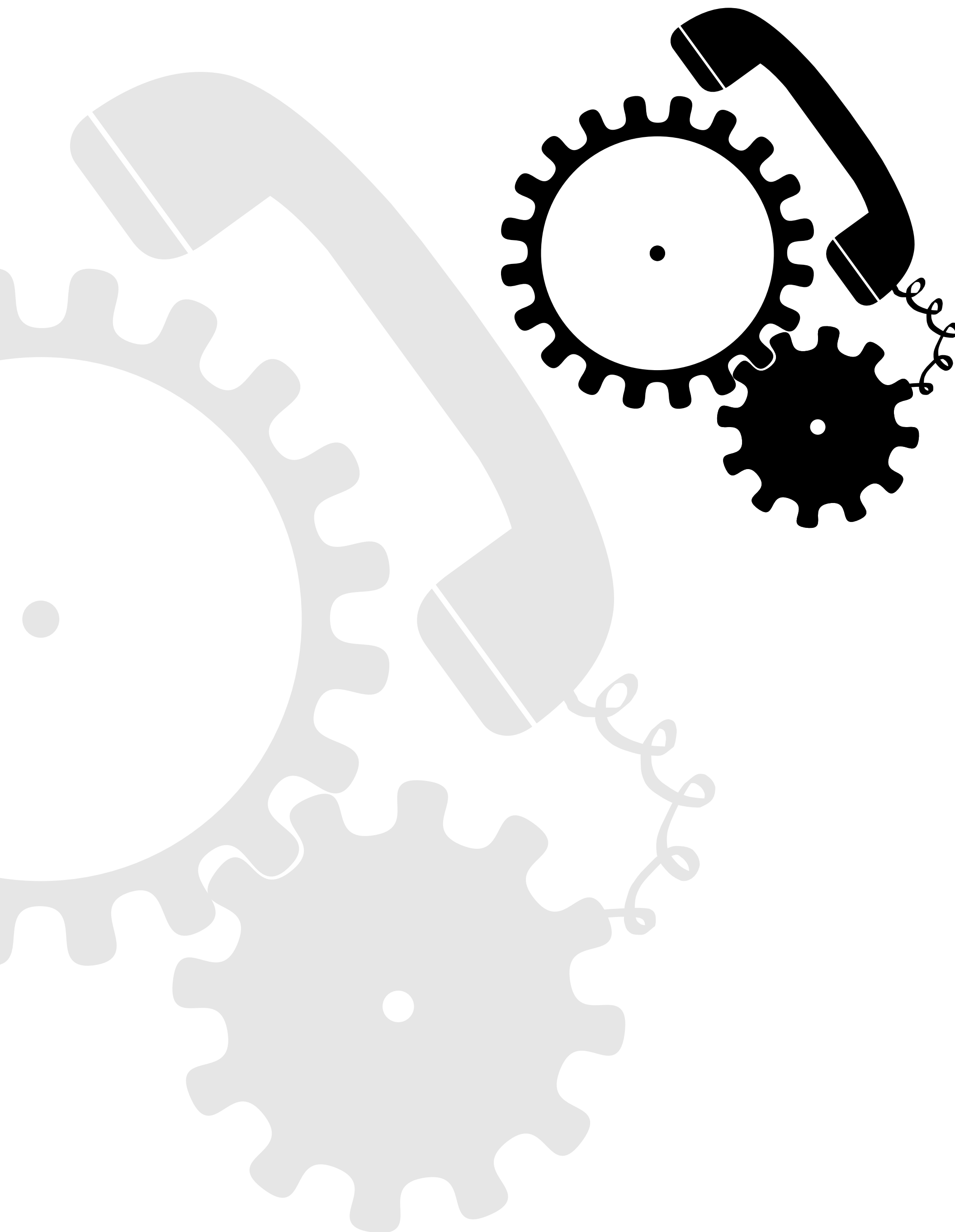
dyncall

Version 1.0

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

April 23, 2018



Contents

1 Motivation **6**
1.1 Static function calls in C . . . 6
1.2 Anatomy of machine-level calls . . . 6

2 Overview **8**
2.1 Features . . . 8
2.2 Showcase . . . 9
2.3 Supported platforms/architectures . . . 10
2.4 Build Requirements . . . 10

3 Building the library **11**
3.1 Requirements . . . 11
3.2 Supported/tested platforms and build systems . . . 11
3.3 Build instructions . . . 12
3.4 Build-tool specific notes . . . 13
3.5 Build with CMake . . . 13

4 Bindings to programming languages **14**
4.1 Common Architecture . . . 14
4.1.1 Dynamic loading of code . . . 14
4.1.2 Functions . . . 14
4.1.3 Signatures . . . 15
4.2 Erlang language bindings . . . 16
4.3 Go language bindings . . . 17
4.4 Python language bindings . . . 17
4.5 R language bindings . . . 18
4.6 Ruby language bindings . . . 19

5 Library Design **20**
5.1 Design considerations . . . 20

6 Developers **21**
6.1 Noteworthy files in the project root . . . 21
6.2 Test suites . . . 21

7 Epilog **23**
7.1 Stability and security considerations . . . 23
7.2 Embedding . . . 23
7.3 Multi-threading . . . 23
7.4 Supported types . . . 23
7.5 Roadmap . . . 23
7.6 Related libraries . . . 24

A Dyncall C library API 25
A.1 Supported C/C++ argument and return types . . . 25
A.2 Call Virtual Machine - CallVM . . . 26
A.3 Allocation . . . 26
A.4 Error Reporting . . . 26
A.5 Configuration . . . 27
A.6 Machine state reset . . . 28
A.7 Argument binding . . . 28
A.8 Call invocation . . . 28
A.9 Formatted argument binding and calls (ANSI C ellipsis interface) . . . 29

B Dyncallback C library API 30
B.1 Callback Object . . . 30
B.2 Allocation . . . 30
B.3 Callback handler . . . 30

C Dynload C library API 32
C.1 Loading code . . . 32
C.2 Retrieving functions . . . 32
C.3 Misc functions . . . 32
C.4 Symbol iteration . . . 32

D Calling Conventions 33
D.1 x86 Calling Conventions . . . 33
D.1.1 cdecl . . . 33
D.1.2 MS fastcall . . . 35
D.1.3 GNU fastcall . . . 37
D.1.4 Borland fastcall . . . 38
D.1.5 Watcom fastcall . . . 39
D.1.6 win32 stdcall . . . 40
D.1.7 MS thiscall . . . 41
D.1.8 GNU thiscall . . . 43
D.1.9 pascal . . . 44
D.1.10 plan9call . . . 45
D.2 x64 Calling Convention . . . 47
D.2.1 MS Windows . . . 47
D.2.2 System V (Linux / *BSD / MacOS X) . . . 49
D.3 PowerPC (32bit) Calling Convention . . . 51
D.3.1 Mac OS X/Darwin . . . 51
D.3.2 System V PPC 32-bit . . . 53
D.4 PowerPC (64bit) Calling Convention . . . 56
D.4.1 PPC64 ELF ABI . . . 56
D.5 ARM32 Calling Convention . . . 57
D.5.1 ATPCS ARM mode . . . 57
D.5.2 ATPCS THUMB mode . . . 59
D.5.3 EABI (ARM and THUMB mode) . . . 61
D.5.4 ARM on Apple's iOS (Darwin) Platform . . . 62
D.5.5 ARM hard float (armhf) . . . 62
D.5.6 Architectures . . . 65
D.6 ARM64 Calling Convention . . . 66
D.6.1 AAPCS64 Calling Convention . . . 66
D.6.2 Apple's ARM64 Function Calling Conventions . . . 68
D.7 MIPS32 Calling Convention . . . 69
D.7.1 MIPS EABI 32-bit Calling Convention . . . 69

D.7.2 MIPS O32 32-bit Calling Convention 71
D.7.3 MIPS N32 32-bit Calling Convention 73
D.8 MIPS64 Calling Convention 74
D.8.1 MIPS N64 Calling Convention 74
D.9 SPARC Calling Convention 76
D.9.1 SPARC (32-bit) Calling Convention 76
D.10 SPARC64 Calling Convention 78
D.10.1 SPARC (64-bit) Calling Convention 78

List of Tables

1 Supported platforms . . . 10
2 Type signature encoding for function call data types . . . 15
3 Type signature examples of C function prototypes . . . 16
4 Type signature encoding for Erlang bindings . . . 16
5 Type signature encoding for Go bindings . . . 17
6 Type signature encoding for Python bindings . . . 17
7 Type signature encoding for R bindings . . . 18
8 Type signature encoding for Ruby bindings . . . 19
9 C interface conventions . . . 25
10 Supported C/C++ argument and return types . . . 25
11 CallVM calling convention modes . . . 26
12 CallVM calling convention modes . . . 27
13 Register usage on x86 cdecl calling convention . . . 33
14 Register usage on x86 fastcall (MS) calling convention . . . 35
15 Register usage on x86 fastcall (GNU) calling convention . . . 37
16 Register usage on x86 fastcall (Borland) calling convention . . . 38
17 Register usage on x86 fastcall (Watcom) calling convention . . . 39
18 Register usage on x86 stdcall calling convention . . . 40
19 Register usage on x86 thiscall (MS) calling convention . . . 42
20 Register usage on x86 thiscall (GNU) calling convention . . . 43
21 Register usage on x86 pascal calling convention . . . 44
22 Register usage on x86 plan9call calling convention . . . 45
23 Register usage on x64 MS Windows platform . . . 47
24 Register usage on x64 System V (Linux/*BSD) . . . 49
25 Register usage on Darwin PowerPC 32-Bit . . . 51
26 Register usage on System V ABI PowerPC Processor . . . 54
27 Register usage on arm32 . . . 57
28 Register usage on arm32 thumb mode . . . 59
29 Register usage on ARM Apple iOS . . . 62
30 Register usage on armhf . . . 63
31 Overview of ARM Architecture, Platforms and Details . . . 65
32 Register usage on arm64 . . . 66
33 Register usage on MIPS32 EABI calling convention . . . 69
34 Register usage on MIPS O32 calling convention . . . 71
35 Register usage on MIPS N64 calling convention . . . 74
36 Register usage on sparc calling convention . . . 76
37 Register usage on sparc64 calling convention . . . 78

