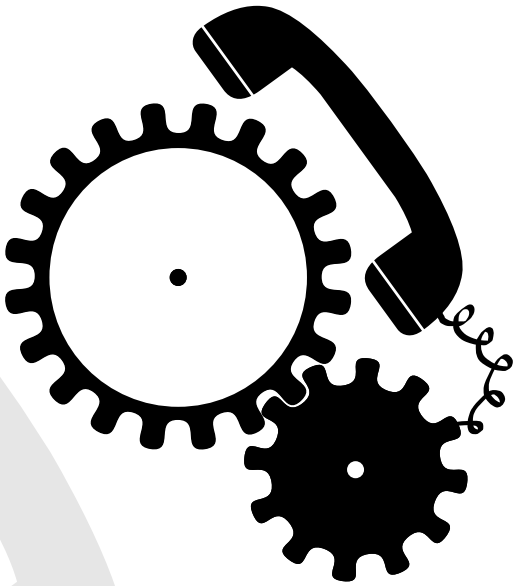
dyncall

Version 1.4

Daniel Adler (dadler@uni-goettingen.de)
Tassilo Philipp (tphilipp@potion-studios.com)

December 6, 2022



Contents

List of Tables

List of Figures

Listings

1 Motivation

Interoperability between programming languages is a desirable feature in complex software systems. While functions in scripting languages and virtual machine languages can be called in a dynamic manner, statically compiled programming languages such as C, C++ and Objective-C lack this ability. The majority of systems use C function interfaces as their system-level interface. Calling these (foreign) functions from within a dynamic environment often involves the development of so called "glue code" on both sides, the use of external tools generating communication code, or integration of other middleware fulfilling that purpose. However, even inside a completely static environment, without having to bridge multiple languages, it can be very useful to call functions dynamically. Consider, for example, message systems, dynamic function call dispatch mechanisms, without even knowing about the target.

The *dyncall* library project provides a clean and portable C interface to dynamically issue calls to foreign code using small call kernels written in assembly. Instead of providing code for every bridged function call, which unnecessarily results in code bloat, only a modest number of instructions are used to invoke all the calls.

1.1 Static function calls in C

The C programming language and its direct derivatives are limited in the way function calls are handled. A C compiler regards a function call as a fully qualified atomic operation. In such a statically typed environment, this includes the function call's argument arity and type, as well as the return type.

1.2 Anatomy of machine-level calls

The process of calling a function on the machine level yields a common pattern:

1. The target function's calling convention dictates how the stack is prepared, arguments are passed, results are returned and how to clean up afterwards.
2. Function call arguments are loaded in registers and on the stack according to the calling convention that take alignment constraints into account.
3. Control flow transfer from caller to callee.
4. Process return value, if any. Some calling conventions specify that the caller is responsible for cleaning up the argument stack.

The following example depicts a C source and the corresponding assembly for the X86 32-bit processor architecture.

```
extern void f(int x, double y,float z);
void caller()
{
  f(1,2.0,3.0f);
}
```

Listing 1: C function call

```
.global f          ; external symbol 'f'
caller:
  push  40400000H ; 3.0f (32 bit float)
                  ; 2.0  (64 bit float)
  push  40000000H ;           low  DWORD
  push  0H        ;           high DWORD
  push  1H        ; 1    (32 bit integer)
  call  f         ; call 'f'
  add   esp, 16   ; cleanup stack
```

Listing 2: Assembly X86 32-bit function call

2 Overview

The *dyncall* library encapsulates architecture-, OS- and compiler-specific function call semantics in a virtual

bind argument parameters from left to right and then call

interface allowing programmers to call C functions in a completely dynamic manner. In other words, instead of calling a function directly, the *dyncall* library provides a mechanism to push the function parameters manually and to issue the call afterwards.
Since the idea behind this concept is similar to call dispatching mechanisms of virtual machines, the object that can be dynamically loaded with arguments, and then used to actually invoke the call, is called CallVM. It is possible to change the calling convention used by the CallVM at run-time. Due to the fact that nearly every platform comes with one or more distinct calling conventions, the *dyncall* library project intends to be a portable and open-source approach to the variety of compiler-specific binary interfaces, platform specific subtleties, and so on...

The core of the library consists of dynamic implementations of different calling conventions written in assembler. Although the library aims to be highly portable, some assembler code needs to be written for nearly every platform/compiler/OS combination. Unfortunately, there are architectures we just don't have at home or work. If you want to see *dyncall* running on such a platform, feel free to send in code and patches, or even to donate hardware you don't need anymore. Check the **supported platforms** section for an overview of the supported platforms and the different calling convention sections for details about the support.

2.1 Features

- A portable and extendable function call interface for the C programming language.
- Ports to major platforms including Windows, Mac OS X, Linux, BSD derivates, iPhone and embedded devices and more, including lesser known and/or older platforms like Plan 9, Playstation Portable, Nintendo DS, etc..
- Add-on language bindings to Python, R, Ruby, Go, Erlang, Java, Lua, sh, ...
- High-level state machine design using C to model calling convention parameter transfer.
- One assembly *hybrid* call routine per calling convention.
- Formatted call, vararg function API.
- Comprehensive test suite.

2.2 Showcase

Foreign function call in C

This section demonstrates how the foreign function call is issued without, and then with, the help of the *dyncall* library and scripting language bindings.

```
double call_as_sqrt(void* funptr, double x)
{
  return ( ( double (*)(double) )funptr)(x);
}
```

Listing 3: Foreign function call in C

Dyncall C library example

The same operation can be broken down into atomic pieces (specify calling convention, binding arguments, invoking the call) using the *dyncall* library.

```
#include <dyncall.h>
double call_as_sqrt(void* funptr, double x)
{
  double r;
  DCCallVM* vm = dcNewCallVM(4096);
  dcMode(vm, DC_CALL_C_DEFAULT);
  dcReset(vm);
  dcArgDouble(vm, x);
  r = dcCallDouble(vm, funptr);
  dcFree(vm);
  return r;
}
```

Listing 4: Dyncall C library example

This is more code than a direct, hardcoded function call, however it's completely dynamic. Also, despite this coming with an overhead of more executed code per single function call, compared to function interface wrapper tools that generate per call glue-code less code is used overall, .

The following are examples from script bindings:

Python example

```
import pydc
def call_as_sqrt(funptr,x):
  return pydc.call(funptr,"d)d", x)
```

Listing 5: Dyncall Python bindings example

R example

```
call.as.sqrt <- function(funptr,x)
  .dyncall(funptr,"d)d", x)
```

Listing 6: Dyncall R bindings example

2.3 Supported platforms/architectures

The feature matrix below gives a brief overview of the currently supported platforms. Different colors are used, where a green cell indicates a supported platform, with both, call and callback support, yellow a platform that might work (but is untested) and red a platform that is currently unsupported. Gray cells are combinations that don't exist at the time of writing, or that are not taken into account.
Light green cells mark complete feature support, including passing aggregates (struct, union) by value. Dark green means basic support but lacking lesser used features (e.g. no aggregate-by-value support). Please note that a green cell (even a light-green one) doesn't imply that all existing calling conventions/features/build tools are supported for that platform (but the most important).

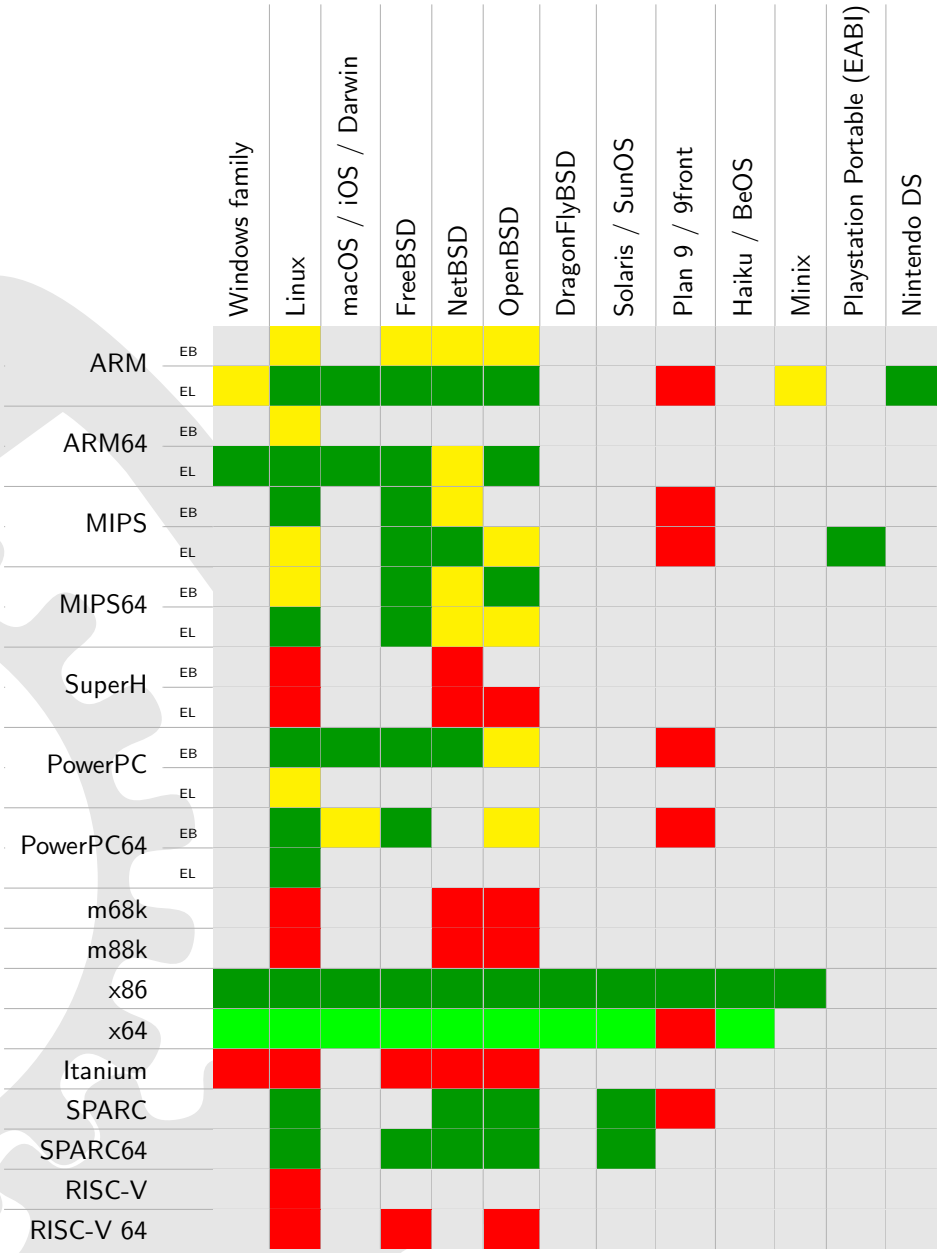


Table 1: Supported platforms

2.4 Build Requirements

The library needs at least a c99 compiler with additional support for anonymous structs/unions (which were introduced officially in c11). Given that those are generally supported by pretty much all major c99 conforming compilers (as default extension), it should build fine with a c99 toolchain.

3 Building the library

The library has been built and used successfully on several platform/architecture configurations and build systems. Please see notes on specfic platforms to check if the target architecture is currently supported.

3.1 Requirements

The following tools are supported directly to build the *dyncall* library. However, as the number of source files to be compiled for a given platform is small, it shouldn't be difficult to build it manually with another toolchain.

- C compiler to build the *dyncall* library (GCC, Clang, SunPro or Microsoft C/C++ compiler)
- C++ compiler to build the optional test cases (GCC, Clang, SunPro or Microsoft C/C++ compiler)
- BSD make, GNU make, Microsoft nmake or mk (on Plan9) as automated build tools
- Python (optional - for generation of some test cases)
- Lua (optional - for generation of some test cases)
- CMake (optional support)

3.2 Supported/tested platforms and build systems

Building *dyncall* is a straightforward two-step process, first configure, then make. The library should be able to be built with the default operating systems' build tools, so BSD make on BSD and derived systems, GNU make on GNU and compatible, mk on Plan9, nmake on Windows, etc.. This is a detailed overview of platforms and their build tools that are known to build *dyncall*:

Platform	Build Tool(s)	Compiler, SDK
Windows	nmake,Visual Studio	cl, cygwin (gcc), mingw (gcc)
Unix-like	GNU/BSD/Sun make	gcc, clang, sunc
Plan9	mk	8c
Haiku/BeOS	GNU make	gcc
iOS/iPhone	GNU make	gcc and iPhone SDK on Mac OS X
Nintendo DS	nmake	devkitPro[43] tools on Windows
Playstation Portable	GNU make	psptoolchain[44] tools

3.3 Build instructions

1. Configure the source (not needed for Makefile.embedded)

   ***nix flavour**

   ```
   ./configure [--option ...]
   ```

   Available options (omit for defaults):

`--help`	display help
`--prefix=`*`path`*	specify installation prefix (Unix shell)
`--target=`*`platform`*	MacOSX,iOS,iPhoneSimulator,PSP,...
`--sdk=`*`version`*	SDK version

   Windows flavour, and cross-build from Windows (PSP, NDS, etc.)

   ```
   .\configure [/option ...]
   ```

   Available options:

`/?`	display help
`/target-x86`	build for x86 architecture (default)
`/target-x64`	build for x64 architecture
`/target-arm64`	build for x64 architecture
`/target-psp`	cross-build for PlayStation Portable (homebrew SDK)
`/target-nds-arm`	cross-build for Nintendo DS (devkitPro, ARM mode)
`/target-nds-thumb`	cross-build for Nintendo DS (devkitPro, THUMB mode)
`/tool-msvc`	use Microsoft Visual C++ compiler (default)
`/tool-gcc`	use GNU Compiler Collection
`/tool-clang`	use GNU Compiler Collection
`/asm-ml`	use Microsoft Macro Assembler (default)
`/asm-as`	use the GNU or LLVM Assembler
`/asm-nasm`	use NASM Assembler

   Plan 9 flavour

   ```
   ./configure.rc [--option ...]
   ```

   Available options (none, at the moment):

`--help`	display help

2. Build the static libraries *dyncall*, *dynload* and *dyncallback*

   ```
   make                        # for {GNU,BSD} Make
   nmake /f Nmakefile          # for NMake on Windows
   mk                          # for mk on Plan9
   ```

3. Install libraries and includes (supported for GNU and BSD make based builds, only)

   ```
   make install
   ```

4. Optionally, build the test suite

```
make tests                          # for {GNU,BSD} Make
nmake /f Nmakefile tests            # for NMake on Windows
mk tests                            # for mk on Plan9
```

3.4 Build-tool specific notes

Some platforms require some manual tweaks:

Problem: Build fails because CC and/or related are not set, or different compiler, linker, etc. should be used.

Solution: Set the 'CC' and other environment variables explicitly to the desired tools. E.g.:

```
CC=gcc make
```

Problem: On windows using mingw and msys/unixutils 'Make', the make uses 'cc' for C compilation, which does not exist in mingw.

Solution: Set the 'CC' environment variable explicitly to e.g. 'gcc' (as in the example above).

3.5 Build with CMake

```
cmake -DCMAKE_INSTALL_PREFIX=<location>
make
```

4 Bindings to programming languages

Through binding of the *dyncall* library into a scripting environment, the scripting language can gain system programming status to a certain degree.
The *dyncall* library provides bindings to Erlang[1], Java[2], Lua[3], Python[4], R[5], Ruby[6], Go[7] and the shell/command line.
However, please note that some of these bindings are work-in-progress and not automatically tested, meaning it might require some additional work to make them work.

4.1 Common Architecture

The binding interfaces of the *dyncall* library to various scripting languages share a common set of functionality to invoke a function call.

4.1.1 Dynamic loading of code

The helper library *dynload* which accompanies the *dyncall* library provides an abstract interface to operating-system specific mechanisms for loading and accessing executable code out of, but not limited to, shared libraries.

4.1.2 Functions

All bindings are based on a common interface convention providing a common set of the following 4 functions (exact spelling depending on the binding's scripting environment):

load - load a module of compiled code

free - unload a module of compiled code

find - find function pointer by symbolic names

call - invoke a function call

4.1.3 Signatures

A signature is a character string that represents a function's arguments and return value types. It is used in the scripting language bindings invoke functions to perform automatic type-conversion of the languages' types to the low-level C/C++ data types. This is an essential part of mapping the more flexible and often abstract data types provided in scripting languages to the strict machine-level data types used by C-libraries. The high-level C interface functions `dcCallF()`, `dcVCallF()`, `dcArgF()` and `dcVArgF()` of the *dyncall* library also make use of this signature string format.

The format of a *dyncall* signature string is as depicted below:

dyncall signature string format

*<input parameter type signature character>** ')' *<return type signature character>*

The *<input parameter type signature character>* sequence left to the ')' is in left-to-right order of the corresponding C function parameter type list.
The special *<return type signature character>* 'v' specifies that the function does not return a value and corresponds to `void` functions in C.

Signature character	C/C++ data type
'v'	void
'B'	_Bool, bool
'c'	char
'C'	unsigned char
's'	short
'S'	unsigned short
'i'	int
'I'	unsigned int
'j'	long
'J'	unsigned long
'l'	long long, int64_t
'L'	unsigned long long, uint64_t
'f'	float
'd'	double
'p'	void*
'Z'	const char* (pointing to C string)
'A'	aggregate (struct, union) by-value

Table 2: Type signature encoding for function call data types

Please note that using a '(' at the beginning of a signature string is possible, although not required. The character doesn't have any meaning and will simply be ignored. However, using it prevents annoying syntax highlighting problems with some code editors.

