64-bit subprogram interface. It differs only in that certain data types (i.e. long and pointer) produce 32-bit data objects while they would have been 64-bit objects in the 64-bit ABI. This ABI is generally called the *new 32-bit ABI,* or the *n32 ABI*. Since this document does not discuss the interface in terms of the source language types, these differences are generally irrelevant.

There are other components of an ABI which we do not address in this document, specifically the source language type mapping and storage layout, the format used for object files, and operating system call interfaces.

The transition to native 64-bit code on the MIPS R4000 and subsequent MIPS architecture processors requires subprogram interface changes due to the changes in register and address size. This specification takes advantage of this transition to revamp the interface to improve performance in a number of areas.

The principal interface for 64-bit native code is superficially similar to the old 32-bit ABI, with all 32-bit objects replaced by 64-bit objects. In particular, this implies:

- All integer parameters are promoted (i.e. sign- or zero-extended) to 64-bit integers and passed in a single register. (No code is normally required for the promotion.)
- All pointers and addresses are 64 bit objects in the 64-bit ABI; they remain 32 bit objects in the n32 ABI.
- Floating point parameters are passed as single- or double-precision according to the ANSI C rules.
- All stack parameter slots become 64-bit doublewords, even for parameters which are smaller, e.g. floats and 32-bit integers.

In more detail, the 64-bit native calling sequence will have the following characteristics. We use square brackets to indicate different old 32-bit ABI conventions.

- All stack regions must be quadword aligned. [The o32 ABI specifies only doubleword alignment, although the MIPSpro 6.0 implementation also provides quadword alignment.]
- Up to eight integer registers (**$4**..**$11**) are used to pass integer arguments. [The o32 ABI uses only the four registers **$4**..**$7**.]
- Up to eight floating point registers (**$f12**..**$f19**) are used to pass floating point arguments. [The o32 ABI uses only the four registers **$f12**..**$f15**, with the odd registers used only for halves of double-precision arguments.]
- The argument registers may be viewed as an image of the initial eight doublewords of a structure containing all of the arguments, where each of the argument fields is a multiple of 64 bits in size with doubleword alignment. The integer and floating point registers are distinct images, i.e. the first doubleword is passed in either **$4** or **$f12** depending on its type, the second in either **$5** or **$f13**, etc. [The o32 ABI associates each floating point argument with an even/odd pair of integer or floating point argument registers.]
- Within each of the 64-bit save area slots, smaller scalar integer parameters are right-justified, i.e. placed at the highest possible address (for big-endian targets). This is relevant only for integer parameters of 32 or fewer bits. Of these, only **int** parameters arise in C (plus *long* and pointer parameters in the n32 ABI) except for prototyped cases — smaller integers are promoted to **int**. [This is true for the o32 ABI, but was only relevant to prototyped small in-

tegers since the **int** objects were register-sized. This is done to simplify the varargs implementation.]

- Within each of the 64-bit save area slots, **float** parameters are left-justified, i.e. placed at the lowest possible address (for big-endian targets). [This case does not arise in the o32 ABI, since **float** parameters are the size of the save area slot.]
- 32-bit integer (i.e. **int**) parameters are always sign extended when passed in registers, whether of signed or unsigned type. [This issue does not arise in the o32 ABI.]
- Quad-precision floating point parameters (i.e. C **long double**, or Fortran **REAL*16**) are always 16-byte aligned. This requires that they be passed in even-odd floating point register pairs, even if doing so requires skipping a register parameter and/or a 64-bit save area slot. [The o32 ABI did not consider **long double** parameters, since they were not supported previously. The MIPSpro 6.0 implementation uses the same rule.]
- Structs, unions, or other composite types are treated as a sequence of doublewords, and are passed in integer or floating point registers as though they were simple scalar parameters to the extent that they fit, with any excess on the stack packed according to the normal memory layout of the object. More specifically:
  a. Regardless of the struct's field structure, it is treated as a sequence of 64-bit *chunks*. If a chunk consists solely of a double float field (but not a double which is part of a union), it is passed in a floating point register. Any other chunk is passed in an integer register.
  b. A union, either as the parameter itself or as a struct parameter field, is treated as a sequence of integer doublewords for purposes of assignment to *integer* parameter registers. No attempt is made to identify floating point components for passing in floating point registers.
  c. Array fields of structs are passed like unions. Array parameters are passed by reference.
  d. Notwithstanding the earlier comment about right-justifying small scalar parameters in their save area slots, struct parameters are always left-justified. This applies both to the case of a struct smaller than 64 bits, and to the final chunk of a struct which is not an integral multiple of 64 bits in size. The implication of this rule is that the address of the first chunk's save area slot will be the address of the struct, and the struct will be laid out in the save area memory exactly as if it were normally

allocated (once any part in registers has been stored to the save area).

[These rules are analogous to the o32 ABI treatment — only the chunk size and the ability to pass double fields in floating point registers are different.]

- Whenever possible, floating point arguments are passed in floating point registers regardless of whether they are preceded by integer parameters. [The o32 ABI allows only leading FP arguments to be passed in FP registers; those coming after integer registers must be moved to integer registers.]