List of Figures

1 Stack layout on x86 cdecl calling convention . . . 34
2 Stack layout on x86 fastcall (MS) calling convention . . . 36
3 Stack layout on x86 fastcall (GNU) calling convention . . . 38
4 Stack layout on x86 fastcall (Borland) calling convention . . . 39
5 Stack layout on x86 fastcall (Watcom) calling convention . . . 40
6 Stack layout on x86 stdcall calling convention . . . 41
7 Stack layout on x86 thiscall (MS) calling convention . . . 42
8 Stack layout on x86 thiscall (GNU) calling convention . . . 43
9 Stack layout on x86 pascal calling convention . . . 45
10 Stack layout on x86 plan9call calling convention . . . 46
11 Stack layout on x64 Microsoft platform . . . 49

12 Stack layout on x64 System V (Linux/*BSD) 50
13 Stack layout on ppc32 Darwin 53
14 Stack layout on System V ABI for PowerPC 32-bit calling convention 55
15 Stack layout on arm32 59
16 Stack layout on arm32 thumb mode 61
17 Stack layout on arm32 armhf 64
18 Stack layout on arm64 67
19 Stack layout on mips32 eabi calling convention 70
20 Stack layout on MIPS O32 calling convention 72
21 Stack layout on mips64 n64 calling convention 75
22 Stack layout on sparc32 calling convention 77
23 Stack layout on sparc64 calling convention 79

Listings

1 C function call 7
2 Assembly X86 32-bit function call 7
3 Foreign function call in C 9
4 Dyncall C library example 9
5 Dyncall Python bindings example 9
6 Dyncall R bindings example 9

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

As you can see, this is more code after all, but completely dynamic. And definitely less than generated glue-code for each function call, if used correctly.

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, 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, as in dyncall and dyncallback. Dark green means basic support but lacking features (e.g. dyncall support, but not dyncallback). 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). For detailed info about a platform's support consult the calling convention appendix.

	Alpha	ARM	ARM64	MIPS		MIPS64		SuperH	PowerPC	PowerPC64	m68k	m88k	x86	x64	Itanium	SPARC	SPARC64	RISC-V
				EB	EL	EB	EL											
Windows family																		
Linux																		
macOS / iOS / Darwin																		
FreeBSD																		
NetBSD																		
OpenBSD																		
DragonFlyBSD																		
Solaris / SunOS																		
Plan 9 / 9front																		
Haiku / BeOS																		
Minix																		
Playstation Portable																		
Nintendo DS																		

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[40] tools on Windows
Playstation Portable	GNU make	psptoolchain[41] 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
/prefix *path*	set installation prefix (GNU make only)
/prefix-bd *path*	set build directory prefix (GNU make only)
/target-x86	build for x86 architecture (default)
/target-x64	build for x64 architecture
/target-psp	build for PlayStation Portable (homebrew SDK)
/target-nds-arm	build for Nintendo DS (devkitPro, ARM mode)
/target-nds-thumb	build for Nintendo DS (devkitPro, THUMB mode)
/tool-msvc	use Microsoft Visual C++ compiler (default)
/tool-gcc	use GNU Compiler Collection
/asm-ml	use Microsoft Macro Assembler (default)
/asm-as	use the GNU Assembler
/asm-nasm	use NASM Assembler
/config-release	build release version (default)
/config-debug	build debug version

   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 '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 conveniently 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
'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)
'v'	void

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.

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"

Table 3: Type signature examples of C function prototypes

4.2 Erlang language bindings

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

Signature character	accepted Erlang data types
'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*
'v'	no return type

Table 4: 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
'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
'v'	no return type

Table 5: 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`.

Signature character	accepted Python data types
'B'	bool
'c'	if string, take first item
's'	int, check in range
'i'	int
'j'	int
'l'	long, casted to long long
'f'	float
'd'	double
'p'	string or long casted to void*
'v'	no return type

Table 6: Type signature encoding for Python bindings

4.5 R language bindings

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

Signature character	accepted R data types
'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
'v'	no return type

Table 7: 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
'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*
'v'	no return type

Table 8: 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, unary function calls for all supported return types and calling conventions.

plain_c++ Similar to plain, but for C++ thiscalls (GNU and MS calling convention).

suite All combinations of parameter types and counts are tested on void function calls. A script written in Python (`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.

suite_x86win32std All combinations of parameter types and counts are tested on `__stdcall` void function calls. A script written in Python (`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 script written in Python (`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 function calls. A script written in Python (`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). A script written in Lua generates the tests.

callf Tests the *formatted call dyncall* C API.

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

thunk Tests *callbacks* 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 script written in Lua generates 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. Due to the still rare and often incomplete support of the `long double` type on various platforms, the latter is currently not officially supported.

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 [42] 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 [43], C/Invoke [44] and libffcall [45].

A Dyncall C library API

The library provides low-level functionality to make foreign function calls from different run-time environments. The flexibility is constrained by the set of supported types.

C interface style conventions

This manual and the *dyncall* library's C interface "dyncall.h" use the following C source code style.

Subject	C symbol	Details	Example
Types	DC<*type name*>	lower-case	DCint, DCfloat, DClong, ...
Structures	DC<*structure name*>	camel-case	DCCallVM
Functions	dc<*function name*>	camel-case	dcNewCallVM, dcArgInt, ...

Table 9: C interface conventions

A.1 Supported C/C++ argument and return types

Type alias	C/C++ data type
DCbool	_Bool, bool
DCchar	char
DCshort	short
DCint	int
DClong	long
DClonglong	long long
DCfloat	float
DCdouble	double
DCpointer	void*
DCvoid	void

Table 10: Supported C/C++ argument and return types

A.2 Call Virtual Machine - CallVM

This *CallVM* is the main entry to the functionality of the library.

Types

```
typedef void DCCallVM; /* abstract handle */
```

Details

The *CallVM* is a state machine that manages all aspects of a function call from configuration, argument passing up the actual function call on the processor.

A.3 Allocation

Functions

```
DCCallVM* dcNewCallVM (DCsize size);
void      dcFree(DCCallVM* vm);
```

`dcNewCallVM` creates a new *CallVM* object, where `size` specifies the max size of the internal stack that will be allocated and used to bind arguments to. Use `dcFree` to destroy the *CallVM* object.

This will allocate memory using the system allocators or custom ones provided custom `dcAllocMem` and `dcFreeMem` macros are defined to override the default behaviour. See `dyncall_alloc.h` for defails.

A.4 Error Reporting

Function

```
DCint dcGetError(DCCallVM* vm);
```

Returns the most recent error state code out of the following:

Errors

Constant	Description
DC_ERROR_NONE	No error occured.
DC_ERROR_UNSUPPORTED_MODE	Unsupported mode, caused by `dcMode()`

Table 11: CallVM calling convention modes

A.5 Configuration

Function

```
void dcMode (DCCallVM* vm, DCint mode);
```

Sets the calling convention to use. Note that some mode/platform combination don't make any sense (e.g. using a PowerPC calling convention on a MIPS platform) and are silently ignored.

Modes

Constant	Description
DC_CALL_C_DEFAULT	C default function call for current platform
DC_CALL_C_ELLIPSIS	C ellipsis function call (named arguments (before '...'))
DC_CALL_C_ELLIPSIS_VARARGS	C ellipsis function call (variable/unnamed arguments (after '...'))
DC_CALL_C_X86_CDECL	C x86 platforms standard call
DC_CALL_C_X86_WIN32_STD	C x86 Windows standard call
DC_CALL_C_X86_WIN32_FAST_MS	C x86 Windows Microsoft fast call
DC_CALL_C_X86_WIN32_FAST_GNU	C x86 Windows GCC fast call
DC_CALL_C_X86_WIN32_THIS_MS	C x86 Windows Microsoft this call
DC_CALL_C_X86_WIN32_THIS_GNU	C x86 Windows GCC this call
DC_CALL_C_X86_PLAN9	C x86 Plan9 call
DC_CALL_C_X64_WIN64	C x64 Windows standard call
DC_CALL_C_X64_SYSV	C x64 System V standard call
DC_CALL_C_PPC32_DARWIN	C ppc32 Mac OS X standard call
DC_CALL_C_PPC32_OSX	alias for DC_CALL_C_PPC32_DARWIN
DC_CALL_C_PPC32_SYSV	C ppc32 SystemV standard call
DC_CALL_C_PPC32_LINUX	alias for DC_CALL_C_PPC32_SYSV
DC_CALL_C_PPC64	C ppc64 SystemV standard call
DC_CALL_C_PPC64_LINUX	alias for DC_CALL_C_PPC64
DC_CALL_C_ARM_ARM	C arm call (arm mode)
DC_CALL_C_ARM_THUMB	C arm call (thumb mode)
DC_CALL_C_ARM_ARM_EABI	C arm eabi call (arm mode)
DC_CALL_C_ARM_THUMB_EABI	C arm eabi call (thumb mode)
DC_CALL_C_ARM_ARMHF	C arm call (arm hardfloat - e.g. raspberry pi)
DC_CALL_C_ARM64	C arm64 call (AArch64)
DC_CALL_C_MIPS32_EABI	C mips32 eabi call
DC_CALL_C_MIPS32_PSPSDK	alias for DC_CALL_C_MIPS32_EABI (deprecated)
DC_CALL_C_MIPS32_O32	C mips32 o32 call
DC_CALL_C_MIPS64_N64	C mips64 n64 call
DC_CALL_C_MIPS64_N32	C mips64 n32 call
DC_CALL_C_SPARC32	C sparc32 call
DC_CALL_C_SPARC64	C sparc64 call
DC_CALL_SYS_DEFAULT	C default syscall for current platform
DC_CALL_SYS_X86_INT80H_BSD	C syscall for x86 BSD platforms
DC_CALL_SYS_X86_INT80H_LINUX	C syscall for x86 Linux
DC_CALL_SYS_PPC32	C syscall for ppc32
DC_CALL_SYS_PPC64	C syscall for ppc64

Table 12: CallVM calling convention modes

Details

DC_CALL_C_DEFAULT is the default standard C call on the target platform. It uses the standard C calling convention. DC_CALL_C_ELLIPSIS is used for C ellipsis calls which allow to build up a variable argument list. On many platforms, there is only one C calling convention. The X86 platform provides a rich family of different calling conventions.