Calling convention modes can be switched using the signature string, as well. A '_' in the signature string is followed by a character specifying what calling convention to use, as this affects how arguments are passed. This makes only sense if there are multiple co-existing calling conventions on a single platform. Usually, this is done at the beginning of the string, except in special cases, like specifying where the varargs part of a variadic function begins. The following signature characters exist:

Signature character	Calling Convention
':'	platform's default calling convention
'*'	platform's default C++/thiscall calling convention
'e'	vararg function
'.'	vararg function's variadic/ellipsis part (...), to be specified before first vararg
'c'	only on x86: cdecl
's'	only on x86: stdcall
'F'	only on x86: fastcall (MS)
'f'	only on x86: fastcall (GNU)
'+'	only on x86: thiscall (MS)
'#'	only on x86: thiscall (GNU)
'A'	only on ARM: ARM mode
'a'	only on ARM: THUMB mode
'$'	syscall

Table 3: Calling convention signature encoding

Examples of C function prototypes

	C function prototype	dyncall signature
void	f1();	")v"
int	f2(int, int);	"ii)i"
long long	f3(void*);	"p)L"
void	f3(int**);	"p)v"
double	f4(int, bool, char, double, const char*);	"iBcdZ)d"
void	f5(short, long long, ...);	"_esl_.di)v" (for (promoted) varargs: double, int)
struct A	f6(int, union B);	"iA)A"
short	Cls::f(unsigned char, ...);	"_*p_eC_.i)s" (C++: this-ptr as 1st arg, int as vararg)

Table 4: Type signature examples of function prototypes

4.2 Erlang language bindings

The OTP library application `erldc` implements the Erlang language bindings.

Signature character	accepted Erlang data types
'v'	no return type
'B'	atoms 'true' and 'false' converted to bool
'c', 'C'	integer cast to (unsigned) char
's', 'S'	integer cast to (unsigned) short
'i', 'I'	integer cast to (unsigned) int
'j', 'J'	integer cast to (unsigned) long
'l', 'L'	integer cast to (unsigned) long long
'f'	decimal cast to float
'd'	decimal cast to double
'p'	binary (previously returned from call_ptr or callf) cast to void*
'Z'	string cast to void*

Table 5: Type signature encoding for Erlang bindings

4.3 Go language bindings

A Go binding is provided through the `godc` package. Since Go's type system is basically a superset of C's, the type mapping from Go to C is straightforward.

Signature character	accepted Go data types
'v'	no return type
'B'	bool
'c', 'C'	int8, uint8
's', 'S'	int16, uint16
'i', 'I'	int, uint
'j', 'J'	int32, uint32
'l', 'L'	int64, uint64
'f'	float32
'd'	float64
'p', 'Z'	uintptr, unsafe.Pointer

Table 6: Type signature encoding for Go bindings

Note that passing a Go-`string` directly to a C-function expecting a pointer is not directly possible. However, the binding comes with two helper functions, `AllocCString(value string) unsafe.Pointer` and `FreeCString(value unsafe.Pointer)` to help with converting a `string` to an `unsafe.Pointer` which then can be passed to `ArgPointer(value unsafe.Pointer)`. Once you are done with this temporary string, free it using `FreeCString(value unsafe.Pointer)`.

4.4 Python language bindings

The python module `pydc` implements the Python language bindings, namely `load`, `find`, `free`, `call`, `new_callback`, `free_callback`.

Signature character	accepted Python 2 types	accepted Python 3 types
'v'	no return type	no return type
'B'	bool	bool
'c', 'C'	int, string (with single char)	int, string (with single char)
's', 'S'	int	int
'i', 'I'	int	int
'j', 'J'	int	int
'l', 'L'	int, long	int
'f'	float	float
'd'	float	float
'p'	bytearray, int, long, None, (PyCObject, PyCapsule)	bytearray, int, None, (PyCObject, PyCapsul
'Z'	string, unicode, bytearray	string, bytes, bytearray

Table 7: Type signature encoding for Python bindings

This is a very brief description that omits many details. For more, refer to the README.txt file of the binding.

4.5 R language bindings

The R package `rdyncall` implements the R langugae bindings providing the function `.dyncall()` .

Signature character	accepted R data types
'v'	no return type
'B'	coerced to logical vector, first item
'c'	coerced to integer vector, first item truncated char
'C'	coerced to integer vector, first item truncated to unsigned char
's'	coerced to integer vector, first item truncated to short
'S'	coerced to integer vector, first item truncated to unsigned short
'i'	coerced to integer vector, first item
'I'	coerced to integer vector, first item casted to unsigned int
'j'	coerced to integer vector, first item
'J'	coerced to integer vector, first item casted to unsigned long
'l'	coerced to numeric, first item casted to long long
'L'	coerced to numeric, first item casted to unsigned long long
'f'	coerced to numeric, first item casted to float
'd'	coerced to numeric, first item
'p'	external pointer or coerced to string vector, first item
'Z'	coerced to string vector, first item

Table 8: Type signature encoding for R bindings

Some notes on the R Binding:

- Unsigned 32-bit integers are represented as signed integers in R.
- 64-bit integer types do not exist in R, therefore we use double floats to represent 64-bit integers (using only the 52-bit mantissa part).

4.6 Ruby language bindings

The Ruby gem `rbdc` implements the Ruby language bindings.

Signature character	accepted Ruby data types
'v'	no return type
'B'	TrueClass, FalseClass, NilClass, Fixnum casted to bool
'c', 'C'	Fixnum cast to (unsigned) char
's', 'S'	Fixnum cast to (unsigned) short
'i', 'I'	Fixnum cast to (unsigned) int
'j', 'J'	Fixnum cast to (unsigned) long
'l', 'L'	Fixnum cast to (unsigned) long long
'f'	Float cast to float
'd'	Float cast to double
'p', 'Z'	String cast to void*

Table 9: Type signature encoding for Ruby bindings

5 Library Design

5.1 Design considerations

The *dyncall* library encapsulates function call invocation semantics that depend on the compiler, operating system and architecture. The core library is driven by a function call invocation engine, named *CallVM*, that encapsulates a call stack to foreign functions and manages the following three phases that constitute a truly dynamic function call:

1. Specify the calling convention. Some run-time platforms, such as Microsoft Windows on a 32-bit X86 architecture, even support multiple calling conventions.
2. Specify the function call arguments in a specific order. The interface design dictates a *left to right* order for C and C++ function calls in which the arguments are bound.
3. Specify the target function address, expected return value and invoke the function call.

The calling convention mode entirely depends on the way the foreign function has been compiled and specifies the low-level details on how a function actually expects input parameters (in memory, in registers or both) and how to return its result(s).

6 Developers

6.1 Noteworthy files in the project root

`configure`	pre-make configuration tool (unix-shell)
`configure.bat`	pre-nmake configuration tool (windows batch)
`configure.rc`	pre-mk configuration tool (Plan 9's rc)
`CMakeLists.txt`	top-level project information for CMake
`Makefile`	GNU/BSD makefile (output of ./configure)
`Nmakefile`	MS nmake makefile
`mkfile`	Plan 9 mkfile
`LICENSE`	license information
`README`	quickstart doc
`buildsys/`	build system details and extras
`doc/`	platform specific readme's and manual
`dyncall/`	dyncall library source code
`dyncallback/`	dyncallback library source code
`dynload/`	dynload library source code
`test/`	test suites

6.2 Test suites

plain Simple, identity, mostly unary function calls for all supported return types and calling conventions. This is not an extensive test suite, but a good place to manually test certain cases.

plain_c++ Similar to plain, but for C++ thiscalls (in different forms and fashion, with aggregates, vararg methots, on x86 with GNU and MS calling convention, etc.).

suite All combinations of parameter types and counts are tested on void function calls. A Python script (`mkcase.py`) generates the tests up to an upper MAXARG limit.

suite_floats Based on suite. Test double/float variants with up to 10 arguments.

call_suite_aggrs Tests passing and returning aggregates by value (struct/union and array members). Test cases can be designed in a signature-style format (and random ones generated via a Lua script).

suite_x86win32std All combinations of parameter types and counts are tested on `__stdcall` void function calls. A Python script (`mkcase.py`) generates the tests up to an upper MAXARG limit. This is a x86/Windows only test.

suite_x86win32fast All combinations of parameter types and counts are tested on `__fastcall` (MS or GNU, depending on the build tool) void function calls. A Python script (`mkcase.py`) generates the tests up to an upper MAXARG limit. This is a x86/Windows only test.

ellipsis All combinations of parameter types and counts are tested on void ellipsis (vararg) function calls. A Python script (`mkcase.py`) generates the tests up to an upper MAXARG limit.

suite2 Designed mass test suite for void function calls. Tests individual void functions with a varying count of arguments and type.

suite2_win32std Designed mass test suite for `__stdcall` void function calls. Tests individual void functions with a varying count of arguments and type. This is a x86/Windows only test.

suite2_win32fast Designed mass test suite for `__fastcall` (MS or GNU, depending on the build tool) void function calls. Tests individual void functions with a varying count of arguments and type. This is a x86/Windows only test.

suite3 All combinations of parameter types integer, long long, float and double and counts are tested on void function calls. A script written in Python (`mkcase.py`) generates the tests up to an upper MAXARG limit. This is a modified version of suite.

call_suite General purpose test suite combining aspects from all others suites (usually enough for testing non-callback calls). Test cases can be designed in a signature-style format (and random ones generated via a Lua script).

callf Tests the *formatted call/arg dyncall* C API. Tries to cover all uses of that API (e.g. calling convention mode switches, aggregates, ...) but is not extensive.

malloc_wx Tests *writable and executable memory allocation* used by the *dyncallback* C API.

thunk Tests only the *callback* thunks for the *dyncallback* C API.

callback_plain Simple callback function test (useful for easy debugging of new ports).

callback_suite Mass test suite for callback function testing. Uses random function argument number and type. A Lua script the tests up to a given number of calls and type combinations.

resolv_self Test for dynload library to resolve symbols from application image itself.

7 Epilog

7.1 Stability and security considerations

Since the *dyncall* library doesn't know anything about the called function itself (except its address), no parameter-type validation is done. This means that in order to avoid crashes, data corruption, etc., the user is required to ascertain the number and types of parameters. It is strongly advised to double check the parameter types of every function to be called, and not to call unknown functions at all.

Consider a simple program that issues a call by directly passing some unchecked command line arguments to the call itself, or even worse, by indirectly choosing a library to load and a function to call without verification. Such unchecked input data can quite easily be used to intentionally crash the program or to take over control of the program flow.
If not used with care, programs depending on the *dyncall*, *dyncallback* and *dynload* libraries, can be exploited as arbitrary function call dispatchers through manipulation of their input data. Successful exploits of badly formed programs like outlined above can be misused as powerful tools for a wide variety of malicious attacks, ...

7.2 Embedding

The *dyncall* library strives to have a minimal set of dependencies, meaning no required runtime dependencies and usually only the necessary tools to build the library as build-time dependencies, like a compiler and assembler, linker, etc.. The library uses some heap-memory to store the CallVM and uses by default the platform's `malloc()` and `free()` calls. However, providing custom `dcAllocMem` and `dcFreeMem` C-preprocessor definitions will override the default behaviour. See `dyncall/dyncall_alloc.h` for details.

7.3 Multi-threading

The *dyncall* library is thread-safe and reentrant, by means that it works correctly during execution of multiple threads if, and only if there is at most a single thread pushing arguments to one CallVM. Since there's no limitation on the number of created CallVM objects, it is recommended to keep a copy per thread if mutliple threads make use of *dyncall* in parallel. Invoking the call should always be thread-safe, however, whether the called function is thread-safe is up to the programmer to verify, of course.

7.4 Supported types

Currently, the *dyncall* library supports all of ANSI C's integer, floating point and pointer types as function call arguments and return values. Additionally, C++'s `bool` and C99's `_Bool` types are supported across all supported platforms. Due to the still rare and often incomplete support of the `long double` type on various platforms, the latter is currently not officially supported. Also, `_Complex` is currently not supported.

Passing or returning aggregates (struct, union) by value is supported, but only on a limited set of platforms (check if the macro DC__Feature_AggrByVal is defined).

7.5 Roadmap

The *dyncall* library should be extended by a wide variety of other calling conventions and ported to other and more esoteric platforms. With its low memory footprint it surely comes in handy on embedded systems. Furthermore, the authors plan to provide more scripting language bindings, examples, and other projects based on *dyncall*.
Besides *dyncall* and *dyncallback*, the *dynload* library needs to be extended with support for other shared library formats (e.g. AmigaOS .library or GEM [45] files).

7.6 Related libraries

Besides the *dyncall* library, there are other free and open projects with similar goals. The most noteworthy libraries are libffi [46], C/Invoke [47] and libffcall [48].

A Dyncall C library API

See the dyncall(3) manpage for more information.

B Dyncallback C library API

See the dyncallback(3) manpage for more information.

C Dynload C library API

See the dynload(3) manpage for more information.

D Calling Conventions

Before we go any further...

It is important to understand that this section isn't a general purpose description of the present calling conventions. It merely explains the calling conventions **for the parameter/return types supported by dyncall** (not for e.g. unsupported types like SIMD data types (__m64, __m128, __m128i, __m128d), etc.).
We strongly advise the reader not to use this document as a general purpose calling convention reference.

D.1 x86 Calling Conventions

Overview

On this processor, a word is defined to be 16 bits in size, a dword 32 bits and a qword 64 bits.

There are numerous different calling conventions on the x86 processor architecture, like cdecl [8], MS fastcall [10], GNU fastcall [11], Borland fastcall [12], Watcom fastcall [13], Win32 stdcall [9], MS thiscall [14], GNU thiscall [15], the pascal calling convention [16] and a cdecl-like version for Plan9 [17] (dubbed plan9call by us), etc.

Name	# of regs for params	# regs to # preserve	push order	cleanup by	64bit args via regs?
cdecl	0	4	←	caller	-
MS fastcall	2	4	←	callee	Y
GNU fastcall	2	4	←	callee	N
Borland fastcall	3	4	→	callee	N
Watcom fastcall	4	2-6	←	callee	N
win32 stdcall	0	4	←	callee	-
MS thiscall	1	4	←	callee	N
GNU thiscall	0	4	←	caller	-
pascal	0	4	→	callee	-
plan9call	0	0	←	caller	-

Table 10: short x86 calling convention comparison

dyncall support

Currently cdecl, stdcall, fastcall (MS and GNU), thiscall (MS and GNU) and plan9call are supported. *Dyncall* can also be used to issue syscalls on Linux and *BSD by using the syscall number as target parameter and selecting the correct mode.

D.1.1 cdecl

Registers and register usage

Name	Brief description
eax	scratch, return value
ebx	preserve
ecx	scratch
edx	scratch, return value
esi	preserve
edi	preserve
ebp	preserve
esp	stack pointer
st0	scratch, floating point return value
st1-st7	scratch

Table 11: Register usage on x86 cdecl calling convention

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- all arguments are pushed onto the stack (as dwords)
- arguments $>$ 64 bits are pushed as a sequence of dwords
- aggregates (structs, unions) are pushed as a sequence of dwords
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

Return values

- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning via the stack), and callee writes return value to this space; the ptr to the aggregate is returned in eax
- return values of pointer or integral type ($<=$ 32 bits) are returned via the eax register
- integers and aggregates (structs, unions) $>$ 32 and $<=$ 64 bits are returned via the eax and edx registers
- return values $>$ 64 bits (e.g. aggregates) are returned by the caller allocating the space and passing a pointer to the callee as a new, implicit first parameter (this means, on the stack)
- floating point types are returned via the st0 register (except on Minix, where they are returned as integers are)

Stack layout

Stack directly after function prolog:

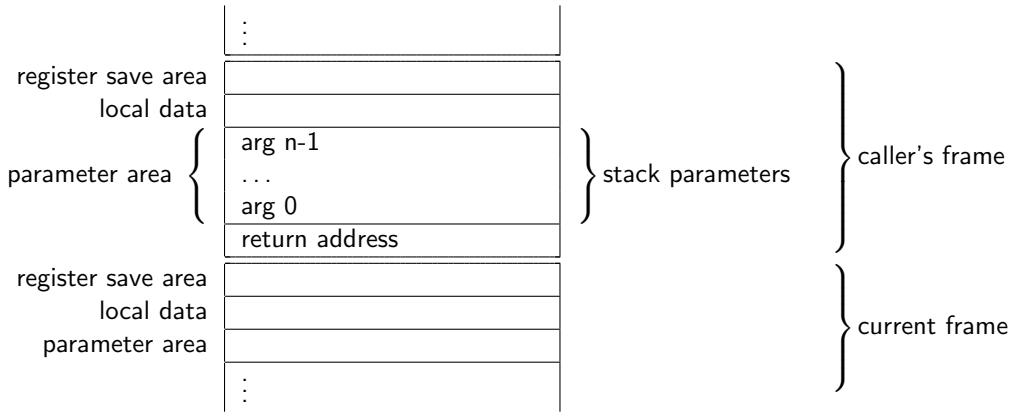


Figure 1: Stack layout on x86 cdecl calling convention

D.1.2 MS fastcall

Registers and register usage

Name	Brief description
eax	scratch, return value
ebx	preserve
ecx	scratch, parameter 0
edx	scratch, parameter 1, return value
esi	preserve
edi	preserve
ebp	preserve
esp	stack pointer
st0	scratch, floating point return value
st1-st7	scratch

Table 12: Register usage on x86 fastcall (MS) calling convention

Parameter passing

- stack parameter order: right-to-left
- called function cleans up the stack
- first two integers/pointers (<= 32bit) are passed via ecx and edx (even if preceded by other arguments)
- if first argument is a 64 bit integer, it is passed via ecx and edx
- all other parameters are pushed onto the stack (as dwords)
- arguments > 64 bits are pushed as a sequence of dwords
- aggregates (structs, unions) are pushed as a sequence of dwords, but are never split between registers and stack (if registers are still available and aggregate doesn't fit entirely into ecx and edx, it is passed via the stack and remaining registers are free for subsequent arguments)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

Return values

- return values of pointer or integral type, as well as aggregates (structs, unions) $<= 64$ are returned via the eax and edx registers
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning via ecx), and callee writes return value to this space; the ptr to the aggregate is returned in eax
- return values > 64 bits (e.g. aggregates) are returned by the caller allocating the space and passing a pointer to the callee as a new, implicit first parameter (always via the stack, never via a register)
- floating point types are returned via the st0 register

Stack layout

Stack directly after function prolog:

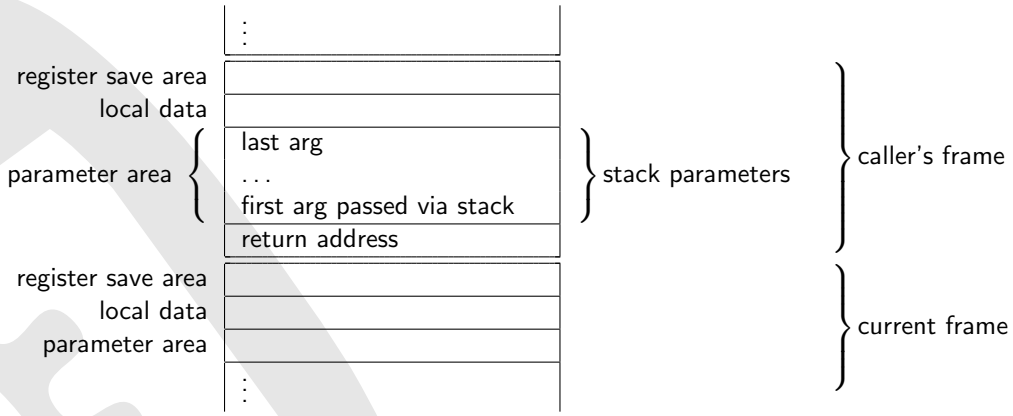


Figure 2: Stack layout on x86 fastcall (MS) calling convention

D.1.3 GNU fastcall

Registers and register usage

Name	Brief description
eax	scratch, return value
ebx	preserve
ecx	scratch, parameter 0
edx	scratch, parameter 1, return value
esi	preserve
edi	preserve
ebp	preserve
esp	stack pointer
st0	scratch, floating point return value
st1-st7	scratch

Table 13: Register usage on x86 fastcall (GNU) calling convention

Parameter passing

- stack parameter order: right-to-left
- called function cleans up the stack
- first two integers/pointers (<= 32bit) are passed via ecx and edx (even if preceded by other arguments)
- arguments > 32 bits are pushed onto the stack as a sequence of dwords (never passed via registers, any respective register is skipped and not used for subsequent args)
- all other parameters are pushed onto the stack (as dwords)
- aggregates (structs, unions) are pushed as a sequence of dwords, and never passed via registers (no matter their size, any respective register is skipped and not used for subsequent args)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- varargs are always passed via the stack

Return values

- return values of pointer or integral type (<= 32 bits) are returned via the eax register
- integers > 32 and <= 64 bits are returned via the eax and edx registers
- aggregates (structs, unions) of any size are returned by the caller allocating the space and passing a pointer to the callee as a new, implicit first parameter (always via ecx), that same pointer is returned in eax
- floating point types are returned via the st0

Stack layout

Stack directly after function prolog:

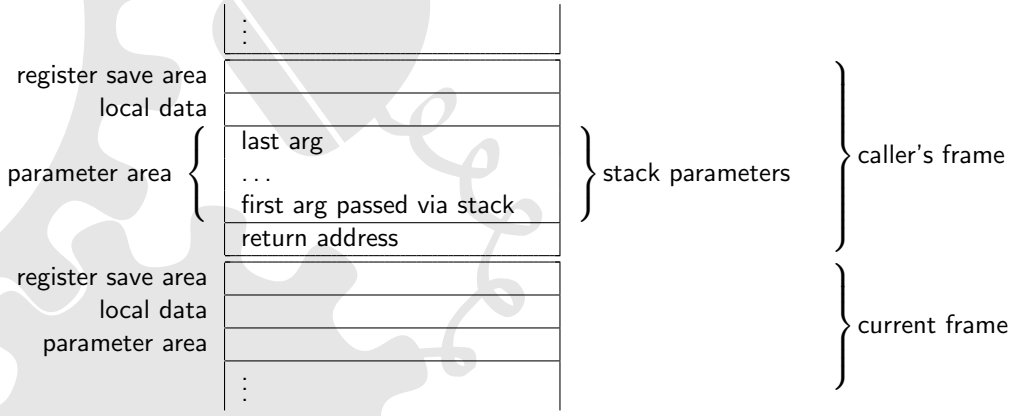


Figure 3: Stack layout on x86 fastcall (GNU) calling convention

D.1.4 Borland fastcall

Also called **register convention** by Borland.

Registers and register usage

Name	Brief description
eax	scratch, parameter 0, return value
ebx	preserve
ecx	scratch, parameter 2
edx	scratch, parameter 1, return value
esi	preserve
edi	preserve
ebp	preserve
esp	stack pointer
st0	scratch, floating point return value
st1-st7	scratch

Table 14: Register usage on x86 fastcall (Borland) calling convention

Parameter passing

- stack parameter order: left-to-right
- called function cleans up the stack
- first three integers/pointers (with exception of method pointers) ($<=$ 32bit) are passed via eax, ecx and edx (preceding or interleaved arguments that are not passed via registers are pushed onto the stack)
- arguments $>$ 32 bits are passed as a pointer to the value
- aggregates (structs, unions) are pushed as a sequence of dwords, and never passed via registers (no matter their size)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- varargs are always passed via the stack
- all other parameters are pushed onto the stack
- the direction flag is clear on entry and must be returned clear

Return values

- return values of pointer or integral type ($<=$ 32 bits) are returned via the eax register
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning via ecx), and callee writes return value to this space; the ptr to the aggregate is returned in eax
- integers and aggregates (structs, unions) $>$ 32 and $<=$ 64 bits are returned via the eax and edx registers
- floating point types are returned via the st0 register

- return values $>$ 32 bits (e.g. aggregates, long long, ...) are returned by the caller allocating the space and passing a pointer to the callee as a new, implicit **last** parameter

Stack layout

Stack directly after function prolog:

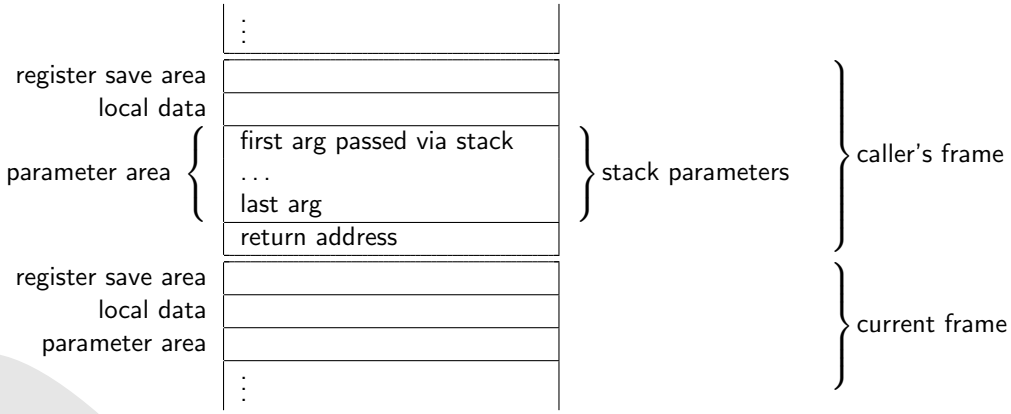


Figure 4: Stack layout on x86 fastcall (Borland) calling convention

D.1.5 Watcom fastcall

Registers and register usage

Name	Brief description
eax	scratch, parameter 0, return value
ebx	scratch when used for parameter, otherwise preserve, parameter 2
ecx	scratch when used for parameter, otherwise preserve, parameter 3
edx	scratch when used for parameter, otherwise preserve, parameter 1, return value
esi	scratch when used for return pointer, otherwise preserve
edi	preserve
ebp	preserve
esp	stack pointer
st0	scratch, floating point return value
st1-st7	scratch

Table 15: Register usage on x86 fastcall (Watcom) calling convention

Parameter passing

- stack parameter order: right-to-left
- called function cleans up the stack
- first four integers/pointers (<= 32bit) are passed via eax, edx, ebx and ecx (even if preceded by other arguments)
- arguments > 32 bits, as well as all subsequent arguments, are passed via the stack
- aggregates (structs, unions) are passed as a pointer to the aggregate (a copy, if needed, to guarantee by-value semantics)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- all other parameters are pushed onto the stack

Return values

- return values of pointer or integral type (<= 32 bits) are returned via the eax register
- integers > 32 bits and <= 64 bits are returned via the eax and edx registers
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee via esi, and callee writes return value to this space; the ptr to the aggregate is returned in eax
- aggregates (structs, unions) <= 32 bits are returned in eax
- aggregates (structs, unions) > 32 bits are returned by the caller allocating the space and passing a pointer to the callee via esi, that same pointer is returned in eax

Stack layout

Stack directly after function prolog:

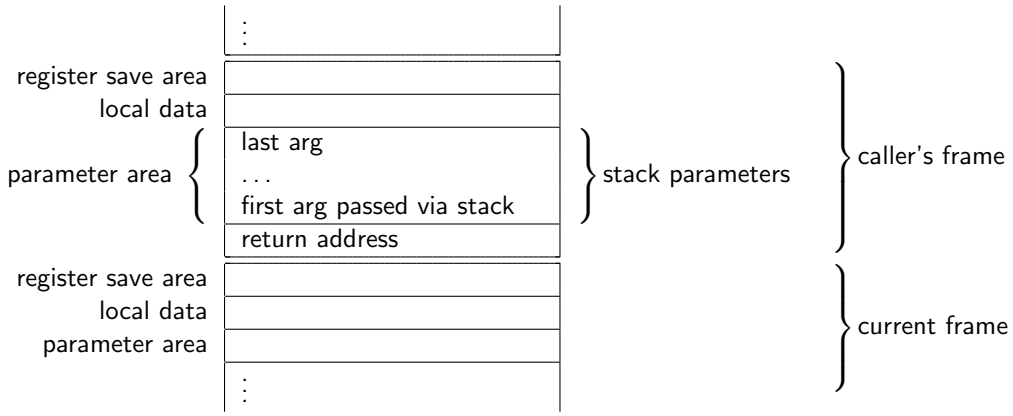


Figure 5: Stack layout on x86 fastcall (Watcom) calling convention

D.1.6 win32 stdcall

Registers and register usage

Name	Brief description
eax	scratch, return value
ebx	preserve
ecx	scratch
edx	scratch, return value
esi	preserve
edi	preserve
ebp	preserve
esp	stack pointer
st0	scratch, floating point return value
st1-st7	scratch

Table 16: Register usage on x86 stdcall calling convention

Parameter passing

- stack parameter order: right-to-left
- called function cleans up the stack
- all parameters are pushed onto the stack (as dwords)
- arguments $>$ 64 bits are pushed as a sequence of dwords
- aggregates (structs, unions) are pushed as a sequence of dwords
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- stack is usually 4 byte aligned (GCC $>=$ 3.x seems to use a 16byte alignment)
- the direction flag is clear on entry and must be returned clear

Return values

- return values of pointer or integral type ($<=$ 32 bits) are returned via the eax register
- integers $>$ 32 and $<=$ 64 bits are returned via the eax and edx registers
- for aggregates and integer return values $>$ 64 bits, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning via stack), and callee writes return value to this space; the ptr to the aggregate is returned in eax
- floating point types are returned via the st0 register

Stack layout

Stack directly after function prolog:

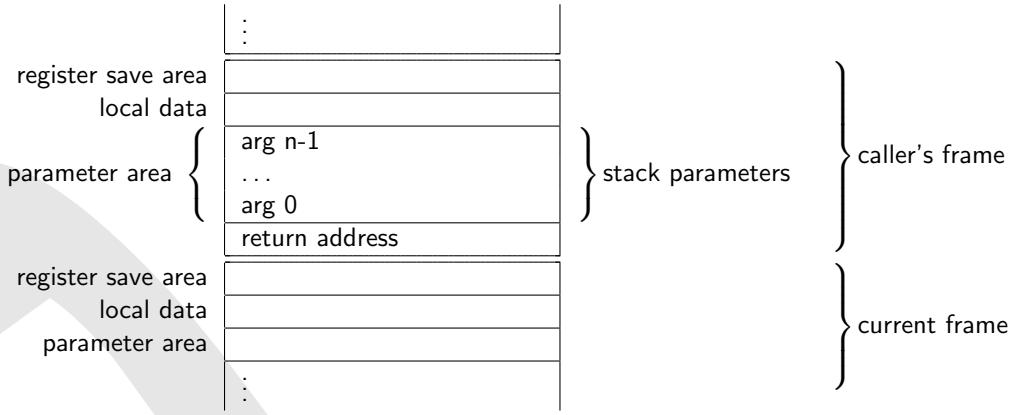


Figure 6: Stack layout on x86 stdcall calling convention

D.1.7 MS thiscall

Registers and register usage

Name	Brief description
eax	scratch, return value
ebx	preserve
ecx	scratch, parameter 0
edx	scratch, return value
esi	preserve
edi	preserve
ebp	preserve
esp	stack pointer
st0	scratch, floating point return value
st1-st7	scratch

Table 17: Register usage on x86 thiscall (MS) calling convention

Parameter passing

- stack parameter order: right-to-left
- called function cleans up the stack (except for variadic functions where the caller cleans up)
- first parameter (this pointer) is passed via ecx
- all other parameters are pushed onto the stack
- arguments $>$ 64 bits are pushed as a sequence of dwords
- aggregates (structs, unions) are pushed as a sequence of dwords
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

Return values

- return values of pointer or integral type ($<=$ 32 bits) are returned via the eax register
- integers $>$ 32 bits and $<=$ 64 bits are returned via the eax and edx
- aggregates (structs, unions) of any size are returned by the caller allocating the space and passing a pointer to the callee as a new, implicit first parameter, that same pointer is returned in eax
- floating point types are returned via the st0 register

Stack layout

Stack directly after function prolog:

D.1.8 GNU thiscall

This is equivalent to the cdecl calling convention, with the first parameter being the this pointer.

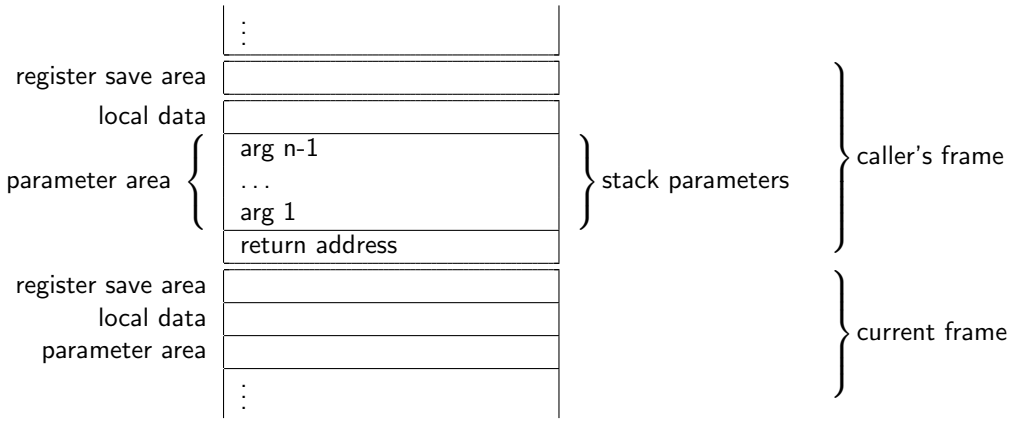


Figure 7: Stack layout on x86 thiscall (MS) calling convention

D.1.9 pascal

The best known uses of the pascal calling convention are the 16 bit OS/2 APIs, Microsoft Windows 3.x and Borland Delphi 1.x. It is a variation of stdcall, however, arguments are passed from left-to-right. Since this calling convention is for 16-bit APIs, it is not discussed in further detail, here.

D.1.10 plan9call

Registers and register usage

Name	Brief description
eax	scratch, return value
ebx	scratch
ecx	scratch
edx	scratch
esi	scratch
edi	scratch
ebp	scratch
esp	stack pointer
st0	scratch, floating point return value
st1-st7	scratch

Table 18: Register usage on x86 plan9call calling convention

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- all parameters are pushed onto the stack
- all parameters are pushed onto the stack (as dwords)
- arguments $>$ 64 bits are pushed as a sequence of dwords
- aggregates (structs, unions) are pushed as a sequence of dwords

- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

Return values

- return values of pointer or integral type (<= 32 bits) are returned via the eax register
- integers > 32 bits and aggregates (structs, unions) of any size are returned by the caller allocating the space and passing a pointer to the callee as a new, implicit first parameter, that same pointer is returned in eax
- floating point types are returned via the st0 register (called F0 in plan9 8a's terms)

Stack layout

Note there is no register save area at all. Stack directly after function prolog:

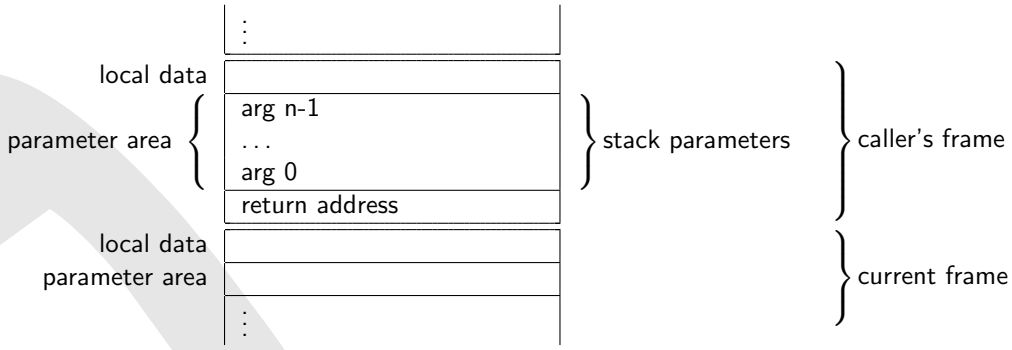


Figure 8: Stack layout on x86 plan9call calling convention

D.1.11 Linux syscalls

Parameter passing

- syscall is issued by triggering *interrupt 80h*
- syscall number is set in eax
- params are passed in the following registers in this order: ebx, ecx, edx, esi, edi, ebp
- for more than six arguments, ebx points to the list of further arguments (not used in practice, as Linux syscalls use a maximum of 5 arguments)
- register eax holds the return value

D.1.12 *BSD syscalls

Parameter passing

- syscall is issued by triggering *interrupt 80h*
- syscall number is set in eax
- params are passed on the stack as with the cdecl calling convention

D.2 x64 Calling Conventions

Overview

The x64 (64bit) architecture designed by AMD is based on Intel's x86 (32bit) architecture, supporting it natively. It is sometimes referred to as x86-64, AMD64, or, cloned by Intel, EM64T or Intel64. On this processor, a word is defined to be 16 bits in size, a dword 32 bits and a qword 64 bits. Note that this is due to historical reasons (terminology didn't change with the introduction of 32 and 64 bit processors).
The x64 calling convention for MS Windows [25] differs from the SystemV x64 calling convention [26] used by Linux/*BSD/... Note that this is not the only difference between these operating systems. The 64 bit programming model in use by 64 bit windows is LLP64, meaning that the C types int and long remain 32 bits in size, whereas long long becomes 64 bits. Under Linux/*BSD/... it's LP64.