  Varargs routines require an exception to this rule. Any floating point parameters in the variable part of the argument list (leading or otherwise) are passed in integer registers. There are several important cases involved:

  a. If a varargs prototype (or the actual definition of the callee) is available to the caller, it places floating point parameters directly in the integer register required, and there are no problems.
  b. If no prototype is available to the caller for a direct call, the caller's parameter profile is provided in the object file (as well as all global subprograms' formal parameter profiles), and the linker (ld/rld) generates an error message if the linked entrypoint turns out to be a varargs routine.
  c. If no prototype is available to the caller for an indirect call (i.e. via a function pointer), the caller assumes that the callee is not a varargs routine and places floating point parameters in floating point registers. (If the callee is varargs, it is not ANSI-conformant.)
  d. For cases (b) and (c), the MIPSpro 6.0 implementation provides an option (-TENV:varargs_prototypes=OFF) to force unprototyped double parameters to be placed in both the floating point register and the corresponding integer register. This option is not to be considered a normal part of this interface.

- The portion of the "argument structure" beyond the initial eight doublewords is passed in memory on the stack, pointed to by the stack pointer at the time of call. The caller does not reserve space for the register arguments; the callee is responsible for reserving it if required (either adjacent to any caller-saved stack arguments if required, or elsewhere as appropriate). No requirement is placed on the callee either to allocate space and save the register parameters, or to save them in any particular place. [The o32 ABI requires the caller to reserve space for the register arguments as well.]

- Function results are returned in **$2** (and **$3** if needed), or **$f0** (and **$f2** if needed), as appropriate for the type. Composite results (struct, union, or array) will be returned in **$2**/**$f0** and **$3**/**$f2** according to the following rules:
  1. A struct with *only* one or two floating point fields is returned in **$f0** (the first field) and **$f2** if necessary (the second field). This is a generalization of the Fortran COMPLEX case.
  2. Any other struct or union results of at most 128 bits are returned in **$2** (first 64 bits) and **$3** (remainder, if necessary).
  3. Larger composite results are handled by converting the function to a procedure with an implicit first parameter which is a pointer to an area reserved by the caller to receive the result. [The o32 ABI also requires that the callee return the address of the result in **$2**.]

  [The o32 ABI requires that all composite results be handled by conversion to implicit first parameters. The MIPS/SGI Fortran implementation has always made a specific exception to return **COMPLEX** results in the floating point registers.]
- There are eight callee-saved floating point registers, **$f24..$f31** for the 64-bit interface. There are six for the n32 ABI interface, the six even registers in **$f20..$f30**. [The o32 ABI specifies the six even registers, or even/odd pairs, **$f20..$f31**.]
- Routines are not restricted to a single exit block. [The o32 ABI makes this restriction, though it is not observed under all optimization options. The MIPSpro 6.0 implementation supports this restriction as part of the option -OPT:space, independent of optimization level.]
- We place no requirement on the register which must be used to hold the return address in exit blocks. The **.mdebug** format was unable to cope with return addresses in different places, but the **DWARF** format can. [The ABI specifies **$31**, but the current implementation supports **.mask** as an alternative.]
- *Additional PIC-related (shared code) requirements are defined elsewhere. We generate PIC (position-independent code, for DSO support) directly from the compiler, rather than converting it later with a separate tool, in order to allow better compiler control for instruction scheduling and other optimizations, and in order to provide greater robustness.*

- The global pointer, or GOT pointer, **gp**, is a callee-saved register. [The o32 ABI makes **gp** a caller-saved register in shared code.]
- Table 1 specifies the usage of registers in native 64-bit mode. Note that “caller-saved” means only that the caller may not assume that the value in the register will be preserved across the call.

Table 1 **Native 64-bit Interface Register Conventions**

Register Name	Software Name	Use
$0	zero	Hardware zero
$1 or $at	at	Assembler temporary — caller saved
$2..$3	v0..v1	Function results — caller saved
$4..$11	a0..a7	Subprogram arguments — caller saved
$8..$11	ta0..ta3	... alternate names
$12..$15	t0..t3	Temporaries — caller saved
$16..$23	s0..s7	Callee saved registers
$24	t8	Temporary — caller saved
$25	t9	Temporary — caller saved
		Address of callee in PIC code
$26..$27	kt0..kt1	Reserved for kernel
$28 or $gp	gp	Global pointer — callee saved
$29 or $sp	sp	Stack pointer — callee saved
$30	s8	Frame pointer (if needed) — callee saved
$31	ra	Return address — caller saved
hi, lo		Multiply/divide special registers — caller saved
$f0, $f2		Floating point function results — caller saved
$f1, $f3		Floating point temporaries — caller saved
$f4..$f11		Floating point temporaries — caller saved
$f12..$f19		Floating point arguments — caller saved
$f20..$f23		-64 Floating point temporaries — caller saved
$f24..$f31		-64 Floating point callee saved registers
$f20..$f31 even		-n32 Floating point temporaries — callee saved
$f20..$f31 odd		-n32 Floating point caller saved registers
$fcc0..$fcc7		Floating point condition codes — caller saved