A.6 Machine state reset

```
void dcReset(DCCallVM* vm);
```

Resets the internal stack of arguments and prepares it for a new call. This function should be called after setting the call mode (using dcMode), but prior to binding arguments to the CallVM. Use it also when reusing a CallVM, as arguments don't get flushed automatically after a function call invocation. Note: you should also call this function after initial creation of the a CallVM object, as dcNewCallVM doesn't do this, implicitly.

A.7 Argument binding

Functions

```
void dcArgBool     (DCCallVM* vm, DCbool     arg);
void dcArgChar     (DCCallVM* vm, DCchar     arg);
void dcArgShort    (DCCallVM* vm, DCshort    arg);
void dcArgInt      (DCCallVM* vm, DCint      arg);
void dcArgLong     (DCCallVM* vm, DClong     arg);
void dcArgLongLong(DCCallVM* vm, DClonglong arg);
void dcArgFloat    (DCCallVM* vm, DCfloat    arg);
void dcArgDouble   (DCCallVM* vm, DCdouble   arg);
void dcArgPointer  (DCCallVM* vm, DCpointer  arg);
```

Details

Used to bind arguments of the named types to the CallVM object. Arguments should be bound in *left-to-right* order regarding the C function prototype.

A.8 Call invocation

Functions

```
DCvoid      dcCallVoid     (DCCallVM* vm, DCpointer funcptr);
DCbool      dcCallBool     (DCCallVM* vm, DCpointer funcptr);
DCchar      dcCallChar     (DCCallVM* vm, DCpointer funcptr);
DCshort     dcCallShort    (DCCallVM* vm, DCpointer funcptr);
DCint       dcCallInt      (DCCallVM* vm, DCpointer funcptr);
DClong      dcCallLong     (DCCallVM* vm, DCpointer funcptr);
DClonglong dcCallLongLong(DCCallVM* vm, DCpointer funcptr);
DCfloat     dcCallFloat    (DCCallVM* vm, DCpointer funcptr);
DCdouble    dcCallDouble   (DCCallVM* vm, DCpointer funcptr);
DCpointer   dcCallPointer  (DCCallVM* vm, DCpointer funcptr);
```

Details

Calls the function specified by *funcptr* with the arguments bound to the *CallVM* and returns. Use the function that corresponds to the dynamically called function's return value.

After the invocation of the foreign function call, the argument values are still bound and a second call using the same arguments can be issued. If you need to clear the argument bindings, you have to reset the *CallVM*.

A.9 Formatted argument binding and calls (ANSI C ellipsis interface)

Functions

```
void dcArgF  (DCCallVM* vm, const DCsigchar* signature, ...);
void dcVArgF (DCCallVM* vm, const DCsigchar* signature, va_list args);
void dcCallF (DCCallVM* vm, DCValue* result, DCpointer funcptr,
              const DCsigchar* signature, ...);
void dcVCallF(DCCallVM* vm, DCValue* result, DCpointer funcptr,
              const DCsigchar* signature, va_list args);
```

Details

These functions can be used to operate *dyncall* via a printf-style functional interface, using a signature string encoding the argument types and return type. `dcArgF()` and `dcVArgF()` just bind arguments to the `DCCallVM` object, so any return value specified in the signature is ignored. `dcCallF()` and `dcVCallF()` also take a function pointer to call after binding the arguments. The return value will be stored in what `result` points to. For more information about the signature format, refer to 2.

B Dyncallback C library API

This library extends *dyncall* with function callback support, allowing the user to dynamically create a callback object that can be called directly, or passed to functions expecting a function-pointer as argument.

Invoking a *dyncallback* calls into a user-defined unified handler that permits iteration and thus dynamic handling over the called-back-function's parameters.

The flexibility is constrained by the set of supported types, though.

For style conventions and supported types, see *dyncall* API section. In order to use *dyncallback*, include "dyncall_callback.h".

B.1 Callback Object

The *Callback Object* is the core component to this library.

Types

```
typedef struct DCCallback DCCallback;
```

Details

The *Callback Object* is an object that mimics a fully typed function call to another function (a generic callback handler, in this case).

This means, a pointer to this object is passed to a function accepting a pointer to a callback function *as the very callback function pointer itself*. Or, if called directly, cast a pointer to this object to a function pointer and issue a call.

B.2 Allocation

Functions

```
DCCallback* dcbNewCallback(const char*        signature,
                           DCCallbackHandler* funcptr,
                           void*              userdata);
void dcbFreeCallback(DCCallback* pcb);
```

dcbNewCallback creates and initializes a new *Callback* object, where signature is the needed function signature (format is the one outlined in the language bindings-section of this manual, see Table 2) of the function to mimic, funcptr is a pointer to a callback handler, and userdata a pointer to custom data that might be useful in the handler. Use dcbFreeCallback to destroy the *Callback* object.

As with dcNewCallVM/dcFree, this will allocate memory using the system allocators or custom overrides.

B.3 Callback handler

The unified callback handler's declaration used when creating a DCCallback is:

```
char cbHandler(DCCallback* cb,
               DCArgs*     args,
               DCValue*    result,
               void*       userdata);
```

cb is a pointer to the DCCallback object in use, args allows for dynamic iteration over the called-back-function's arguments (input) and result is a pointer to a DCValue object in order to store the callback's return value (output, to be set by handler).
Finally, userdata is a pointer to some user defined data that can be set when creating the callback object. The handler itself returns a signature character (see Table 2) specifying the data type used for result.

C Dynload C library API

The *dynload* library encapsulates dynamic loading mechanisms and gives access to functions in foreign dynamic libraries and code modules.

C.1 Loading code

```
DLLib* dlLoadLibrary(const char* libpath);
void  dlFreeLibrary(void* libhandle);
```

`dlLoadLibrary` loads a dynamic library at `libpath` and returns a handle to it for use in `dlFreeLibrary` and `dlFindSymbol` calls. Passing a null pointer for the `libpath` argument is valid, and returns a handle to the main executable of the calling code. Also, searching libraries in library paths (e.g. by just passing the library's leaf name) should work, however, they are OS specific. Returns a null pointer on error.

`dlFreeLibrary` frees the loaded library with handle `pLib`.

C.2 Retrieving functions

```
void* dlFindSymbol(void* libhandle, const char* symbol);
```

This function returns a pointer to a symbol with name `pSymbolName` in the library with handle `pLib`, or returns a null pointer if the symbol cannot be found. The name is specified as it would appear in C source code (mangled if C++, etc.).

C.3 Misc functions

```
int dlGetLibraryPath(DLLib* pLib, char* sOut, int bufSize);
```

This function can be used to get a copy of the path to the library loaded with handle `pLib`. The parameter `sOut` is a pointer to a buffer of size `bufSize` (in bytes), to hold the output string. The return value is the size of the buffer (in bytes) needed to hold the null-terminated string, or 0 if it can't be looked up. If `bufSize` ¿= return value ¿ 1, a null-termintedated string with the path to the library should be in `sOut`. If it returns 0, the library name wasn't able to be found. Please note that this might happen in some rare cases, so make sure to always check.

C.4 Symbol iteration

```
DLSyms*      dlSymsInit(const char* libPath);
void         dlSymsCleanup(DLSyms* pSyms);
int          dlSymsCount(DLSyms* pSyms);
const char* dlSymsName(DLSyms* pSyms, int index);
const char* dlSymsNameFromValue(DLSyms* pSyms, void* value); /* symbol must be l
```

These functions can be used to iterate over symbols. Since they can be used on libraries that are not linked, they are made for symbol name lookups, not to get a symbol's address. For that refer to `dlFindSymbol`. `dlSymsInit` will return a handle (or a null pointer on error) to the shared object specified by `libPath`, to be used with the other dlSyms* functions. Note that contrary to loading and linking libraries, no (OS-specific) rules for searching libraries in library paths, etc. apply. The handle must be freed with `dlSymsCleanup`. `dlSymsCount` returns the number of symbols in the shared object, `dlSymsName` and `dlSymsNameFromValue` are used to lookup symbol names using an index or symbol's address, respectively, returning a null pointer on error. The names are returned as they would appear in C source code (mangled if C++, etc.). The address passed to `dlSymsNameFromValue` must point to a loaded symbol.

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 aggregates (structures, unions and classes), 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

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.

dyncall support

Currently cdecl, stdcall, fastcall (MS and GNU), thiscall (MS and GNU) and plan9call are supported.

D.1.1 cdecl

Registers and register usage

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

Table 13: 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

Return values

- return values of pointer or integral type (<= 32 bits) are returned via the eax register
- integers > 32 bits are returned via the eax and edx registers
- 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:

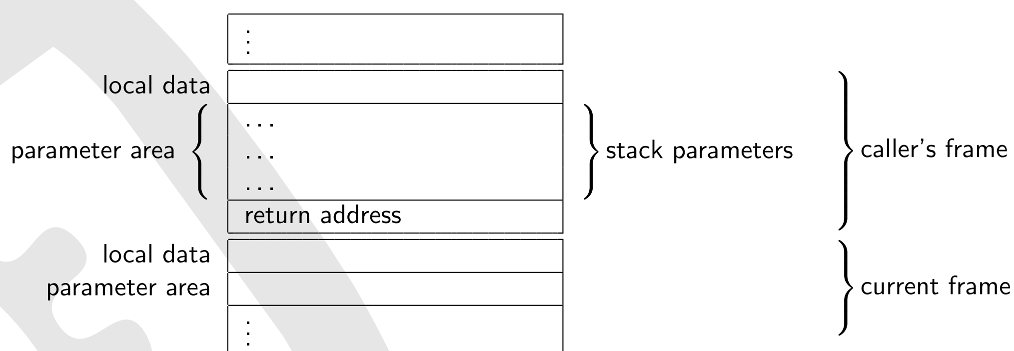


Figure 1: Stack layout on x86 cdecl calling convention

D.1.2 MS fastcall

Registers and register usage

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

Table 14: 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)
- integer types 64 bits in size @@@ ? first in edx:eax ?
- if first argument is a 64 bit integer, it is passed via ecx and edx
- 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 are returned via the eax and edx registers@@@verify
- floating point types are returned via the st0 register@@@ really ?