Compared to the x86 architecture, the 64 bit versions of the registers are called rax, rbx, etc.. Furthermore, there are eight new general purpose registers r8-r15.

dyncall support

Currently, the MS Windows and System V calling conventions are supported.
Dyncall can also be used to issue syscalls on System V platforms by using the syscall number as target parameter and selecting the correct mode.

D.2.1 MS Windows

Registers and register usage

Name	Brief description
rax	scratch, return value
rbx	permanent
rcx	scratch, parameter 0 if integer or pointer
rdx	scratch, parameter 1 if integer or pointer
rdi	permanent
rsi	permanent
rbp	permanent, may be used as frame pointer
rsp	stack pointer
r8-r9	scratch, parameter 2 and 3 if integer or pointer
r10-r11	scratch, permanent if required by caller (used for syscall/sysret)
r12-r15	permanent
xmm0	scratch, floating point parameter 0, floating point return value
xmm1-xmm3	scratch, floating point parameters 1-3
xmm4-xmm5	scratch, permanent if required by caller
xmm6-xmm15	permanent

Table 19: Register usage on x64 MS Windows platform

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack

- first 4 integer/pointer parameters are passed via rcx, rdx, r8, r9 (from left to right), others are pushed on stack (there is a spill area for the first 4)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- aggregates (structs and unions) $<$ 64 bits are passed like equal-sized integers
- float and double parameters are passed via xmm0l-xmm3l
- first 4 parameters are passed via the correct register depending on the parameter type - with mixed float and int parameters, some registers are left out (e.g. first parameter ends up in rcx or xmm0, second in rdx or xmm1, etc.)
- parameters in registers are right justified
- parameters $<$ 64bits are not zero extended - zero the upper bits contiaining garbage if needed (but they are always passed as a qword)
- parameters $>$ 64 bits are passed by via a pointer to a copy (for aggregate types, that caller-allocated memory must be 16-byte aligned)
- if callee takes address of a parameter, first 4 parameters must be dumped (to the reserved space on the stack) - for floating point parameters, value must be stored in integer AND floating point register
- caller cleans up the stack, not the callee (like cdecl)
- stack is always 16byte aligned - since return address is 64 bits in size, stacks with an odd number of parameters are already aligned
- ellipsis calls take floating point values in int and float registers (single precision floats are promoted to double precision as required by ellipsis calls)
- if size of parameters $>$ 1 page of memory (usually between 4k and 64k), chkstk must be called

Return values

- return values of pointer, integral or aggregate (structs and unions) type ($<=$ 64 bits) are returned via the rax register
- floating point types are returned via the xmm0 register
- for any other type $>$ 64 bits (or for *non-trivial* C++ aggregates of any size), a hidden first parameter, with an address to the return value is passed (for C++ thiscalls it is passed as **second** parameter, after the this pointer)

Stack layout

Stack frame is always 16-byte aligned. Stack directly after function prolog:

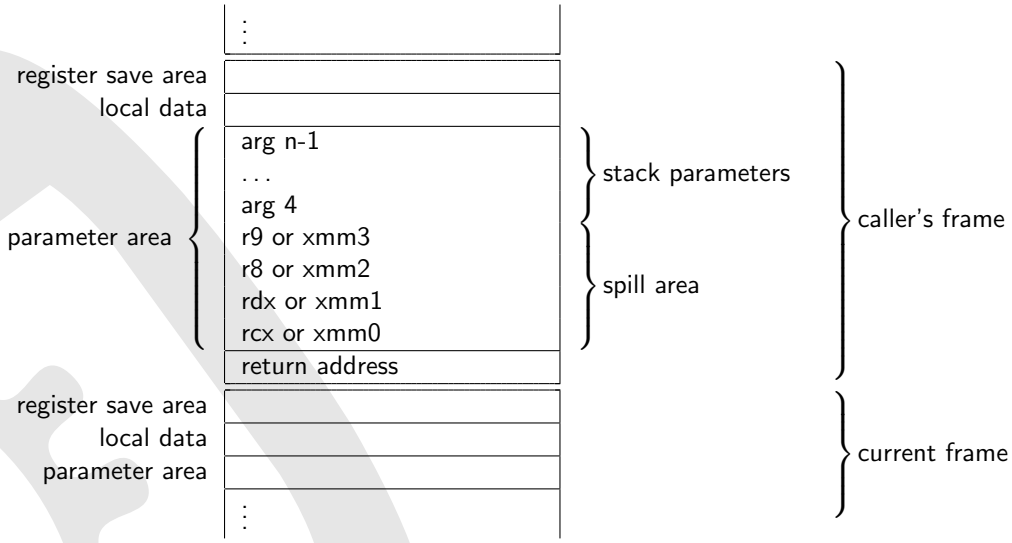


Figure 9: Stack layout on x64 Microsoft platform

D.2.2 System V (Linux / *BSD / MacOS X)

Registers and register usage

Name	Brief description
rax	scratch, return value, special use for varargs (in al, see below)
rbx	permanent
rcx	scratch, parameter 3 if integer or pointer
rdx	scratch, parameter 2 if integer or pointer, return value
rdi	scratch, parameter 0 if integer or pointer
rsi	scratch, parameter 1 if integer or pointer
rbp	permanent, may be used as frame pointer
rsp	stack pointer
r8-r9	scratch, parameter 4 and 5 if integer or pointer
r10-r11	scratch
r12-r15	permanent
xmm0-xmm1	scratch, floating point parameters 0-1, floating point return value
xmm2-xmm7	scratch, floating point parameters 2-7
xmm8-xmm15	scratch
st0-st1	scratch, 16 byte floating point return value
st2-st7	scratch

Table 20: Register usage on x64 System V (Linux/*BSD)

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- first 6 integer/pointer parameters are passed via rdi, rsi, rdx, rcx, r8, r9
- first 8 floating point parameters <= 64 bits are passed via xmm0l-xmm7l
- parameters in registers are right justified
- parameters that are not passed via registers are pushed onto the stack (with their sizes rounded up to qwords)
- parameters < 64bits are not zero extended - zero the upper bits contiaining garbage if needed (but they are always passed as a qword)
- integer/pointer parameters > 64 bit are passed via 2 registers
- if callee takes address of a parameter, number of used xmm registers is passed silently in al (passed number doesn't need to be exact but an upper bound on the number of used xmm registers)
- aggregates (structs, unions (and arrays within those)) follow a more complicated logic (the following **only considers field types supported by dyncall**):
  - *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
  - aggregates > 16 bytes are always passed entirely via the stack

  - all other aggregates are classified per qword, by looking at all fields occupying all or part of that qword, recursively
    * if any field would be passed via the stack, the entire qword will
    * otherwise, if any field would be passed like an integer/pointer value, the entire qword will
    * otherwise the qword is passed like a floating point value
  - after qword classification, the logic is:
    * if any qword is classified to be passed via the stack, the entire aggregate will
    * if the size of the aggregate is $>$ 2 qwords, it is passed via the stack (except for single floating point values $>$ 128bits)
    * all others are passed qword by qword according to their classification, like individual arguments
    * however, an aggregate is never split between registers and the stack, if it doesn't fit into available registers it is entirely passed via the stack (freeing such registers for subsequent arguments)

- stack is always 16byte aligned - since return address is 64 bits in size, stacks with an odd number of parameters are already aligned
- no spill area is used on stack, iterating over varargs requires a specific va_list implementation

Return values

- return values of pointer or integral type are returned via the rax register (and rdx if needed)
- floating point types are returned via the xmm0 register (and xmm1 if needed)
- aggregates are first classified in the same way as when passing them by value, then:
  - for aggregates that would be passed via the stack (or for *non-trivial* C++ aggregates of any size), a hidden pointer to a non-shared, caller provided space is **passed** as hidden, first argument; this pointer will be returned via rax
  - otherwise, qword by qword is passed, using rax and rdx for integer/pointer qwords, and xmm0 and xmm1 for floating point ones
- floating point values $>$ 64 bits are returned via st0 and st1

Stack layout

Stack frame is always 16-byte aligned. A 128 byte large zone beyond the location pointed to by the stack pointer is referred to as "red zone", considered to be reserved and not be modified by signal or interrupt handlers (useful for temporary data not needed to be preserved across calls, and for optimizations for leaf functions). Stack directly after function prolog:

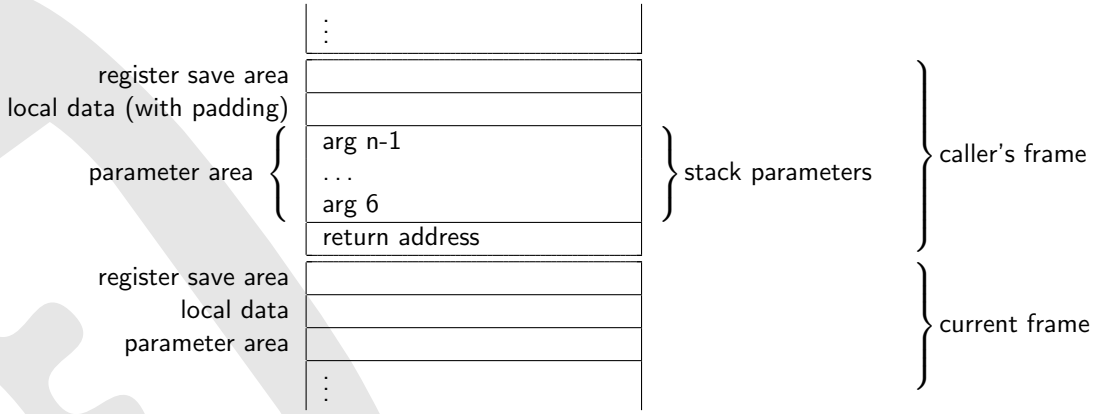


Figure 10: Stack layout on x64 System V (Linux/*BSD)

D.2.3 System V syscalls

Parameter passing

- syscall is issued via the *syscall* instruction
- kernel destroys registers rcx and r11
- syscall number is set in rax
- params are passed in the following registers in this order: rdi, rsi, rdx, rcx, r8, r9
- no stack in use, meaning syscalls are in theory limited to six arguments
- register rax holds the return value (values in between -4095 and -1 indicate errors)

D.3 PowerPC (32bit) Calling Conventions

Overview

- Word size is 32 bits
- Big endian (MSB) and litte endian (LSB) operating modes.
- Processor operates on floats in double precision floating point arithmetc (IEEE-754) values directly (single precision is converted on the fly)
- Apple macos/Mac OS X/Darwin PPC is specified in "Mac OS X ABI Function Call Guide"[32]. It uses Big Endian (MSB)
- Linux PPC 32-bit ABI is specified in "LSB for PPC"[33] which is based on "System V ABI". It uses Big Endian (MSB)
- PowerPC EABI is defined in the "PowerPC Embedded Application Binary Interface 32-Bit Implementation"[34]
- There is also the "PowerOpen ABI"[36], a nearly identical version of it is used in AIX

dyncall support

Dyncall and *dyncallback* are supported for PowerPC (32bit) Big Endian (MSB), for Darwin's and System V's calling convention.
Dyncall can also be used to issue syscalls by using the syscall number as target parameter and selecting the correct mode.

D.3.1 Mac OS X/Darwin

Registers and register usage

Parameter passing

- stack grows down
- stack parameter order: right-to-left
- caller cleans up the stack
- the first 8 integer parameters are passed in registers gpr3-gpr10
- the first 13 floating point parameters are passed in registers fpr1-fpr13
- 64 bit arguments are passed as if they were two 32 bit arguments, without skipping registers for alignment (this means passing half via a register and half via the stack is allowed)
- if a float parameter is passed via a register, gpr registers are skipped for subsequent integer parameters (based on the size of the float - 1 register for single precision and 2 for double precision floating point values)
- the caller pushes subsequent parameters onto the stack
- for every parameter passed via a register, space is reserved in the stack parameter area (in order to spill the parameters if needed - e.g. varargs)

Name	Brief description
gpr0	scratch
gpr1	stack pointer
gpr2	scratch
gpr3,gpr4	return value, parameter 0 and 1 for integer or pointer, scratch
gpr5-gpr10	parameter 2-7 for integer or pointer parameters, scratch
gpr11	preserve
gpr12	branch target for dynamic code generation
gpr13-31	preserve
fpr0	scratch
fpr1	floating point return value, floating point parameter 0 (always double precision)
fpr2-fpr13	floating point parameters 1-12 (always double precision)
fpr14-fpr31	preserve
v0-v1	scratch
v2-v13	vector parameters
v14-v19	scratch
v20-v31	preserve
lr	link-register, scratch
ctr	count-register, scratch
cr0-cr7	conditional register fields, each 4-bit wide (cr0-cr1 and cr5-cr7 are scratch)

Table 21: Register usage on Darwin PowerPC 32-Bit

- ellipsis calls take floating point values in int and float registers (single precision floats are promoted to double precision as required by ellipsis calls)
- all nonvector parameters are aligned on 4-byte boundaries
- vector parameters are aligned on 16-byte boundaries
- composite parameters with size of 1 or 2 bytes occupy low-order bytes of their 4-byte area. INCONSISTENT with other 32-bit PPC binary interfaces. In AIX and mac OS 9, padding bytes always follow the data structure
- composite parameters 3 bytes or larger in size occupy high-order bytes
- integer parameters $<$ 32 bit are right-justified (meaning occupy higher-address bytes) in their 4-byte slot on the stack, requiring extra-care for big-endian targets
- aggregates (struct, union) with only one (non-aggregate / non-array) field are passed as if the field itself would be passed
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- all other aggregates are passed as a sequence of words (like integer parameters)

Return values

- return values of integer $<=$ 32bit or pointer type use gpr3
- 64 bit integers use gpr3 and gpr4 (hiword in gpr3, loword in gpr4)
- floating point values are returned via fpr1

- for all aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in gpr3), and callee writes return value to this space; the ptr to the aggregate is returned in gpr3

Stack layout

Stack frame is always 16-byte aligned. Prolog opens frame with additional, fixed space for a linkage area, to hold a number of values (not all of them are required to be saved, though). Stack directly after function prolog:

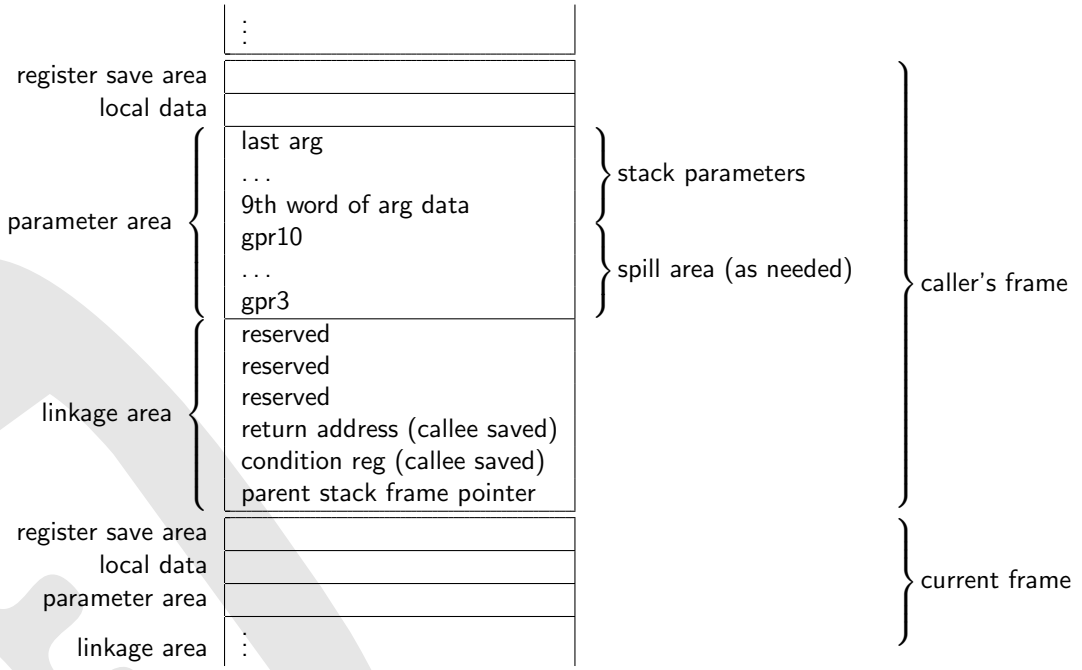


Figure 11: Stack layout on ppc32 Darwin

D.3.2 System V PPC 32-bit

Status

Registers and register usage

Name	Brief description
r0	scratch
r1	stack pointer, preserve
r2	system-reserved
r3-r4	parameter passing and return value, scratch
r5-r10	parameter passing, scratch
r11-r12	scratch
r13	small data area pointer register
r14-r30	local variables, preserve
r31	used for local variables or *environment pointer*, preserve
f0	scratch
f1	parameter passing and return value, scratch
f2-f8	parameter passing, scratch
f9-13	scratch
f14-f31	local variables, preserve
cr0-cr7	conditional register fields, each 4-bit wide (cr0-cr1 and cr5-cr7 are scratch)
lr	link register, scratch
ctr	count register, scratch
xer	fixed-point exception register, scratch
fpscr	floating-point Status and Control Register