Stack layout

Stack directly after function prolog:

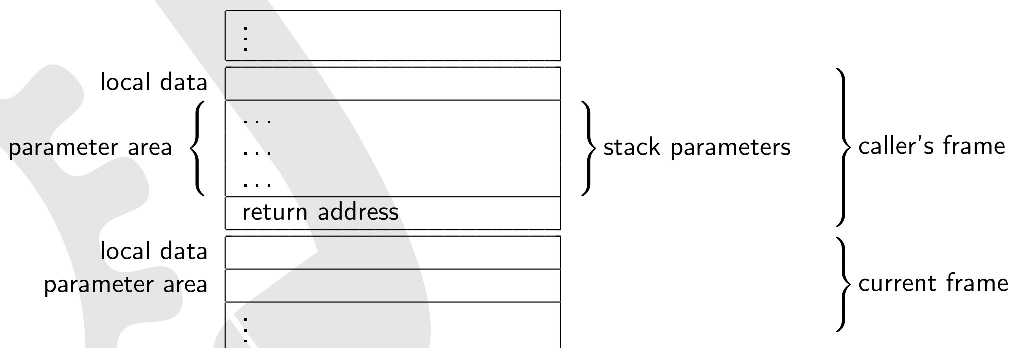


Figure 2: Stack layout on x86 fastcall (MS) calling convention

D.1.3 GNU fastcall

Registers and register usage

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

Table 15: 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)
- if first argument is a 64 bit integer, it is pushed on the stack and the two registers are skipped
- 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 are returned via the eax and edx registers.
- floating point types are returned via the st0.

Stack layout

Stack directly after function prolog:

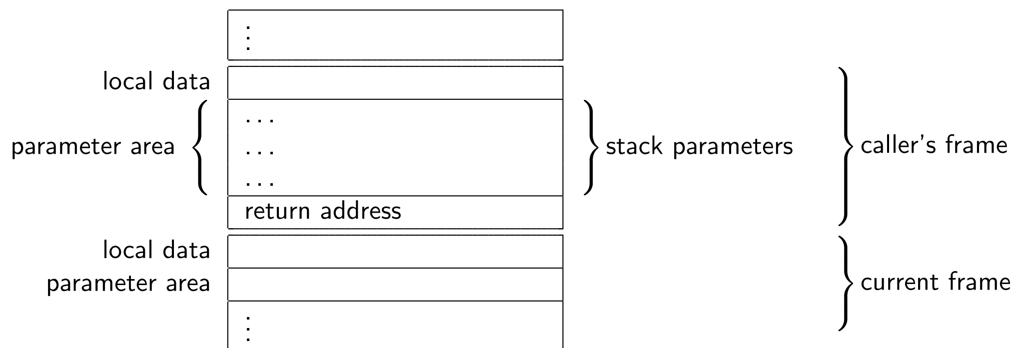


Figure 3: Stack layout on x86 fastcall (GNU) calling convention

D.1.4 Borland fastcall

Registers and register usage

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

Table 16: 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 (<= 32bit) are passed via eax, ecx and edx (even if preceded by other arguments@@@?)
- integer types 64 bits in size @@@ ?
- 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 are returned via the eax and edx registers@@@ verify
- floating point types are returned via the st0 register@@@ really ?

Stack layout

Stack directly after function prolog:

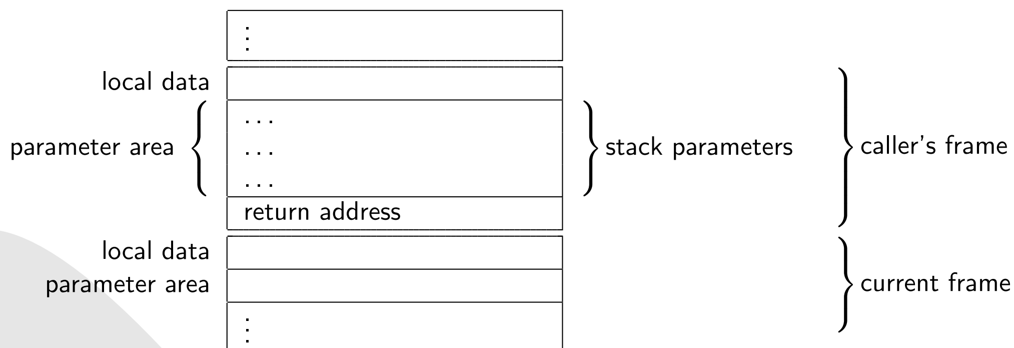


Figure 4: Stack layout on x86 fastcall (Borland) calling convention

D.1.5 Watcom fastcall

Registers and register usage

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

Table 17: 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@@@?)

- integer types 64 bits in size @@@ ?
- 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@@@verify, I thnik its esi?
- integers > 32 bits are returned via the eax and edx registers@@@ verify
- floating point types are returned via the st0 register@@@ really ?

Stack layout

Stack directly after function prolog:

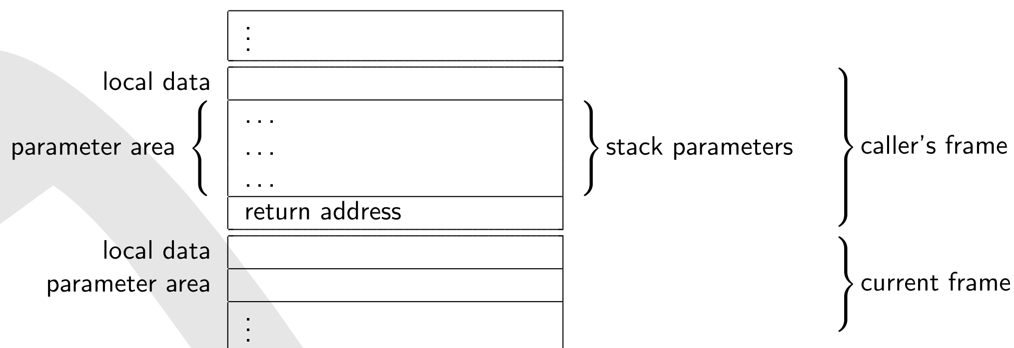


Figure 5: Stack layout on x86 fastcall (Watcom) calling convention

D.1.6 win32 stdcall

Registers and register usage

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

Table 18: 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
- Stack is usually 4 byte aligned (GCC >= 3.x seems to use a 16byte alignement@@@)
- Function name is decorated by prepending an underscore character and appending a '@' character and the number of bytes of stack space required

Return values

- return values of pointer or integral type (<= 32 bits) are returned via the eax register
- integers > 32 bits are returned via the eax and edx registers
- floating point types are returned via the st0 register

Stack layout

Stack directly after function prolog:

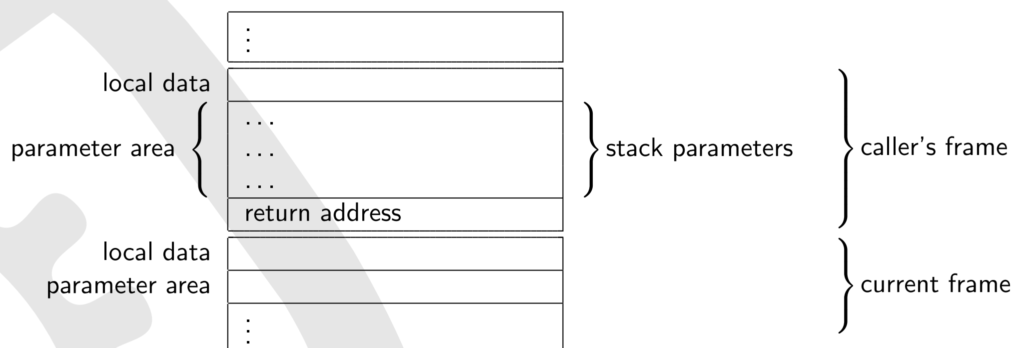


Figure 6: Stack layout on x86 stdcall calling convention

D.1.7 MS thiscall

Registers and register usage

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

Table 19: Register usage on x86 thiscall (MS) calling convention

Parameter passing

- stack parameter order: right-to-left
- called function cleans up the stack
- first parameter (this pointer) is passed via ecx
- all other parameters are pushed onto the stack
- Function name is decorated by prepending a '@' character and appending a '@' character and the number of bytes (decimal) of stack space required

Return values

- return values of pointer or integral type ($<=$ 32 bits) are returned via the eax register
- integers $>$ 32 bits are returned via the eax and edx registers@@@verify
- floating point types are returned via the st0 register@@@ really ?

Stack layout

Stack directly after function prolog:

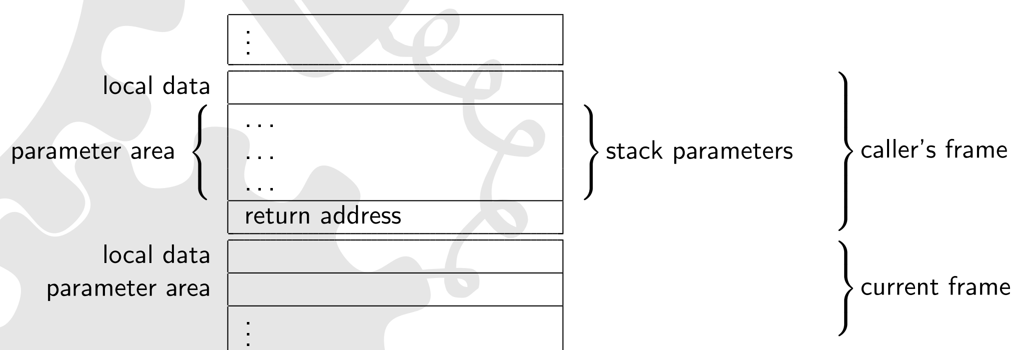


Figure 7: Stack layout on x86 thiscall (MS) calling convention

D.1.8 GNU thiscall

Registers and register usage

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

Table 20: Register usage on x86 thiscall (GNU) calling convention

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- all 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 are returned via the eax and edx registers@@@verify
- floating point types are returned via the st0 register@@@ really ?

Stack layout

Stack directly after function prolog:

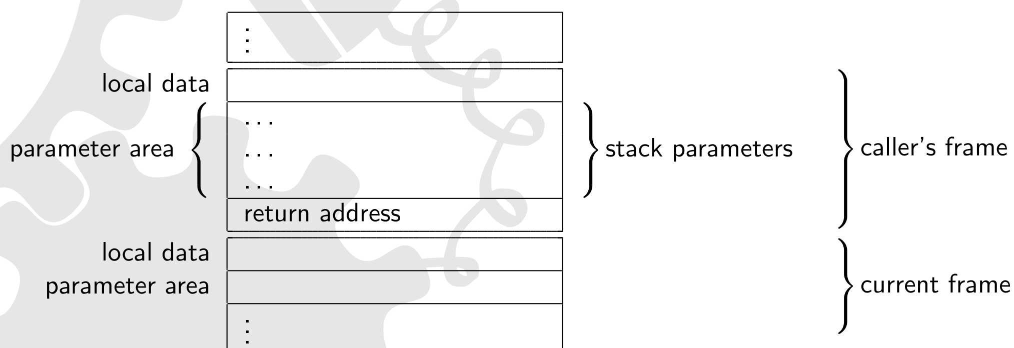


Figure 8: Stack layout on x86 thiscall (GNU) calling convention

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.

Registers and register usage

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

Table 21: Register usage on x86 pascal calling convention

Parameter passing

- stack parameter order: left-to-right
- called function cleans up the stack
- all 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 are returned via the eax and edx registers
- floating point types are returned via the st0 register

Stack layout

Stack directly after function prolog:

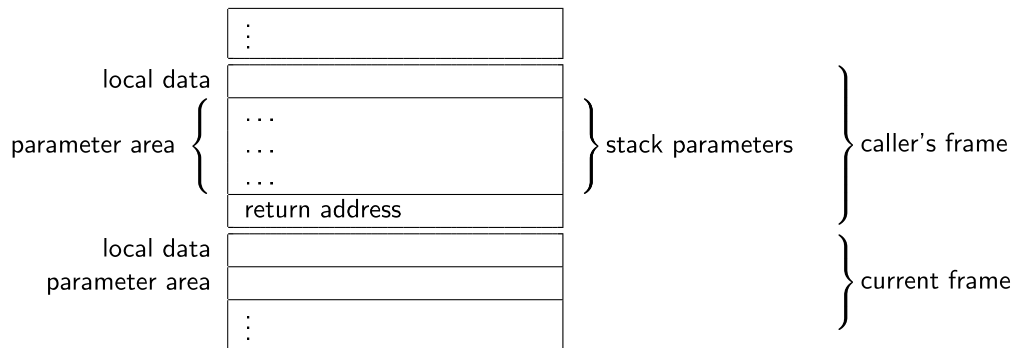


Figure 9: Stack layout on x86 pascal calling convention

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 22: 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

Return values

- return values of pointer or integral type (<= 32 bits) are returned via the eax register
- integers > 32 bits or structures 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 (called F0 in plan9 8a's terms)

Stack layout

Stack directly after function prolog:

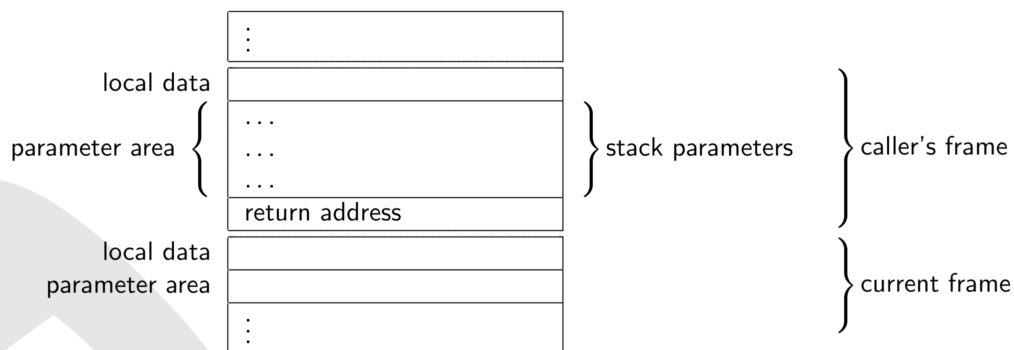


Figure 10: Stack layout on x86 plan9call calling convention

D.2 x64 Calling Convention

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 [24] differs from the SystemV x64 calling convention [25] 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

dyncall supports the MS Windows and System V calling convention.

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 23: 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 preserve area for the first 4)
- 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 bit are passed by reference
- 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 defined for 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 or integral type (<= 64 bits) are returned via the rax register
- floating point types are returned via the xmm0 register
- for types > 64 bits, a secret first parameter with an address to the return value is passed

Stack layout

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

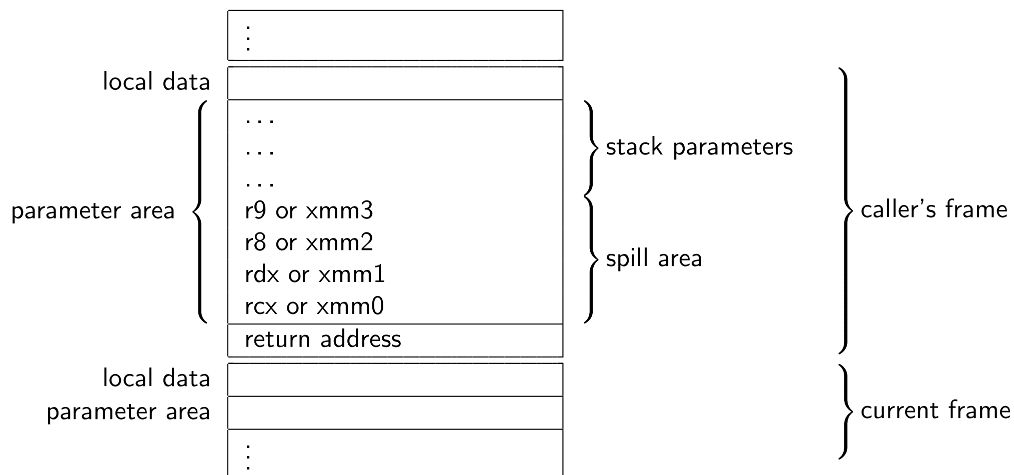


Figure 11: Stack layout on x64 Microsoft platform

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

Registers and register usage

Name	Brief description
rax	scratch, return value
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	scratch, floating point parameters 0, floating point return value
xmm1-xmm7	scratch, floating point parameters 1-7
xmm8-xmm15	scratch
st0-st1	scratch, 16 byte floating point return value
st2-st7	scratch

Table 24: 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
- 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 mustn't be exact but an upper bound on the number of used xmm registers)
- stack is always 16byte aligned - since return address is 64 bits in size, stacks with an odd number of parameters are already aligned

Return values

- return values of pointer or integral type ($<=$ 64 bits) are returned via the rax register
- floating point types are returned via the xmm0 register
- for types $>$ 64 bits, a secret first parameter with an address to the return value is passed - the passed in address will be returned in rax
- floating point values $>$ 64 bits are returned via st0 and st1

Stack layout

Stack frame is always 16-byte aligned. Note that there is no spill area. Stack directly after function prolog:

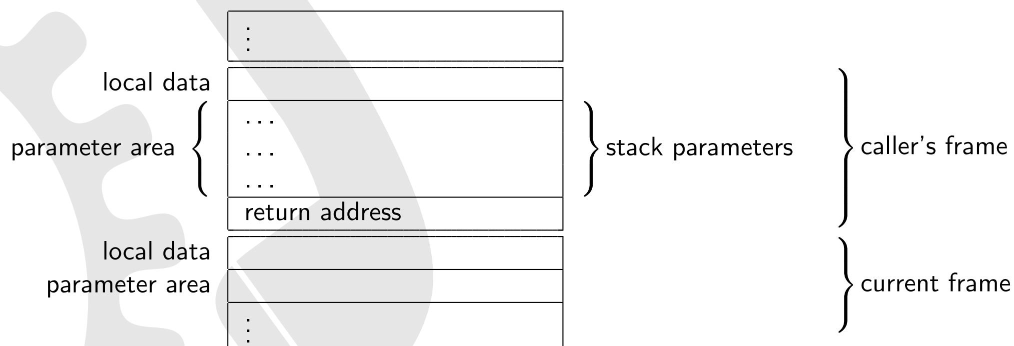


Figure 12: Stack layout on x64 System V (Linux/*BSD)

D.3 PowerPC (32bit) Calling Convention

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 Mac OS X/Darwin PPC is specified in "Mac OS X ABI Function Call Guide"[30]. It uses Big Endian (MSB).
- Linux PPC 32-bit ABI is specified in "LSB for PPC"[31] 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".

dyncall support

Dyncall and *dyncallback* are supported for PowerPC (32bit) Big Endian (MSB) on Darwin (tested on Apple Mac OS X) and Linux, however, fail for *BSD.