Table 22: Register usage on System V ABI PowerPC Processor

Parameter passing

- Stack pointer (r1) is always 16-byte aligned. The EABI differs here - it is 8-byte alignment
- 8 general-purpose registers (r3-r10) for integer and pointer types
- 8 floating-pointer registers (f1-f8) for float (promoted to double) and double types
- Additional arguments are passed on the stack directly after the back-chain and saved return address (8 bytes structure) on the callers stack frame
- 64-bit integer data types are passed in general-purpose registers as a whole in two 32-bit general purpose registers (an odd and an even e.g. r3 and r4), skipping an even integer register or passed on the stack; they are never splitted into a register and stack part
- Ellipsis calls set CR bit 6
- integer parameters < 32 bit are right-justified (meaning occupy high-order bytes) in their 4-byte area, requiring extra-care for big-endian targets
- no spill area is used on stack, iterating over varargs requires a specific va_list implementation
- aggregates (struct, union) and types > 64 bits are passed indirectly, as a pointer to the data (or a copy of it, if necessary to avoid modification)

- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

Return values

- 32-bit integers use register r3, 64-bit use registers r3 and r4 (hiword in r3, loword in r4)
- floating-point values are returned using register f1
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in gpr3), and callee writes return value to this space; the ptr to the aggregate is returned in gpr3
- aggregates (struct, union) $<=$ 64 bits use gpr3 and gpr4
- for all other aggregates and types $>$ 64 bits, a secret first parameter with an address to a caller allocated space is passed to the function (in gpr3), which is written to by the callee

Stack layout

Stack frame is always 16-byte aligned. Stack directly after function prolog:

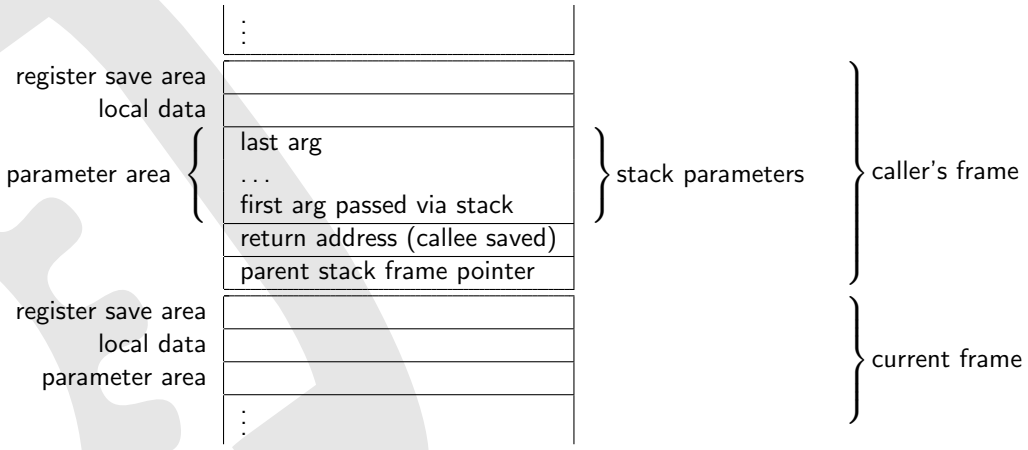


Figure 12: Stack layout on System V ABI for PowerPC 32-bit calling convention

D.3.3 System V PPC 32-bit / Linux Standard Base version

This is in essence the same as the System V PPC 32-bit calling convention, but differs for aggregate return values:

- for all aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in gpr3), and callee writes return value to this space; the ptr to the aggregate is returned in gpr3

D.3.4 System V syscalls

Parameter passing

- syscall is issued via the *sc* instruction
- kernel destroys registers r13
- syscall number is set in r0
- params are passed in registers r3 through r10
- no stack in use, meaning syscalls are in theory limited to eight arguments
- register r3 holds the return value, overflow flag in conditional register cr0 signals errors in syscall

D.4 PowerPC (64bit) Calling Conventions

Overview

- Word size is 32 bits for historical reasons
- Doublework size is 64 bits.
- Big endian (MSB) and litte endian (LSB) operating modes.
- Apple Mac OS X/Darwin PPC is specified in "Mac OS X ABI Function Call Guide"[32]. It uses Big Endian (MSB).
- Linux PPC 64-bit ABI is specified in "64-bit PowerPC ELF Application Binary Interface Supplement"[37] which is based on "System V ABI".

dyncall support

Dyncall and *dyncallback* are supported for PowerPC (64bit) Big Endian and Little Endian ELF ABIs on System V systems. Mac OS X is not supported.
Dyncall can also be used to issue syscalls by using the syscall number as target parameter and selecting the correct mode.

D.4.1 PPC64 ELF ABI

Registers and register usage

Name	Brief description
gpr0	scratch
gpr1	stack pointer
gpr2	TOC base ptr (offset table and data for position independent code), scratch
gpr3	return value, parameter 0 for integer or pointer, scratch
gpr4-gpr10	parameter 1-7 for integer or pointer parameters, scratch
gpr11	env pointer if needed, scratch
gpr12	used for exception handling and glink code, scratch
gpr13	used for system thread ID, preserve
gpr14-31	preserve
fpr0	scratch
fpr1-fpr4	floating point return value, floating point parameter 0-3 (always double precision)
fpr5-fpr13	floating point parameters 4-12 (always double precision)
fpr14-fpr31	preserve
v0-v1	scratch
v2-v13	vector parameters
v14-v19	scratch
v20-v31	preserve
lr	link-register, scratch
ctr	count-register, scratch
xer	fixed point exception register, scratch
fpscr	floating point status and control register, scratch
cr0-cr7	conditional register fields, each 4-bit wide (cr0-cr1 and cr5-cr7 are scratch)

Table 23: Register usage on PowerPC 64-Bit ELF ABI

Parameter passing

- stack grows down
- stack parameter order: right-to-left
- caller cleans up the stack
- stack is always 16 byte aligned
- the stack pointer must be atomically updated (to avoid any timing window in which an interrupt can occur with a partially updated stack), usually with the stdu (store doubleword with update) instruction
- the first 8 integer parameters are passed in registers gpr3-gpr10
- the first 13 floating point parameters are passed in registers fpr1-fpr13
- preserved registers are saved using a defined order (from high to low addresses): fpr* (64bit aligned), gpr*, VRSAVE save word (32 bits), padding for alignment (4 or 12 bytes), v* (128bit aligned)
- if a floating point parameter is passed via a register, a gpr registers is skipped for subsequent integer parameters
- the caller pushes subsequent parameters onto the stack
- single precision floating point values use the second word in a doubleword
- a quad precision floating point argument is passed as two consecutive double precision ones
- integer types $<$ 64 bit are sign or zero extended and use a doubleword
- ellipsis calls take floating point values in int and float registers (single precision floats are promoted to double precision as required by ellipsis calls)
- space for all potential gpr* register passed arguments is reserved in the stack parameter area (in order to spill the parameters if needed - e.g. varargs), meaning a minimum of 64 bytes to hold gpr3-gpr10
- all nonvector parameters are aligned on 8-byte boundaries
- vector parameters are aligned on 16-byte boundaries
- integer parameters $<$ 64 bit are right-justified (meaning occupy higher-address bytes) in their 8-byte slot on the stack, requiring extra-care for big-endian targets
- aggregates (struct, union) are passed as a sequence of doublewords (following above rules for doublewords)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

Return values

- return values of integer $<=$ 32bit or pointer type use gpr3 and are zero or sign extended depending on their type
- 64 bit integers use gpr3

- floating point values are returned via fpr1
- for any aggregate (struct, union), the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in gpr3), and callee writes return value to this space; the ptr to the aggregate is returned in gpr3

Stack layout

Stack frame is always 16-byte aligned. Stack directly after function prolog:

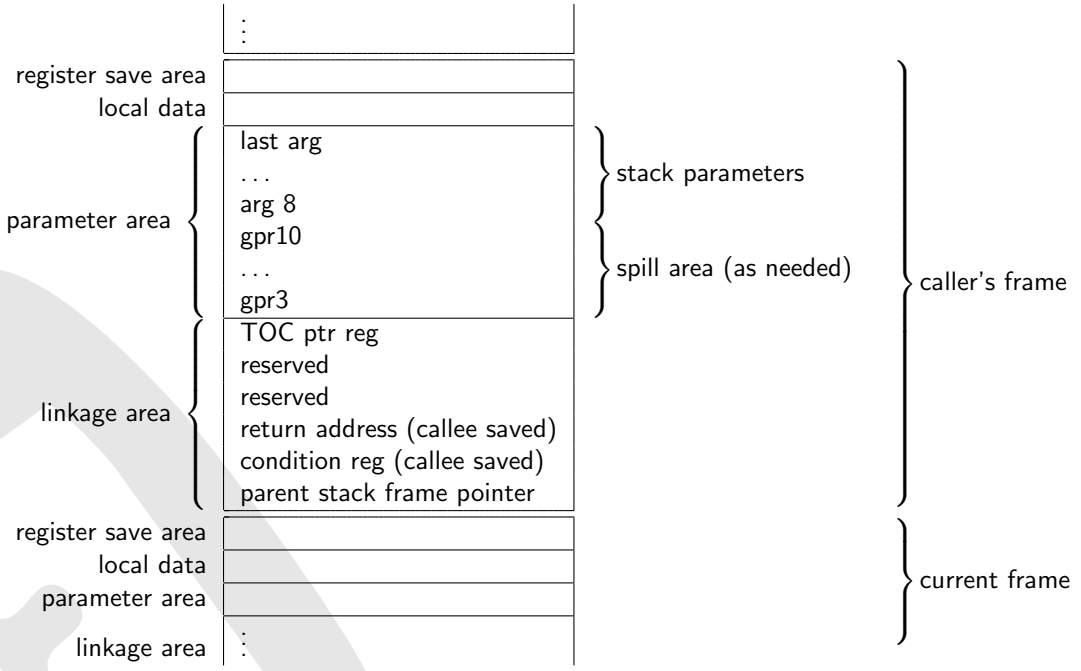


Figure 13: Stack layout on ppc64 ELF ABI

D.4.2 System V syscalls

Parameter passing

- syscall is issued via the *sc* instruction
- kernel destroys registers r13
- syscall number is set in r0
- params are passed in registers r3 through r10
- no stack in use, meaning syscalls are in theory limited to eight arguments
- register r3 holds the return value, overflow flag in conditional register cr0 signals errors in syscall

D.5 ARM32 Calling Conventions

Overview

The ARM32 family of processors is based on the Advanced RISC Machines (ARM) processor architecture (32 bit RISC). The word size is 32 bits (and the programming model is LLP64).
Basically, this family of microprocessors can be run in 2 major modes:

Mode	Description
ARM	32bit instruction set
THUMB	compressed instruction set using 16bit wide instruction encoding

For more details, take a look at the ARM-THUMB Procedure Call Standard (ATPCS) [18], the Procedure Call Standard for the ARM Architecture (AAPCS) [19], as well as Debian's ARM EABI port [23] and hard-float [24] wiki pages.

dyncall support

Currently, the *dyncall* library supports the ARM and THUMB mode of the ARM32 family (ATPCS [18], EABI [23], the ARM hard-float (armhf) [23] varian, as well as Apple's calling convention based on the ATPCS), excluding manually triggered ARM-THUMB interworking calls.
Also supported is armhf, a calling convention with register support to pass floating point numbers. FPA and the VFP (scalar mode) procedure call standards, as well as some instruction sets accelerating DSP and multimedia application like the ARM Jazelle Technology (direct Java bytecode execution, providing acceleration for some bytecodes while calling software code for others), etc., are not supported by the dyncall library.

D.5.1 ATPCS ARM mode

Registers and register usage

In ARM mode, the ARM32 processor has sixteen 32 bit general purpose registers, namely r0-r15:

Name	Alias	Brief description
r0	**a1**	parameter 0, scratch, return value
r1	**a2**	parameter 1, scratch, return value
r2,r3	**a3,a4**	parameters 2 and 3, scratch
r4-r9	**v1-v6**	permanent
r10	**sl**	permanent
r11	**fp**	frame pointer, permanent
r12	**ip**	scratch
r13	**sp**	stack pointer, permanent
r14	**lr**	link register, permanent
r15	**pc**	program counter (note: due to pipeline, r15 points to 2 instructions ahead)

Table 24: Register usage on arm32

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- first four words are passed using r0-r3
- subsequent parameters are pushed onto the stack (in right to left order, such that the stack pointer points to the first of the remaining parameters)
- if the callee takes the address of one of the parameters and uses it to address other parameters (e.g. varargs) it has to copy - in its prolog - the first four words to a reserved stack area adjacent to the other parameters on the stack
- parameters $<=$ 32 bits are passed as 32 bit words
- 64 bit parameters are passed as two 32 bit parts (even partly via the register and partly via the stack, although this doesn't seem to be specified in the ATPCS)
- aggregates (struct, union) are passed by value (after rounding up the size to the nearest multiple of 4), as a sequence of words (splitting across registers and stack is allowed)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- keeping the stack eight-byte aligned can improve memory access performance and is required by LDRD and STRD on ARMv5TE processors which are part of the ARM32 family, so, in order to avoid problems one should always align the stack (tests have shown, that GCC does care about the alignment when using the ellipsis)

Return values

- return values $<=$ 32 bits use r0
- 64 bit return values use r0 and r1
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in r0), and callee writes return value to this space; the ptr to the aggregate is returned in r0
- aggregates (struct, union) $<=$ 32 bits are returned like an integer (in r0)
- aggregates (struct, union) $>$ 32 bits the caller allocates space for the return value on the stack in its frame and passes a pointer to it in r0
- for all other aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in r0), and callee writes return value to this space; the ptr to the aggregate is returned in r0

Stack layout

Stack directly after function prolog:

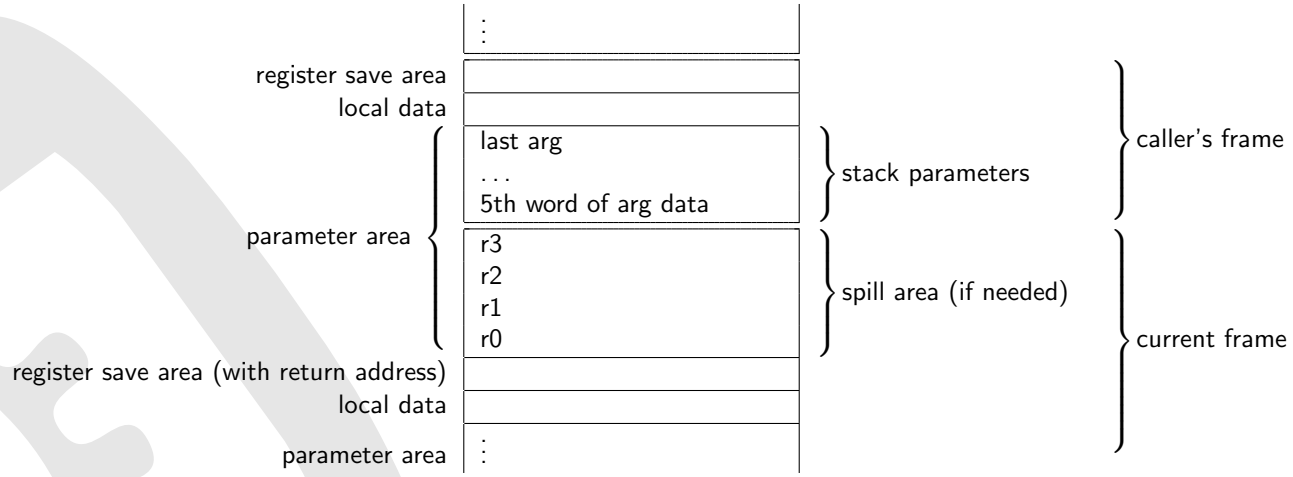


Figure 14: Stack layout on arm32

D.5.2 ATPCS THUMB mode

Status

Registers and register usage

In THUMB mode, the ARM32 processor family supports eight 32 bit general purpose registers r0-r7 and access to high order registers r8-r15:

Name	Alias	Brief description
r0	**a1**	parameter 0, scratch, return value
r1	**a2**	parameter 1, scratch, return value
r2,r3	**a3,a4**	parameters 2 and 3, scratch
r4-r6	**v1-v3**	permanent
r7	**v4**	frame pointer, permanent
r8-r11	**v5-v8**	permanent
r12	**ip**	scratch
r13	**sp**	stack pointer, permanent
r14	**lr**	link register, permanent
r15	**pc**	program counter (note: due to pipeline, r15 points to 2 instructions ahead)

Table 25: Register usage on arm32 thumb mode

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- first four words are passed using r0-r3
- subsequent parameters are pushed onto the stack (in right to left order, such that the stack pointer points to the first of the remaining parameters)
- if the callee takes the address of one of the parameters and uses it to address other parameters (e.g. varargs) it has to copy - in its prolog - the first four words to a reserved stack area adjacent to the other parameters on the stack
- parameters $<=$ 32 bits are passed as 32 bit words
- 64 bit parameters are passed as two 32 bit parts (even partly via the register and partly via the stack, although this doesn't seem to be specified in the ATPCS)
- aggregates (struct, union) are passed by value (after rounding up the size to the nearest multiple of 4), as a sequence of words (splitting across registers and stack is allowed)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- keeping the stack eight-byte aligned can improve memory access performance and is required by LDRD and STRD on ARMv5TE processors which are part of the ARM32 family, so, in order to avoid problems one should always align the stack (tests have shown, that GCC does care about the alignment when using the ellipsis)

Return values

- return values <= 32 bits use r0
- 64 bit return values use r0 and r1
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in r0), and callee writes return value to this space; the ptr to the aggregate is returned in r0
- aggregates (struct, union) <= 32 bits are returned like an integer (in r0)
- aggregates (struct, union) > 32 bits the caller allocates space for the return value on the stack in its frame and passes a pointer to it in r0
- for all other aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in r0), and callee writes return value to this space; the ptr to the aggregate is returned in r0

Stack layout

Stack directly after function prolog:

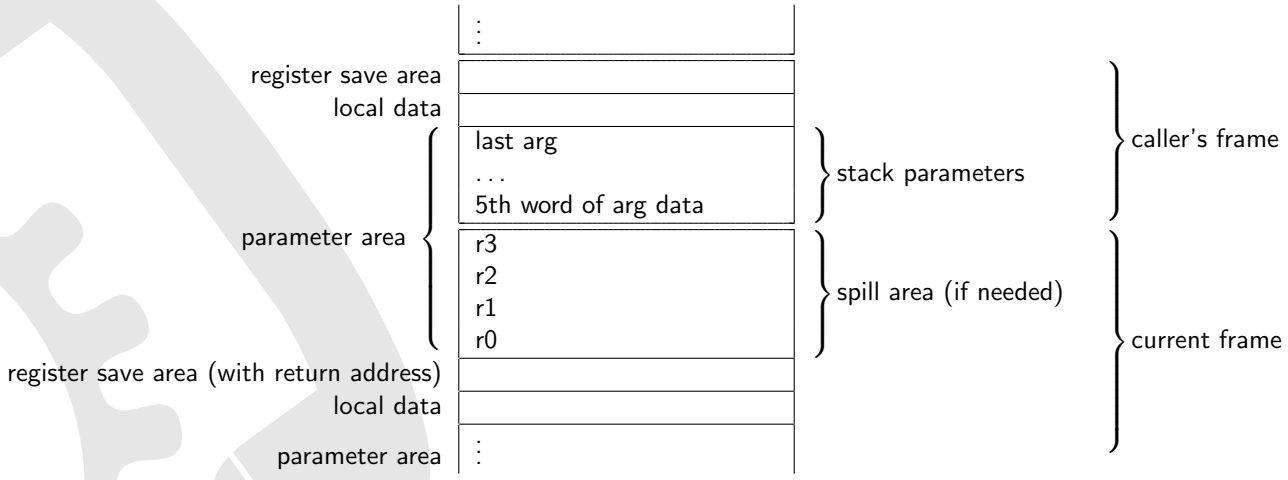


Figure 15: Stack layout on arm32 thumb mode

D.5.3 EABI (ARM and THUMB mode)

The ARM EABI is very similar to the ABI outlined in ARM-THUMB procedure call standard (ATPCS) [18] - however, the EABI requires the stack to be 8-byte aligned at function entries, as well as for 64 bit parameters. The latter are aligned on 8-byte boundaries on the stack and 2-registers for a parameter passed via register. In order to achieve such an alignment, a register might have to be skipped for parameters passed via registers, or 4-bytes on the stack for parameters passed via the stack. Refer to the Debian ARM EABI port wiki for more information [23].