D.3.1 Mac OS X/Darwin

Registers and register usage

Name	Brief description
gpr0	scratch
gpr1	stack pointer
gpr2	scratch
gpr3,gpr4	return value, parameter 0 and 1 for integer or pointer
gpr5-gpr10	parameter 2-7 for integer or pointer parameters
gpr11	permanent
gpr12	branch target for dynamic code generation
gpr13-31	permanent
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	permanent
v0-v1	scratch
v2-v13	vector parameters
v14-v19	scratch
v20-v31	permanent
lr	scratch, link-register
ctr	scratch, count-register
cr0-cr1	scratch
cr2-cr4	permanent
cr5-cr7	scratch

Table 25: Register usage on Darwin PowerPC 32-Bit

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- the first 8 integer parameters are passed in registers gpr3-gpr10
- the first 12 floating point parameters are passed in registers fpr1-fpr13
- 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)
- ellipsis calls take floating point values in int and float registers (single precision floats are promoted to double precision as defined for 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 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

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
- structures $<=$ 64 bits use gpr3 and gpr4
- for types $>$ 64 bits, a secret first parameter with an address to the return value is passed

Stack layout

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

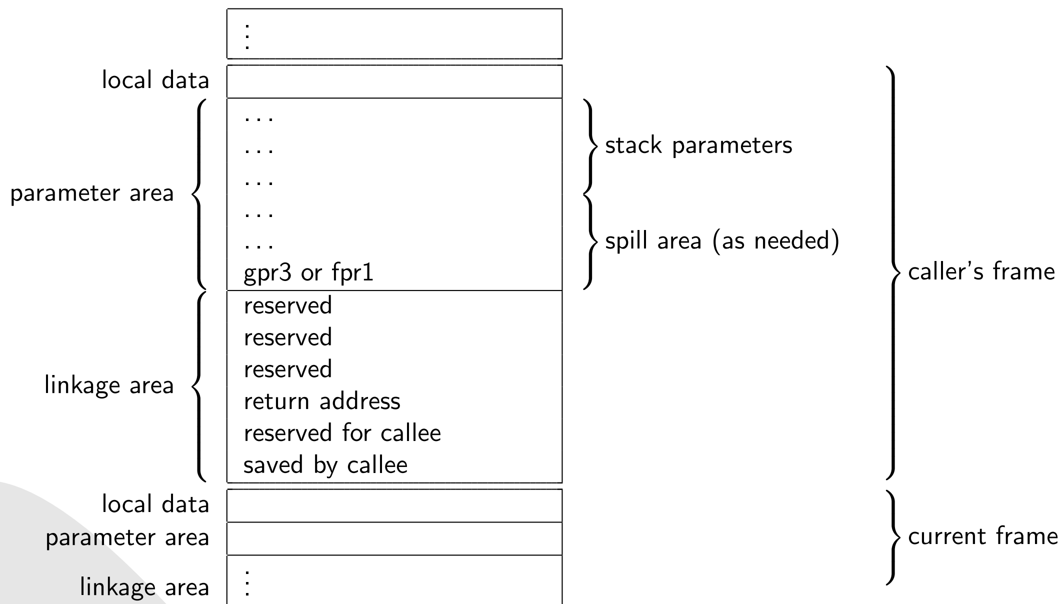


Figure 13: Stack layout on ppc32 Darwin

D.3.2 System V PPC 32-bit

Status

- C++ this calls do not work.

Registers and register usage

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), probably skipping an even integer register. or passed on the stack. They are never splitted into a register and stack part.
- Ellipse 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

Name	Brief description
r0	scratch
r1	stack pointer
r2	system-reserved
r3-r4	parameter passing and return value
r5-r10	parameter passing
r11-r12	scratch
r13	Small data area pointer register
r14-r30	Local variables
r31	Used for local variables or *environment pointer*
f0	scratch
f1	parameter passing and return value
f2-f8	parameter passing
f9-13	scratch
f14-f31	Local variables
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 26: Register usage on System V ABI PowerPC Processor

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.

Stack layout

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

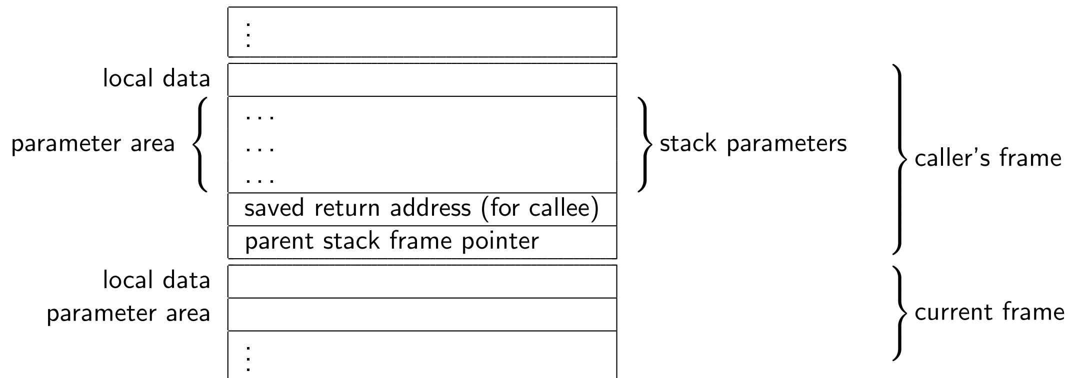


Figure 14: Stack layout on System V ABI for PowerPC 32-bit calling convention

D.4 PowerPC (64bit) Calling Convention

Overview

- Word 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"[30]. It uses Big Endian (MSB).
- Linux PPC 64-bit ABI is specified in "64-bit PowerPC ELF Application Binary Interface Supplement"[34] which is based on "System V ABI".

dyncall support

Dyncall supports PowerPC (64bit) Big Endian and Little Endian ELF ABIs on System V systems (Linux, etc.), including syscalls. Mac OS X is not supported.

D.4.1 PPC64 ELF ABI

Registers and register usage

@@@

Parameter passing

@@@

- 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

Return values

@@@

Stack layout

@@@

D.5 ARM32 Calling Convention

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 the Debian ARM EABI port wiki [22].

dyncall support

Currently, the *dyncall* library supports the ARM and THUMB mode of the ARM32 family (ATPCS [18] and EABI [22]), excluding manually triggered ARM-THUMB interworking calls. Although it's quite possible that the current implementation runs on other ARM processor families as well, please note that only the ARMv4t family has been thoroughly tested at the time of writing. Please report if the code runs on other ARM families, too.
It is important to note, that dyncall supports the ARM architecture calling convention variant **with floating point hardware disabled** (meaning that the FPA and the VFP (scalar mode) procedure call standards are not supported). This processor family features 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. that 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	Brief description
r0	parameter 0, scratch, return value
r1	parameter 1, scratch, return value
r2-r3	parameters 2 and 3, scratch
r4-r10	permanent
r11	frame pointer, permanent
r12	scratch
r13	stack pointer, permanent
r14	link register, permanent
r15	program counter (note: due to pipeline, r15 points to 2 instructions ahead)

Table 27: 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), with the loword coming first
- structures and unions are passed by value, with the first four words of the parameters in r0-r3
- if return value is a structure, a pointer pointing to the return value's space is passed in r0, the first parameter in r1, etc... (see **return values**)
- 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
- if return value is a structure, the caller allocates space for the return value on the stack in its frame and passes a pointer to it in r0

Stack layout

Stack directly after function prolog:

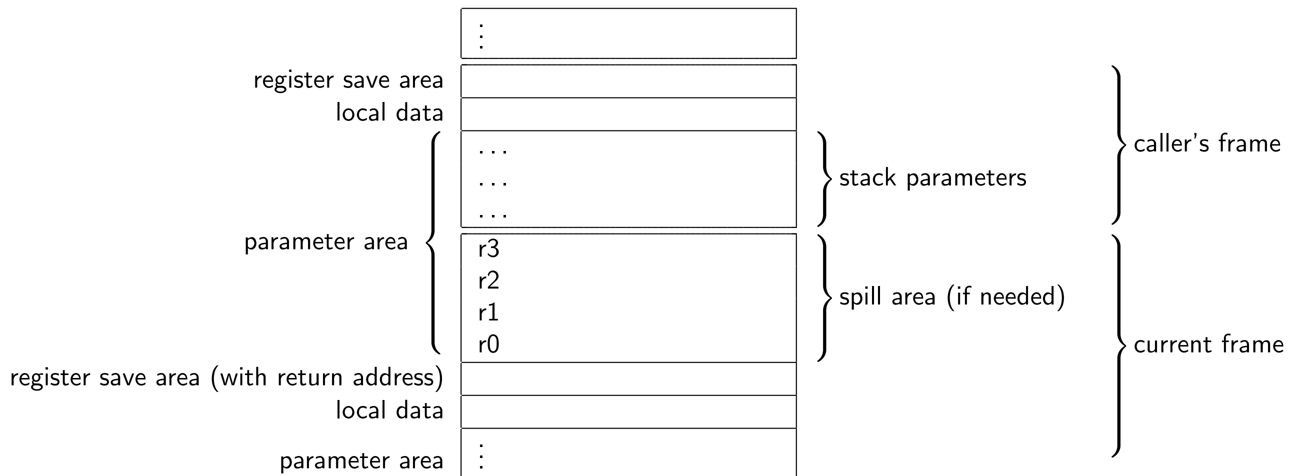


Figure 15: Stack layout on arm32

D.5.2 ATPCS THUMB mode

Status

- The ATPCS THUMB mode is untested.
- Ellipse calls may not work.
- C++ this calls do not work.