D.5.4 ARM on Apple's iOS (Darwin) Platform (ARM and THUMB mode)

The iOS runs on ARMv6 (iOS 2.0) and ARMv7 (iOS 3.0) architectures. Both, ARM and THUMB are available, code is usually compiled in THUMB mode.

Register usage

Name	Alias	Brief description
r0		parameter 0, scratch, return value
r1		parameter 1, scratch, return value
r2,r3		parameters 2 and 3, scratch
r4-r6		permanent
r7		frame pointer, permanent
r8		permanent
r9		permanent (iOS 2.0) / scratch (since iOS 3.0)
r10-r11		permanent
r12		scratch, intra-procedure scratch register (IP) used by dynamic linker
r13	**sp**	stack pointer, permanent
r14	**lr**	link register, permanent
r15	**pc**	program counter (note: due to pipeline, r15 points to 2 instructions ahead)
cpsr		program status register
d0-d7		scratch, aliases s0-s15, on ARMv7 also as q0-q3; not accessible from Thumb mode on ARMv6
d8-d15		permanent, aliases s16-s31, on ARMv7 also as q4-q7; not accesible from Thumb mode on ARMv6
d16-d31		only available in ARMv7, aliases q8-q15
fpscr		VFP status register

Table 26: Register usage on ARM Apple iOS

Parameter passing and Return values

The ABI is based on the AAPCS but with the following important differences:

- in ARM mode, r7 is used as frame pointer instead of r11 (so both, ARM and THUMB mode use the same convention)
- r9 does not need to be preserved on iOS 3.0 and greater

Stack layout

Stack directly after function prolog:

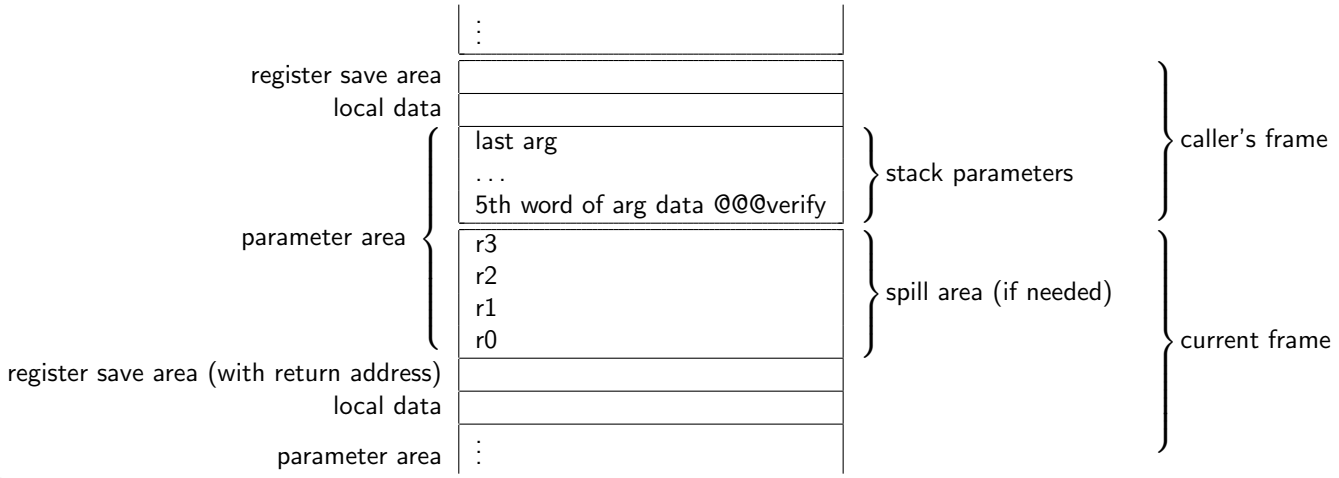


Figure 16: Stack layout on arm32 (Apple)

D.5.5 ARM hard float (armhf)

Most debian-based Linux systems on ARMv7 (or ARMv6 with FPU) platforms use a calling convention referred to as armhf, using 16 32-bit floating point registers of the FPU of the VFPv3-D16 extension to the ARM architecture. Refer to the debian wiki for more information [24].

Code is little-endian, rest is similar to EABI with an 8-byte aligned stack, etc..

Register usage

Name	Alias	Brief description
r0	**a1**	parameter 0, scratch, non floating point return value
r1	**a2**	parameter 1, scratch, non floating point return value
r2,r3	**a3,a4**	parameters 2 and 3, scratch
r4-r9	**v1-v6**	permanent
r10	**sl**	permanent
r11	**fp**	frame pointer, permanent
r12	**ip**	scratch, intra-procedure scratch register (IP) used by dynamic linker
r13	**sp**	stack pointer, permanent
r14	**lr**	link register, permanent
r15	**pc**	program counter (note: due to pipeline, r15 points to 2 instructions ahead)
cpsr		program status register
s0		floating point argument, floating point return value, single precision
d0		floating point argument, floating point return value, double precision, aliases s0-s1
s1-s15		floating point arguments, single precision
d1-d7		aliases s2-s15, floating point arguments, double precision
fpscr		VFP status register

Table 27: Register usage on armhf

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- first four non-floating-point words are passed using r0-r3
- out of those, 64bit parameters use 2 registers, either r0,r1 or r2,r3 (skipped registers are left unused)
- first 16 single-precision, or 8 double-precision arguments are passed via s0-s15 or d0-d7, respectively (note that since s and d registers are aliased, already used ones are skipped)
- subsequent parameters are pushed onto the stack (in right to left order, such that the stack pointer points to the first of the remaining parameters)
- note that as soon one floating point parameter is passed via the stack, subsequent single precision floating point parameters are also pushed onto the stack even if there are still free S* registers
- float and double vararg function parameters (no matter if in ellipsis part of function, or not) are passed like int or long long parameters, vfp registers aren't used
- if the callee takes the address of one of the parameters and uses it to address other parameters (e.g. varargs) it has to copy - in its prolog - the first four words (for first 4 integer arguments) to a reserved stack area adjacent to the other parameters on the stack

- parameters <= 32 bits are passed as 32 bit words
- aggregates (struct, union) with 1 to 4 identical floating-point members (either float or double) are passed field-by-field, except if passed as a vararg
- aggregates that could be passed via floating point register are never split across those and the stack, so if not enough registers are available an aggregate is passed entirely via the stack (implying above rule that any still unused float registers will be skipped for any subsequent arg)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- all other aggregates (struct, union), after rounding up the size to the nearest multiple of 4, are passed as a sequence of dwords, like integers (splitting across registers and stack is allowed)
- callee spills, caller reserves spill area space, though

Return values

- non floating point return values <= 32 bits use r0
- non floating point 64-bit return values use r0 and r1
- floating point return value uses s0 (for float) or d0 (for double), respectively
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in r0), and callee writes return value to this space; the ptr to the aggregate is returned in r0
- aggregates (struct, union) with 1 to 4 identical floating-point members are returned in s0-s3 (for float) or d0-d3 (for double), respectively
- all other aggregates <= 32 bits are returned via r0
- for all other aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in r0), and callee writes return value to this space; the ptr to the aggregate is returned in r0

Stack layout

Stack directly after function prolog:

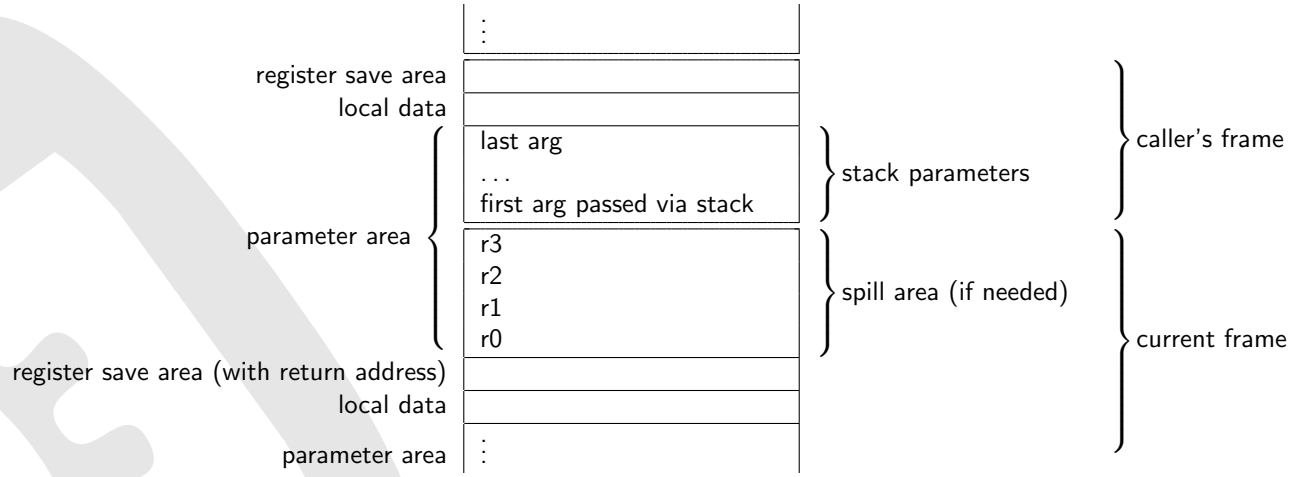


Figure 17: Stack layout on arm32 armhf

D.5.6 Architectures

The ARM architecture family contains several revisions with capabilities and extensions (such as thumb-interworking, more vector registers, ...) The following table sums up the most important properties of the various architecture standards, from a calling convention perspective.

Arch	Platforms	Details
ARMv4		
ARMv4T	ARM 7, ARM 9, Neo FreeRunner (OpenMoko)	
ARMv5	ARM 9E	BLX instruction available
ARMv6		No vector registers available in thumb
ARMv7	iPod touch, iPhone 3GS/4, Raspberry Pi 2	VFP, armhf convention on some platforms
ARMv8	iPhone 6 and higher	64bit support

Table 28: Overview of ARM Architecture, Platforms and Details

D.6 ARM64 Calling Conventions

Overview

ARMv8 introduced the AArch64 calling convention. ARM64 chips can be run in 64 or 32bit mode, but not by the same process. Interworking is only intra-process.
The word size is defined to be 32 bits, a dword 64 bits. Note that this is due to historical reasons (terminology didn't change from ARM32).
For more details, take a look at the Procedure Call Standard for the ARM 64-bit Architecture [20].

dyncall support

The *dyncall* library supports the ARM 64-bit AArch64 PCS ABI, as well as Apple's and Microsoft's conventions which are derived from it, for both, calls and callbacks.

D.6.1 AAPCS64 Calling Convention

Registers and register usage

ARM64 features thirty-one 64 bit general purpose registers, namely **r0**-**r30**, which are referred to as either **x0**-**x30** for 64bit access, or **w0**-**w30** for 32bit access (with upper bits either cleared or sign extended on load).
Also, there is **sp/xzr/wzr**, a register with restricted use, used for the stack pointer in instructions dealing with the stack (**sp**) or a hardware zero register for all other instructions **xzr/wzr**, and **pc**, the program counter. Additionally, there are thirty-two 128 bit registers **v0**-**v31**, to be used as SIMD and floating point registers, referred to as **q0**-**q31**, **d0**-**d31** and **s0**-**s31**, respectively (in contrast to AArch32, those do not overlap multiple narrower registers), depending on their use:

Name	Brief description
x0-**x7**	parameters, scratch, return value
x8	indirect result location pointer
x9-**x15**	scratch
x16	permanent in some cases, can have special function (IP0), see doc
x17	permanent in some cases, can have special function (IP1), see doc
x18	reserved as platform register, advised not to be used for handwritten, portable asm, see doc
x19-**x28**	permanent
x29	permanent, frame pointer
x30	permanent, link register
sp	permanent, stack pointer
pc	program counter
v0-**v7**	scratch, float parameters, return value
v8-**v15**	lower 64 bits are permanent, scratch
v16-**v31**	scratch
xzr	zero register, always zero

Table 29: Register usage on arm64

Parameter passing

- stack parameter order: right-to-left

- caller cleans up the stack
- first 8 integer arguments are passed using x0-x7
- first 8 floating point arguments are passed using d0-d7
- subsequent parameters are pushed onto the stack
- if the callee takes the address of one of the parameters and uses it to address other parameters (e.g. varargs) it has to copy - in its prolog - the first 8 integer and 8 floating-point registers to a reserved stack area adjacent to the other parameters on the stack (only the unnamed integer parameters require saving, though)
- aggregates (struct, union) with 1 to 4 identical floating-point members (either float or double) are passed field-by-field (8-byte aligned if passed via stack), except if passed as a vararg
- other aggregates (struct, union) $>$ 16 bytes in size are passed indirectly, as a pointer to a copy (if needed)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- all other aggregates (struct, union), after rounding up the size to the nearest multiple of 8, are passed as a sequence of dwords, like integers
- aggregates are never split across registers and stack, so if not enough registers are available an aggregated is passed via the stack (for aggregates that would've been passed as floating point values, any still unused float registers will be skipped for any subsequent arg)
- stack is required throughout to be eight-byte aligned

Return values

- integer return values use x0
- floating-point return values use d0
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee via x8, and callee writes return value to this space; the ptr to the aggregate is returned in x0
- aggregates (struct, union) that would be passed via registers if passed as a first param, are returned via those registers
- for aggregates not returnable via registers (e.g. if regs exhausted, or $>$ 16b, ...), the caller allocates space, passes pointer to it to the callee through x8, and callee writes return value to this space (note that this is not a hidden first param, as x8 is not used for passing params); the ptr to the aggregate is returned in x0

Stack layout

Stack directly after function prolog:

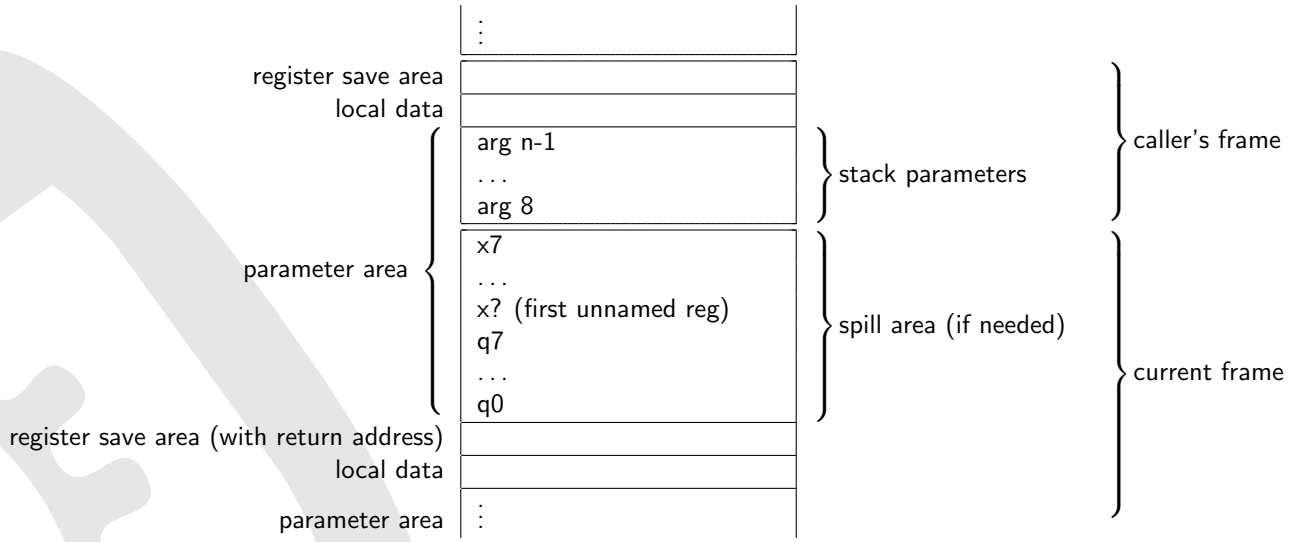


Figure 18: Stack layout on arm64

D.6.2 Apple's ARM64 Function Calling Convention

Overview

Apple's ARM64 calling convention is based on the AAPCS64 standard, however, diverges in some ways. Only the differences are listed here, for more details, take a look at Apple's official documentation [21].

- arguments passed via stack use only the space they need, but are subject to type alignment requirements (which is 1 byte for char and bool, 2 for short, 4 for int and 8 for every other type)
- caller is required to sign and zero-extend arguments smaller than 32bits
- empty aggregates (allowed in C++, but non-standard in C, however compiler extensions exist) as parameters:
  - allowed to be ignored in C
  - allowed to be ignored in C++, if aggregate is trivial, otherwise it's treated as an aggregate with one byte field

D.6.3 Microsoft's ARM64 Function Calling Convention

Overview

Microsoft's ARM64 calling convention is based on the AAPCS64 standard, however, diverges for variadic functions. Only the differences are listed here, for more details, take a look at Microsoft's official documentation [22].

- variadic function calls do not use any SIMD or floating point registers (for fixed and variable args), meaning first 8 params are passed via x0-x7, the rest via the stack
- a function that returns an aggregate indirectly via a pointer passed to via x8 does not seem to be required to put that address in x0 on return (but should be safe to do so)

D.7 MIPS32 Calling Conventions

Overview

Multiple revisions of the MIPS Instruction set exist, namely MIPS I, MIPS II, MIPS III, MIPS IV, MIPS32 and MIPS64. Nowadays, MIPS32 and MIPS64 are the main ones used for 32-bit and 64-bit instruction sets, respectively.
Given MIPS processors are often used for embedded devices, several add-on extensions exist for the MIPS family, for example:

MIPS-3D simple floating-point SIMD instructions dedicated to common 3D tasks.

MDMX (MaDMaX) more extensive integer SIMD instruction set using 64 bit floating-point registers.

MIPS16e adds compression to the instruction stream to make programs take up less room (allegedly a response to the THUMB instruction set of the ARM architecture).

MIPS MT multithreading additions to the system similar to HyperThreading.

Unfortunately, there is actually no such thing as "The MIPS Calling Convention". Many possible conventions are used by many different environments such as *O32*[38], *O64*[39], *N32*[40], *N64*[40], *EABI*[41] and *NUBI*[42].

dyncall support

Currently, dyncall supports for MIPS 32-bit architectures the widely-used O32 calling convention (for all four combinations of big/little-endian, and soft/hard-float targets), as well as EABI (little-endian/hard-float, which is used on the Homebrew SDK for the Playstation Portable). *dyncall* currently does not support MIPS16e (contrary to the like-minded ARM-THUMB, which is supported). Both, calls and callbacks are supported.

D.7.1 MIPS EABI 32-bit Calling Convention

Register usage

Name	Alias	Brief description
$0	**$zero**	hardware zero, scratch
$1	**$at**	assembler temporary, scratch
$2-$3	**$v0-$v1**	integer results, scratch
$4-$11	**$a0-$a7**	integer arguments, or double precision float arguments, scratch
$12-$15,$24	**$t4-$t7,$t8**	integer temporaries, scratch
$25	**$t9**	integer temporary, address of callee for PIC calls (by convention), scratch
$16-$23	**$s0-$s7**	preserve
$26,$27	**$kt0,$kt1**	reserved for kernel
$28	**$gp**	global pointer, preserve
$29	**$sp**	stack pointer, preserve
$30	**$s8/$fp**	frame pointer (some assemblers name it $fp), preserve
$31	**$ra**	return address, preserve
hi, lo		multiply/divide special registers
$f0,$f2		float results, scratch
$f1,$f3,$f4-$f11,$f20-$f23		float temporaries, scratch
$f12-$f19		single precision float arguments, scratch

Table 30: Register usage on MIPS32 EABI calling convention

Parameter passing

- Stack grows down
- Stack parameter order: right-to-left
- Caller cleans up the stack
- first 8 integers (<= 32bit) are passed in registers $a0-$a7
- first 8 single precision floating point arguments are passed in registers $f12-$f19
- 64-bit stack arguments are always aligned to 8 bytes
- 64-bit integers or double precision floats are passed in two general purpose registers starting at an even register number, skipping one odd register
- if either integer or float registers are used up, the stack is used
- if the callee takes the address of one of the parameters and uses it to address other unnamed parameters (e.g. varargs) it has to copy - in its prolog - the the argument registers to a reserved stack area adjacent to the other parameters on the stack (only the unnamed integer parameters require saving, though)
- float registers don't seem to ever need to be saved that way, because floats passed to an ellipsis function are promoted to doubles, which in turn are passed in a? register pairs, so only $a0-$a7 are need to be spilled
- aggregates (struct, union) <= 32bit are passed like an integer
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

- all other aggregates (struct, union) are passed indirectly, as a pointer to a copy (if needed, and for vararg arguments required to be copied by the caller) of the struct

Return values

- results are returned in $v0 (32-bit), $v0 and $v1 (64-bit), $f0 or $f0 and $f2 ($2 \times 32$ bit float e.g. complex)
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in %a0), and callee writes return value to this space; the ptr to the aggregate is returned in %v0
- aggregates (struct, union) $<=$ 64bit are returned like an integer (aligned within the register according to endianness)
- all other aggregates (struct, union) are returned in a space allocated by the caller, with a pointer to it passed as first parameter to the function called (meaning in %a0); the ptr to the aggregate is returned in %v0

Stack layout

Stack directly after function prolog:

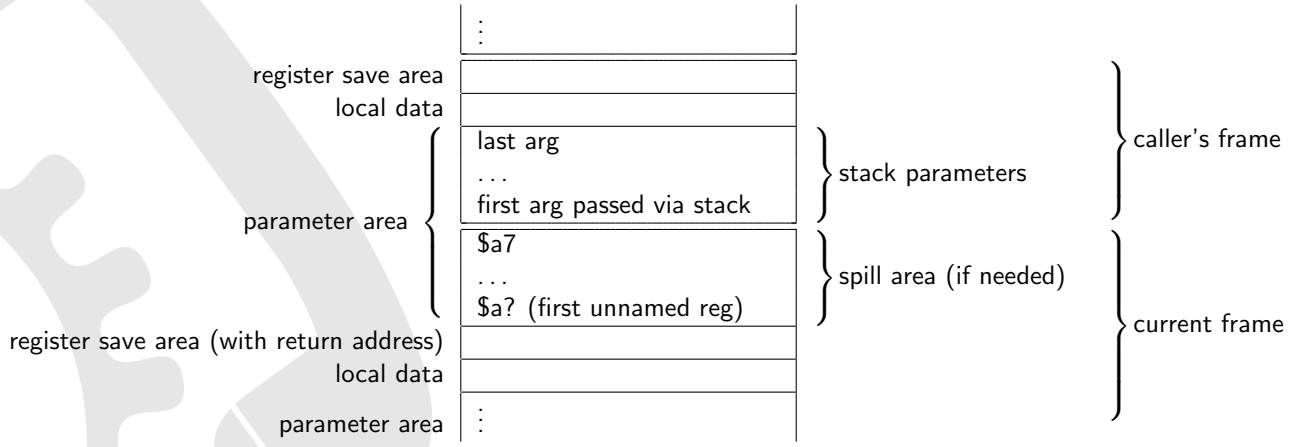


Figure 19: Stack layout on MIPS EABI 32-bit calling convention

D.7.2 MIPS O32 32-bit Calling Convention

Register usage

Name	Alias	Brief description
$0	**$zero**	hardware zero
$1	**$at**	assembler temporary
$2-$3	**$v0-$v1**	return value (only integer on hard-float targets), scratch
$4-$7	**$a0-$a3**	first arguments (only integer on hard-float targets), scratch
$8-$15,$24	**$t0-$t7,$t8**	temporaries, scratch
$25	**$t9**	temporary, holds address of called function for PIC calls (by convention)
$16-$23	**$s0-$s7**	preserved
$26,$27	**$k0,$k1**	reserved for kernel
$28	**$gp**	global pointer, preserved by caller
$29	**$sp**	stack pointer, preserve
$30	**$s8/$fp**	frame pointer (some assemblers name it $fp), preserve
$31	**$ra**	return address, preserve
hi, lo		multiply/divide special registers
$f0-$f3		only on hard-float targets: float return value, scratch
$f4-$f11,$f16-$f19		only on hard-float targets: float temporaries, scratch
$f12-$f15		only on hard-float targets: first floating point arguments, scratch
$f20-$f31		only on hard-float targets: preserved

Table 31: Register usage on MIPS O32 calling convention

Parameter passing

- Stack grows down
- Stack parameter order: right-to-left
- Caller cleans up the stack
- Caller is required to always leave a 16-byte spill area for $a0-$a3 at the end of **its** frame, to be used and spilled to by the callee, if needed
- The different stack areas (local data, register save area, parameter area) are each aligned to 8 bytes
- generally, first four 32bit arguments are passed in registers $a0-$a3, respectively (only on hard-float targets: see below for exceptions if first arg is a float)
- subsequent parameters are passed vie the stack
- 64-bit params passed via registers are passed using either two registers (starting at an even register number, skipping an odd one if necessary), or via the stack using an 8-byte alignment
- only on hard-float targets: if the very first call argument is a float, up to 2 floats or doubles can be passed via $f12 and $f14, respectively, for first and second argument
- only on hard-float targets: if any arguments are passed via float registers, skip $a0-$a3 for subsequent arguments as if the values were passed via them
- only on hard-float targets: note that if the first argument is not a float, but the second, it'll get passed via the $a? registers

- single precision float parameters (32 bit) are right-justified in their 8-byte slot on the stack on big endian targets, as they aren't promoted
- aggregates (struct, union) are passed as a sequence of words like integers, no matter the fields or if hard-float target (splitting across registers and stack is allowed)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

Return values

- results are returned in $v0 and $v1, with $v0 for all values $<$ 64bit (only integer on hard-float targets)
- only on hard-float targets: floating point results are returned in $f0 (32-bit float), or $f0 and $f3 (64bit float)
- aggregates (struct, union) are returned in a space allocated by the caller, with a pointer to it passed as first parameter to the function called (meaning in %a0); the ptr to the aggregate is returned in %v0

Stack layout

Stack directly after function prolog:

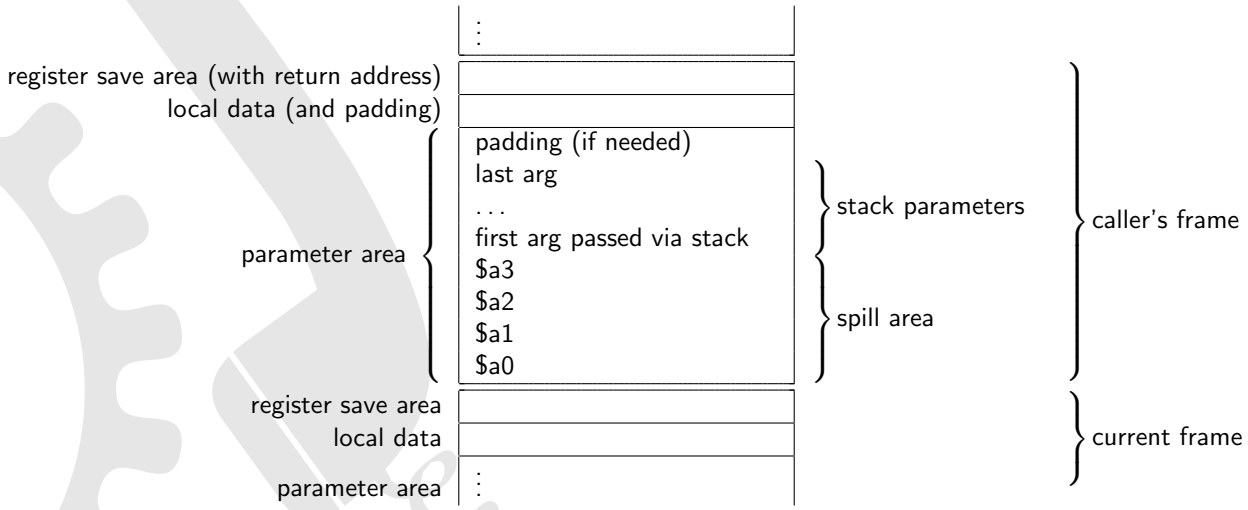


Figure 20: Stack layout on MIPS O32 calling convention

D.8 MIPS64 Calling Conventions

Overview

There are two main ABIs in use for MIPS64 chips, *N64*[40] and *N32*[40]. Both are basically the same, except that N32 uses ILP32 as programming model (32-bit pointers and long integers), whereas N64 uses LP64 (64-bit pointers and long integers). All registers of a MIPS64 chip are considered to be 64-bit wide, even for the N32 calling convention.
The word size is defined to be 32 bits, a dword 64 bits. Note that this is due to historical reasons (terminology didn't change from MIPS32).
Other than that there are correspoding 64-bit versions other MIPS32 ABIs, e.g. the EABI[41] and O64[39].

dyncall support

For MIPS 64-bit machines, dyncall supports the N64 calling conventions for calls and callbacks (for all four combinations of big/little-endian, and soft/hard-float targets). The N32 calling convention might work - it used to, but hasn't been tested, recently.

D.8.1 MIPS N64 Calling Convention

Register usage

Name	Alias	Brief description
$0	**$zero**	hardware zero
$1	**$at**	assembler temporary, scratch
$2-$3	**$v0-$v1**	return value (only integers on hard-float targets), scratch
$4-$11	**$a0-$a7**	first arguments (only integers on hard-float targets), scratch
$12-$15,$24	**$t4-$t7,$t8**	temporaries, scratch
$25	**$t9**	temporary, address callee for all PIC calls (by convention), scratch
$16-$23	**$s0-$s7**	preserve
$26,$27	**$kt0,$kt1**	reserved for kernel
$28	**$gp**	global pointer, preserve
$29	**$sp**	stack pointer, preserve
$30	**$s8**	frame pointer, preserve
$31	**$ra**	return address, preserve
hi, lo		multiply/divide special registers
$f0,$f2		only on hard-float targets: float return values, scratch
$f1,$f3,$f4-$f11		only on hard-float targets: float temporaries, scratch
$f12-$f19		only on hard-float targets: float arguments, scratch
$f20-$f23		only on hard-float targets: float temporaries, scratch
$f24-$f31		only on hard-float targets: preserved

Table 32: Register usage on MIPS N64 calling convention

Parameter passing

- Stack grows down
- Stack parameter order: right-to-left
- Caller cleans up the stack

- generally, first 8 params >= 64-bit are passed via registers
- for hard-float targets: register arguments are passed via $a0-$a7 for integers and $f12-$f19 for floats - with mixed float and int parameters, some registers are left out (e.g. first parameter ends up in $a0 or $f12, second in $a1 or $f13, etc.)
- for soft-float targets: register arguments are passed via $a0-$a7
- subsequent arguments are pushed onto the stack
- all stack entries are 64-bit aligned
- all stack regions are 16-byte aligned
- if the callee takes the address of one of the parameters and uses it to address other unnamed parameters (e.g. varargs) it has to copy - in its prolog - the the argument registers to a reserved stack area adjacent to the other parameters on the stack (only the unnamed integer parameters require saving, though)
- float arguments passed in the variable part of a vararg call are passed like integers, meaning float registers don't ever need to be saved that way, so only $a0-$a7 are need to be spilled
- quad precision float arguments are passed in even-odd register pairs, skipping one register if needed
- integer parameters < 64 bit are right-justified (meaning occupy higher-address bytes) in their 8-byte slot on the stack, requiring extra-care for big-endian targets
- single precision float parameters (32 bit) are left-justified in their 8-byte slot on the stack, but are right justified in fp-registers on big endian targets, as they aren't promoted (actually, official docs says "undecided", but real world implementations seem to use what is described here)
- aggregates (struct, union) are passed as a sequence of dwords in (integer registers and the stack), with the following particularities:
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
  - if a dword happens to be a double precision floating point struct field, it is passed in a floating point register
  - array and union fields are always passed like integers (even if their type is float or double)
  - splitting an argument across registers and the stack is fine

Return values

- results are returned in $v0, and for a second one $v1 is used
- only on hard-float targets: floating point results are returned in $f0 (and $f2 if needed)
- only on hard-float targets: structs with only one or two floating point fields are returned in $f0 (and $f2 if necessary), field-by-field
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in %a0), and callee writes return value to this space; the ptr to the aggregate is returned in %v0
- any other aggregates (struct, union) <= 16 bytes are returned via registers $v0 (and $v1 if necessary), dword-by-dword

- all other aggregates (struct, union) >16 bytes are returned in a space allocated by the caller, with a pointer to it passed as first parameter to the function called (meaning in %a0); the ptr to the aggregate is returned in %v0

Stack layout

Stack directly after function prolog:

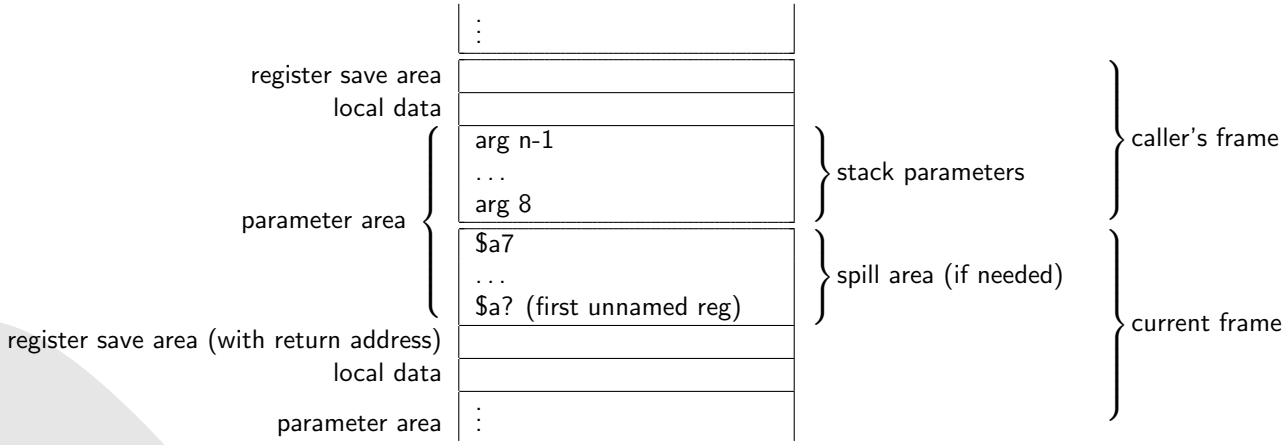


Figure 21: Stack layout on MIPS N64 calling convention

D.8.2 MIPS N32 Calling Convention

Despite what one might think given the name, this is a MIPS 64-bit calling convention. As mentioned in the overview of this chapter, it is nearly identical to the N64 one, the differences being:

- uses ILP32 as programming model instead of LP64
- floating point registers $f20-$f23 are to be preserved

D.9 SPARC Calling Conventions

Overview

The SPARC family of processors is based on the SPARC instruction set architecture, which comes in basically three revisions, V7, V8[29][27] and V9[30][28]. The former two are 32-bit whereas the latter refers to the 64-bit SPARC architecture (see next chapter). SPARC uses big endian byte order. The word size is defined to be 32 bits.

dyncall support

dyncall fully supports the SPARC 32-bit instruction set (V7 and V8), for calls and callbacks.