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	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-r11	permanent
r12	scratch
r13	stack pointer, permanent
r14	link register, permanent
r15	program counter (note: due to pipeline, r15 points to 2 instructions ahead)

Table 28: 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), with the loword coming first
- structures and unions are passed by value, with the first four words of the parameters in r0-r3
- if return value is a structure, a pointer pointing to the return value's space is passed in r0, the first parameter in r1, etc. (see **return values**)
- 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
- if return value is a structure, the caller allocates space for the return value on the stack in its frame and passes a pointer to it in r0

Stack layout

Stack directly after function prolog:

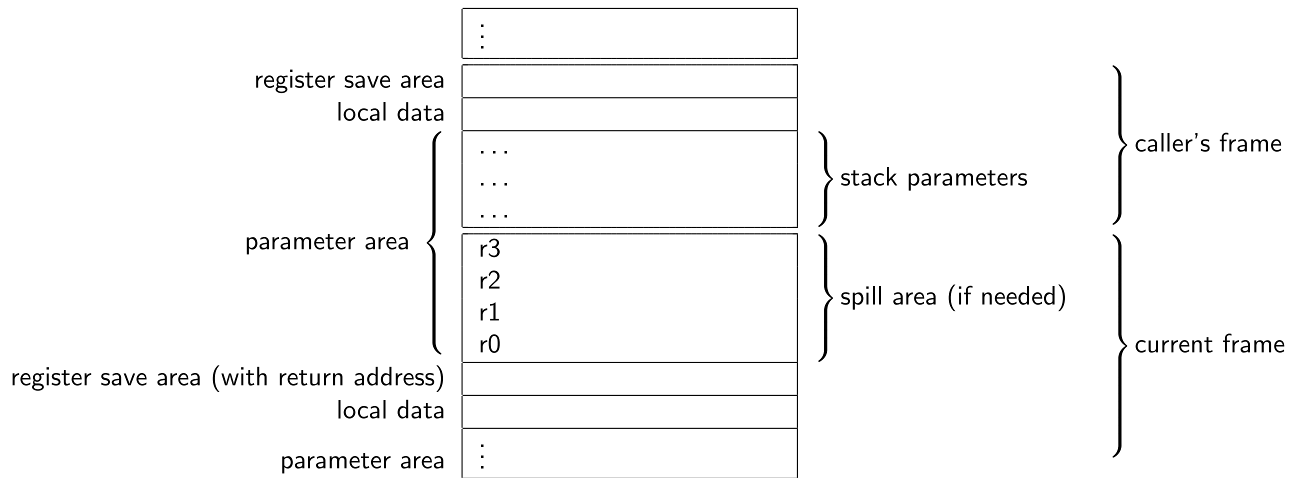


Figure 16: Stack layout on arm32 thumb mode

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 [22].

Status

- The EABI THUMB mode is tested and works fine (contrary to the ATPCS).
- Ellipse calls do not work.
- C++ this calls do not work.

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

The iOS runs on ARMv6 (iOS 2.0) and ARMv7 (iOS 3.0) architectures. Typically code is compiled in Thumb mode.

Register usage

Name	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) and scratch (since iOS 3.0)
R10-R11	permanent
R12	scratch, intra-procedure scratch register (IP) used by dynamic linker
R13	stack pointer, permanent
R14	link register, permanent
R15	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-A7. Not accesible from Thumb mode on ARMv6.
D16-D31	Only available in ARMv7, aliases Q8-Q15.
FPSCR	VFP status register.

Table 29: Register usage on ARM Apple iOS

The ABI is based on the AAPCS but with some important differences listed below:

- R7 instead of R11 is used as frame pointer
- R9 is scratch since iOS 3.0, was preserved before.

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. The instruction set used for armhf is Thumb-2. Refer to the debian wiki for more information [23].

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

Register usage

Parameter passing

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

Name	Brief description
R0	parameter 0, scratch, non floating point return value
R1	parameter 1, scratch, non floating point return value
R2,R3	parameters 2 and 3, scratch
R4,R5	permanent
R6	scratch
R7	frame pointer, permanent
R8	permanent
R9,R10	scratch
R11	permanent
R12	scratch, intra-procedure scratch register (IP) used by dynamic linker
R13	stack pointer, permanent
R14	link register, permanent
R15	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 30: Register usage on armhf

- 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
- structures and unions are passed by value, with the first four words of the parameters in r0-r3 @@@?check doc
- if return value is a structure, a pointer pointing to the return value's space is passed in r0, the first parameter in r1, etc. (see **return values**)
- 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
- single precision floating point return value uses s0
- double precision floating point return value uses d0
- if return value is a structure, the caller allocates space for the return value on the stack in its frame and passes a pointer to it in r0

Stack layout

Stack directly after function prolog:

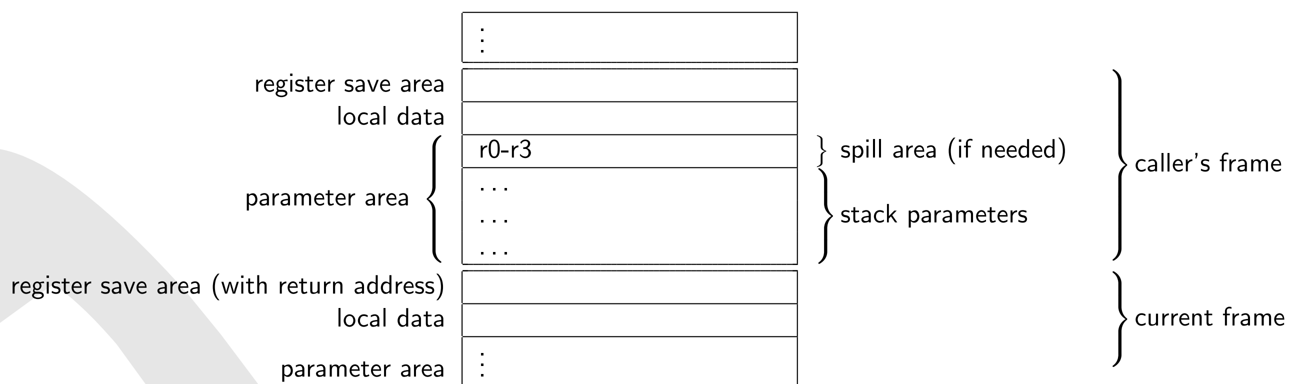


Figure 17: Stack layout on arm32 armhf

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 throughout available, armhf calling convention on so
ARMv8	iPhone 6 and higher	64bit support

Table 31: Overview of ARM Architecture, Platforms and Details

D.6 ARM64 Calling Convention

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, for calls and callbacks.

D.6.1 AAPCS64 Calling Convention

Registers and register usage

ARM64 features thirty-one 64 bit general purpose registers, namely x0-x30. Also, there is SP, a register with restricted use, used for the stack pointer, and PC dedicated as 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, 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

Table 32: 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 parameters require saving, though)

- structures and unions are passed by value, with the first four words of the parameters in r0-r3
- if return value is a structure, a pointer pointing to the return value's space is passed in r0, the first parameter in r1, etc... (see **return values**)
- stack is required to be throughout eight-byte aligned

Return values

- integer return values use x0
- floating-point return values use d0
- otherwise, the caller allocates space, passes pointer to it to the callee through x8, and callee writes return value to this space

Stack layout

Stack directly after function prolog:

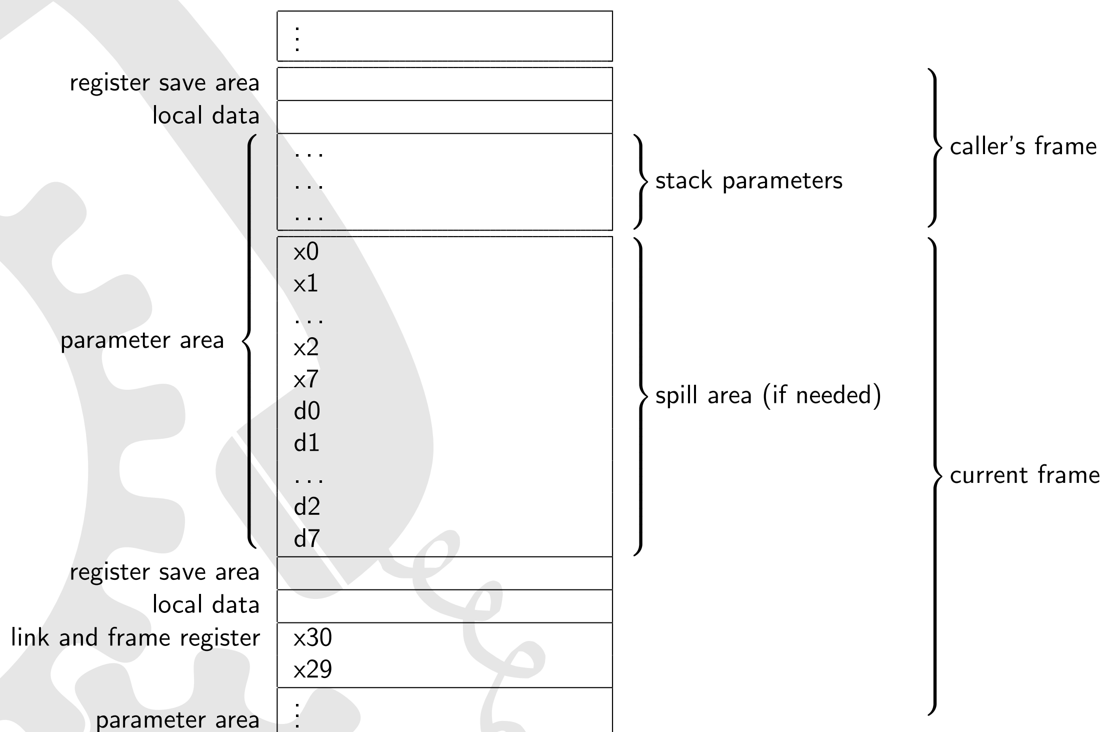


Figure 18: Stack layout on arm64

D.6.2 Apple's ARM64 Function Calling Conventions

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 the 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

D.7 MIPS32 Calling Convention

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 processor 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*[35], *O64*[36], *N32*[37], *N64*[37], *EABI*[38] and *NUBI*[39].