D.9.1 SPARC (32-bit) Calling Convention

Register usage

- 32 single floating point registers (f0-f31, usable as 8 quad precision q0,q4,q8,...,q28, 16 double precision d0,d2,d4,...,d30)
- 32 32-bit integer/pointer registers out of a bigger (vendor/model dependent) number that are accessible at a time (8 are global ones (g*), whereas the remaining 24 form a register window with 8 input (i*), 8 output (o*) and 8 local (l*) ones)
- calling a function shifts the register window, the old output registers become the new input registers (old local and input ones are not accessible anymore)

Name	Alias	Brief description
%g0	%r0	Read-only, hardwired to 0
%g1-%g7	%r1-%r7	Global
%o0,%o1 and %i0,%i1	%r8,%r9 and %r24,%r25	Output and input argument registers, return value
%o2-%o5 and %i2-%i5	%r10-%r13 and %r26-%r29	Output and input argument registers
%o6 and %i6	%r14 and %r30, %sp and %fp	Stack and frame pointer
%o7 and %i7	%r15 and %r31	Return address (caller writes to o7, callee uses i7)
%l0-%l7	%r16-%r23	preserve
%f0,%f1		Floating point return value
%f2-%f31		scratch

Table 33: Register usage on sparc calling convention

Parameter passing

- stack grows down
- stack parameter order: right-to-left
- caller cleans up the stack
- stack always aligned to 8 bytes
- first 6 integers/pointers and floats are passed independently in registers using %o0-%o5
- for every other argument the stack is used

- all arguments <= 32 bit are passed as 32 bit values
- 64 bit arguments are passed like two consecutive <= 32 bit values (which allows for an argument to be split between the stack and %i5)
- aggregates (struct, union) of any size, as well as quad precision values are passed indirectly as a pointer to a **copy** of the aggregate (like: struct s2 = s; callee(&s2);)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate
- minimum stack size is 64 bytes, b/c stack pointer must always point at enough space to store all %i* and %l* registers, used when running out of register windows
- if needed, register spill area is adjacent to parameters

Return values

- results are expected by caller to be returned in %o0/%o1 (after reg window restore, meaning callee writes to %i0/%i1) for integers
- %f0/%f1 are used for floating point values
- aggregates (struct, union) and quad precision values are returned in a space allocated by the caller, with a pointer to it passed as an **additional**, hidden **stack** parameter (always at %sp+64 for the caller, see below); that pointer is returned in %o0

Stack layout

Stack directly after function prolog:

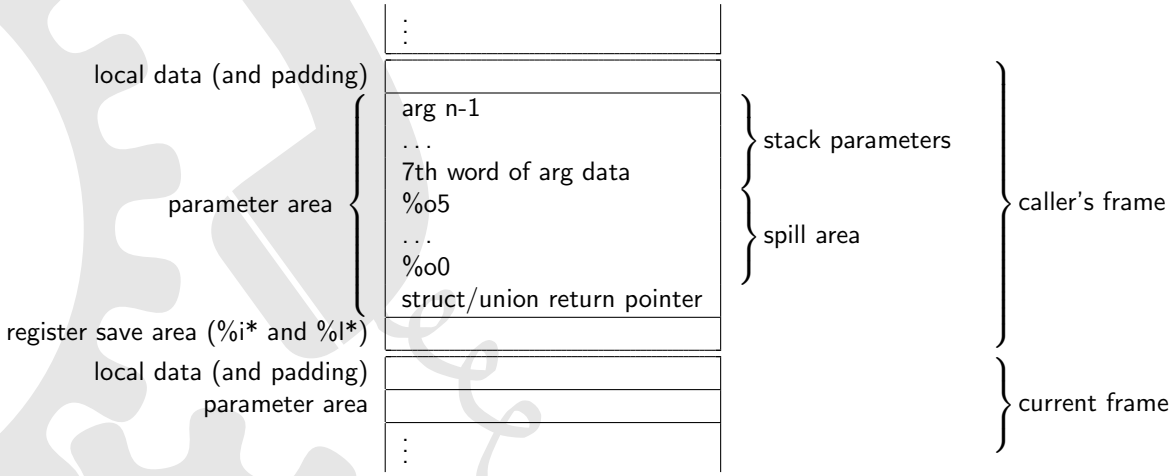


Figure 22: Stack layout on sparc32 calling convention

D.10 SPARC64 Calling Conventions

Overview

The SPARC family of processors is based on the SPARC instruction set architecture, which comes in basically three revisions, V7, V8[29][27][31] and V9[30][28][31]. The former two are 32-bit (see previous chapter) whereas the latter refers to the 64-bit SPARC architecture. SPARC uses big endian byte order, however, V9 supports also little endian byte order, but for data access only, not instruction access.

There are two proposals, one from Sun and one from Hal, which disagree on how to handle some aspects of this calling convention.

dyncall support

dyncall fully supports the SPARC 64-bit instruction set (V9), for calls and callbacks.

D.10.1 SPARC (64-bit) Calling Convention

- 32 double precision floating point registers (d0,d2,d4,...,d62, usable as 16 quad precision ones q0,q4,q8,...q60, and also first half of them are usable as 32 single precision registers f0-f31)
- 32 64-bit integer/pointer registers out of a bigger (vendor/model dependent) number that are accessible at a time (8 are global ones (g*), whereas the remaining 24 form a register window with 8 input (i*), 8 output (o*) and 8 local (l*) ones)
- calling a function shifts the register window, the old output registers become the new input registers (old local and input ones are not accessible anymore)
- stack and frame pointer are offset by a BIAS of 2047 (see official doc for reasons)

Name	Alias	Brief description
%g0	%r0	Read-only, hardwired to 0
%g1-%g7	%r1-%r7	Global
%o0-%o3 and %i0-%i3	%r8-%r11 and %r24-%r27	Output and input argument registers, return value
%o4,%o5 and %i4,%i5	%r12,%r13 and %r28,%r29	Output and input argument registers
%o6 and %i6	%r14 and %r30, %sp and %fp	Stack and frame pointer (NOTE, offset with a BIAS of 2
%o7 and %i7	%r15 and %r31	Return address (caller writes to o7, callee uses i7)
%l0-%l7	%r16-%r23	preserve
%d0,%d2,%d4,%d6		scratch, Floating point arguments, return value
%d8,%d10,...,%d14		scratch, Floating point arguments
%d16,%d18,...,%d30		scratch (preserve for Hal), Floating point arguments
%d32,%d34,...,%d62		scratch (preserve for Hal)

Table 34: Register usage on sparc64 calling convention

Parameter passing

- stack grows down
- stack parameter order: right-to-left
- caller cleans up the stack
- stack frame is always aligned to 16 bytes

- first 6 integers are passed in registers using %o0-%o5
- first 8 quad precision floating point args (or 16 double precision, or 32 single precision) are passed in floating point registers (%q0,%q4,...,%q28 or %d0,%d2,...,%d30 or %f0-%f31, respectively)
- for every other argument the stack is used
- single precision floating point args are passed in odd %f* registers, and are "right aligned" in their 8-byte space on the stack
- for every argument passed, corresponding %o*, %f* register or stack space is skipped (e.g. passing a double as 3rd call argument, %d4 is used and %o2 is skipped)
- all arguments <= 64 bit are passed as 64 bit values
- minimum stack size is 128 bytes, b/c stack pointer must always point at enough space to store all %i* and %l* registers, used when running out of register windows
- if needed, register spill area (both, integer and float arguments are spilled in order) is adjacent to parameters
- structs with only one field are passed as if the param would be the field itself
- structs <= 16 bytes (which have more than one field) are passed field-by-field, **however** evaluated as a sequence of 8-byte parameter slots
  - note that due to aggregate alignment rules, any floating point value is either the entire slot (for double precision) or exactly one half
  - fields are left justified in register or stack slots
  - integers in a slot are passed as such (either via %o* registers or the stack)
  - single precision floats (using half of the slot) use even numbered %f* registers when they occupy the left half, odd numbered ones otherwise (no register skipping logic applied within a slot)
  - splitting struct fields between registers and stack is allowed
- unions <= 16 bytes passed by-value are passed like integers in left-justified 8-byte slots (either via %o* registers or the stack)
- aggregates (struct, union) and types > 16 bytes are passed indirectly, as a pointer to a correctly aligned copy of the data (that copy can be avoided under certain conditions)
- *non-trivial* C++ aggregates (as defined by the language) of any size, are passed indirectly via a pointer to a copy of the aggregate

Return values

- results are expected by caller to be returned in %o0-%o3 (after reg window restore, meaning callee writes to %i0-%i3) for integers
- %d0,%d2,%d4,%d6 are used for floating point values
- for *non-trivial* C++ aggregates, the caller allocates space, passes pointer to it to the callee as a hidden first param (meaning in %o0), and callee writes return value to this space; the ptr to the aggregate is returned in the same register (after reg window restore)
- the fields of aggregates (struct, union) <= 32 bytes are returned via registers mentioned above (which are assigned following the same logic as when passing the aggregate as a first argument to a function)

- aggregates (struct, union) >32 bytes are returned in a space allocated by the caller, with a pointer to it passed as first parameter to the function called (meaning in %o0)

Stack layout

Stack directly after function prolog:

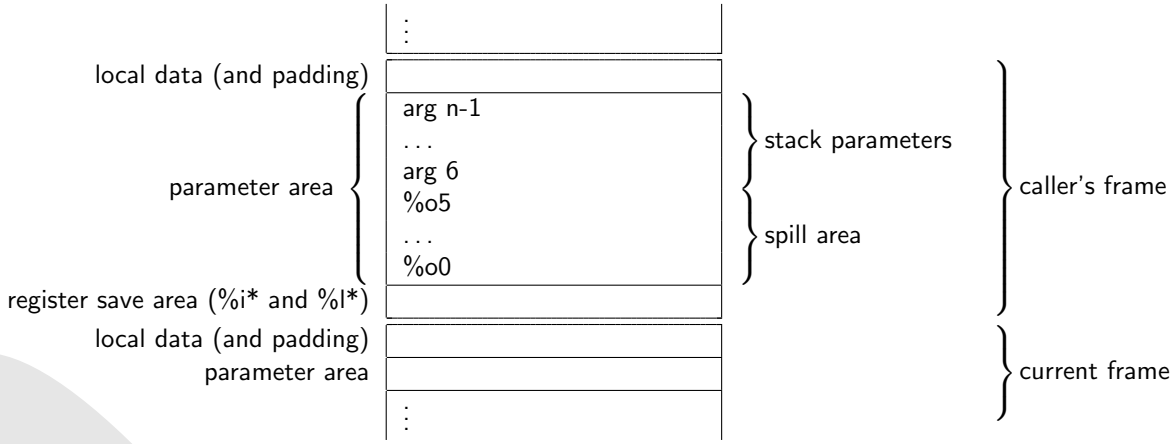


Figure 23: Stack layout on sparc64 calling convention

References

[1] Erlang/OTP
http://www.erlang.org

[2] Java Programming Language
http://www.java.com/

[3] The Programming Language Lua
http://www.lua.org/

[4] Python Programming Language
http://www.python.org/

[5] The R Project for Statistical Computing
http://www.r-project.org/

[6] Ruby Programming Language
http://www.ruby-lang.org/

[7] Go Programming Language
http://www.golang.org/

[8] cdecl calling convention / Calling conventions on the x86 platformn
http://en.wikipedia.org/wiki/X86_calling_conventions#cdecl
http://www.angelcode.com/dev/callconv/callconv.html#thiscall

[9] Windows stdcall calling convention / Microsoft calling conventions
http://msdn.microsoft.com/en-us/library/zxk0tw93(vs.71).aspx
http://www.cs.cornell.edu/courses/cs412/2001sp/resources/
microsoft-calling-conventions.html

[10] Windows fastcall calling convention / Microsoft calling conventions
http://msdn.microsoft.com/en-us/library/Aa271991
http://www.cs.cornell.edu/courses/cs412/2001sp/resources/
microsoft-calling-conventions.html

[11] GNU fastcall calling conventio / Calling conventions on the x86 platformn
http://www.ohse.de/uwe/articles/gcc-attributes.html#func-fastcall
http://www.angelcode.com/dev/callconv/callconv.html#thiscall

[12] Borland register calling convention
http://docwiki.embarcadero.com/RADStudio/en/Program_Control#Register_
Convention

[13] Watcom 32-bit register-based calling convention
http://homepage.ntlworld.com/jonathan.deboynepollard/FGA/
function-calling-conventions.html#Watcall32R http://www.openwatcom.org/index.
php/Calling_Conventions

[14] Microsoft calling conventions / Calling conventions on the x86 platform
http://www.cs.cornell.edu/courses/cs412/2001sp/resources/
microsoft-calling-conventions.html
http://www.angelcode.com/dev/callconv/callconv.html#thiscall

[15] Calling conventions on the x86 platform
http://www.angelcode.com/dev/callconv/callconv.html#thiscall

[16] Pascal calling convention
http://en.wikipedia.org/wiki/X86_calling_conventions#pascal

[17] Plan9 C compiler calling convention
http://plan9.bell-labs.com/sys/doc/compiler.pdf
http://www.mail-archive.com/9fans@9fans.net/msg16421.html

[18] ARM-THUMB Procedure Call Standard
http://infocenter.arm.com/help/topic/com.arm.doc.dui0056d/DUI0056.pdf

[19] Procedure Call Standard for the ARM Architecture
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042c/IHI0042C_aapcs.pdf

[20] Procedure Call Standard for the ARM 64-bit Architecture
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

[21] ARM64 Function Calling Conventions
https://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/Articles/ARM64FunctionCallingConventions.html

[22] Overview of ARM64 ABI conventions (Microsoft)
https://docs.microsoft.com/en-us/cpp/build/arm64-windows-abi-conventions

[23] Debian ARM EABI Port Wiki
http://wiki.debian.org/ArmEabiPort

[24] Debian ArmHardFloatPort
https://wiki.debian.org/ArmHardFloatPort

[25] MSDN: x64 Software Conventions
http://msdn.microsoft.com/en-us/library/ms235286%28VS.80%29.aspx

[26] System V Application Binary Interface - AMD64 Architecture Processor Supplement
http://www.x86-64.org/documentation/abi.pdf

[27] System V Application Binary Interface - SPARC Processor Supplement
http://sparc.org/wp-content/uploads/2014/01/psABI3rd.pdf.gz

[28] System V Application Binary Interface - SPARC Version 9 Processor Supplement
http://sparc.org/wp-content/uploads/2014/01/64.psabi_.1.35.pdf1.gz

[29] The SPARC Architecture Manual - Version 8
http://sparc.org/wp-content/uploads/2014/01/v8.pdf.gz

[30] The SPARC Architecture Manual - Version 9
http://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz

[31] SPARC Compliance Definition
https://sparc.org/wp-content/uploads/2014/01/SCD.2.4.1.pdf.gz

[32] Introduction to Mac OS X ABI Function Call Guide
https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/LowLevelABI/000-Introduction/introduction.html

[33] Linux Standard Base Core Specification for PPC32 3.2 - Chapter 8. Low Level System Information
http://refspecs.linuxbase.org/LSB_3.2.0/LSB-Core-PPC32/LSB-Core-PPC32/callingsequence.html

[34] PowerPC Embedded Application Binary Interface 32-bit Implementation
http://ftp.twaren.net/Unix/sourceware.org/binutils/ppc-docs/ppc-eabi-1995-01.pdf

[35] Developing PowerPC Embedded Application Binary Interface (EABI)
http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699700

[36] Introduction to the PowerOpen ABI
ftp://www.sourceware.org/pub/binutils/ppc-docs/ppc-poweropen/

[37] 64-bit PowerPC ELF Application Binary Interface Supplement 1.9
http://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi.html

[38] MIPS Calling Conventions Summary
http://courses.cs.washington.edu/courses/cse410/09sp/examples/MIPSCallingConventionsSummary.pdf

[39] MIPS O64 Application Binary Interface for GCC
http://gcc.gnu.org/projects/mipso64-abi.html

[40] MIPSpro™ N32 ABI Handbook
https://www.linux-mips.org/pub/linux/mips/doc/ABI/MIPS-N32-ABI-Handbook.pdf

[41] mips eabi documentation...
http://www.cygwin.com/ml/binutils/2003-06/msg00436.html

[42] NUBI - A Revised ABI for the MIPS® Architecture
ftp://ftp.linux-mips.org//pub/linux/mips/doc/NUBI/MD00438-2C-NUBIDESC-SPC-00.20.pdf

[43] devkitPro - homebrew game development
http://www.devkitpro.org/

[44] psptoolchain - all the homebrew related material ps2dev.org
http://ps2dev.org/psp/

[45] a GEM Dynamical Library system for TOS computer
http://ldg.sourceforge.net/

[46] libffi - a portable foreign function interface library
http://sources.redhat.com/libffi/

[47] C/Invoke - library for connecting to C libraries at runtime
http://www.nongnu.org/cinvoke/

[48] libffcall - foreign function call libraries
http://www.haible.de/bruno/packages-ffcall.html

[49] Universal Binary Programming Guidelines, Second Edition
http://developer.apple.com/legacy/mac/library/documentation/MacOSX/Conceptual/universal_binary/universal_binary.pdf

[50] See Mips Run, Second Edition, 2006, Dominic Sweetman