dyncall support

Currently, dyncall supports for MIPS 32-bit architectures the widely-used O32 calling convention (for big- and little-endian targets), as well as EABI (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
$1	**$at**	Assembler temporary
$2-$3	**$v0-$v1**	Integer results
$4-$11	**$a0-$a7**	Integer arguments, or double precision float arguments
$12-$15,$24	**$t4-$t7,$t8**	Integer temporaries
$25	**$t9**	Integer temporary, hold the address of the called function for all PIC calls
$16-$23	**$s0-$s7**	Preserved
$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		Float results
$f1,$f3,$f4-$f11,$f20-$f23		Float temporaries
$f12-$f19		Single precision float arguments

Table 33: 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
- if either integer or float registers are used up, the stack is used
- 64-bit stack arguments are always aligned to 8 bytes
- 64-bit integers or double precision floats are passed on two general purpose registers starting at an even register number, skipping one odd register
- $a0-$a7 and $f12-$f19 are not required to be preserved
- results are returned in $v0 (32-bit), $v0 and $v1 (64-bit), $f0 or $f0 and $f2 (2 × 32 bit float e.g. complex)

Stack layout

Stack directly after function prolog:

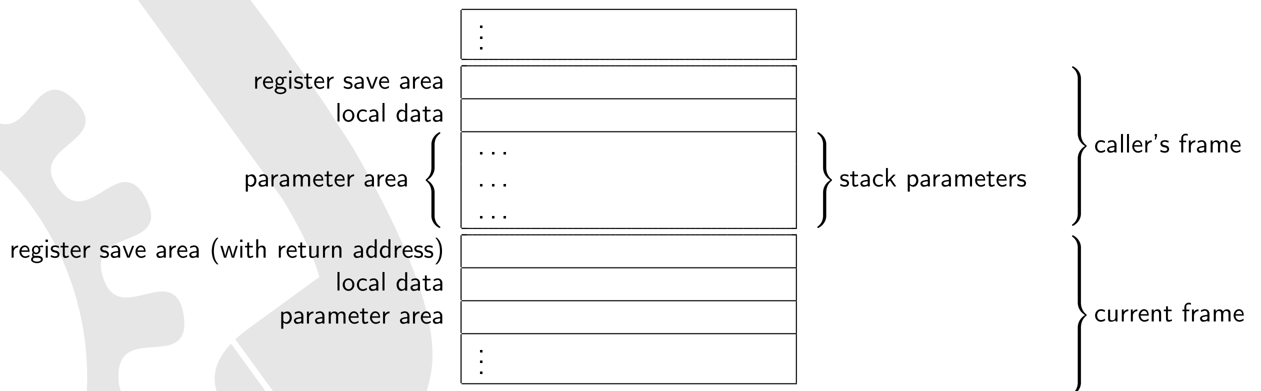


Figure 19: Stack layout on mips32 eabi calling convention

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, scratch
$4-$7	**$a0-$a3**	first integer arguments, scratch
$8-$15,$24	**$t0-$t7,$t8**	temporaries, scratch
$25	**$t9**	temporary, hold the address of the called function for all 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
$30	**$fp**	frame pointer, preserve
$31	**$ra**	return address, preserve
hi, lo		multiply/divide special registers
$f0-$f3		float return value, scratch
$f4-$f11,$f16-$f19		float temporaries, scratch
$f12-$f15		first floating point arguments, scratch
$f20-$f31		preserved

Table 34: 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 and 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 (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
- 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
- if any arguments are passed via float registers, skip $a0-$a3 for subsequent arguments as if the values were passed via them
- note that if the first argument is not a float, but the second, it'll get passed via the $a? registers

- results are returned in $v0 (32-bit int return values), $f0 (32-bit float), $v0 and $v1 (64-bit int), $f0 and $f3 (64bit float)

Stack layout

Stack directly after function prolog:

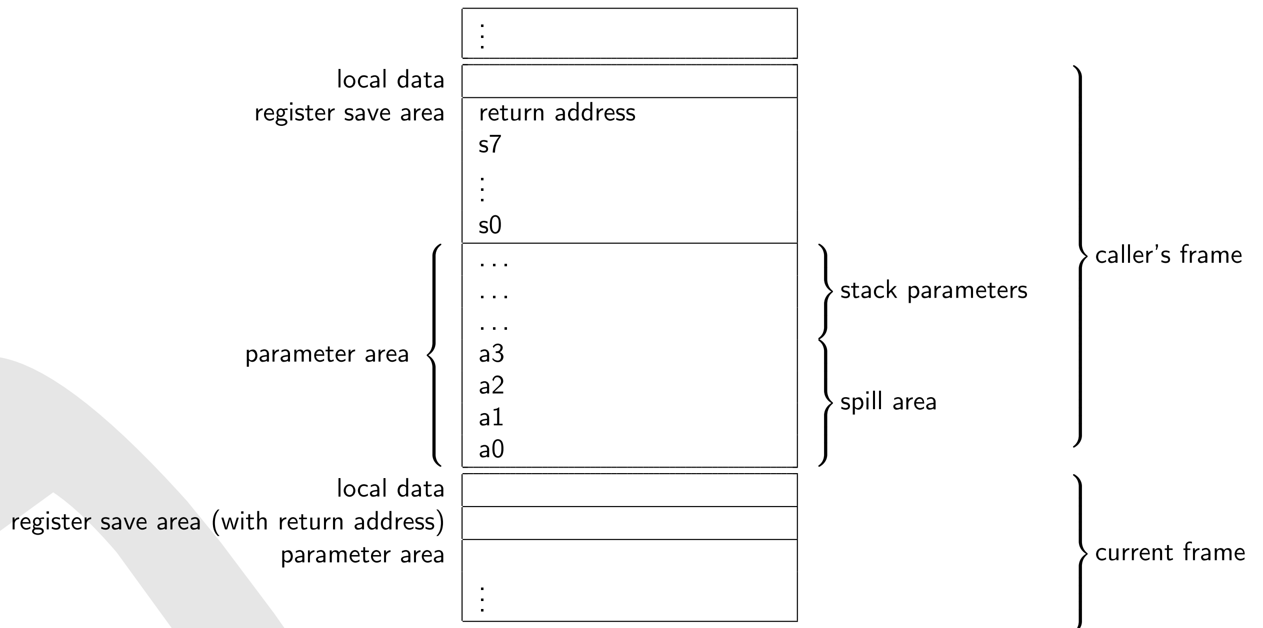


Figure 20: Stack layout on MIPS O32 calling convention

D.7.3 MIPS N32 32-bit Calling Convention

@@@

D.8 MIPS64 Calling Convention

Overview

There are two main ABIs in use for MIPS64 chips, *N64*[37] and *N32*[37]. Both are basically the same, except that N32 uses 32-bit pointers and long integers, instead of 64. 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 64-bit versions of the other ABIs found for MIPS32, e.g. the EABI[38] and O64[36].

dyncall support

For MIPS 64-bit machines, dyncall supports the N64 calling conventions for calls and callbacks (for big- and little-endian 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
$2-$3	**$v0-$v1**	Integer results
$4-$11	**$a0-$a7**	Integer arguments, or double precision float arguments
$12-$15,$24	**$t4-$t7,$t8**	Integer temporaries
$25	**$t9**	Integer temporary, hold the address of the called function for all PIC calls
$16-$23	**$s0-$s7**	Preserved
$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		Float results
$f1,$f3,$f4-$f11,$f20-$f23		Float temporaries
$f12-$f19		Float arguments
$f24-$f31		Preserved

Table 35: Register usage on MIPS N64 calling convention

Parameter passing

- Stack grows down
- Stack parameter order: right-to-left
- Caller cleans up the stack
- first 8 params >= 64-bit are passed in registers $a0-$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.)

- subsequent arguments are pushed onto the stack
- all stack entries are 64-bit aligned
- all stack regions are 16-byte aligned
- results are returned in $v0, and for a second one $v1 is used
- float arguments passed in the variable part of a vararg call are passed like integers
- 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 @@@doc says "undecided", but openbsd/octeon(mipseb) has it as described here

Stack layout

Stack directly after function prolog:
@@@ WIP, might be wrong

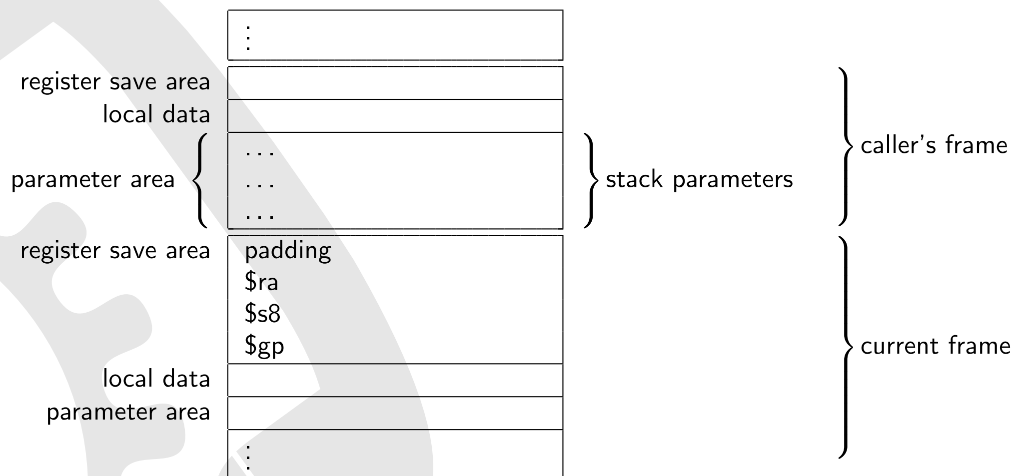


Figure 21: Stack layout on mips64 n64 calling convention

D.9 SPARC Calling Convention

Overview

The SPARC family of processors is based on the SPARC instruction set architecture, which comes in basically tree revisions, V7, V8[28][26] and V9[29][27]. 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.

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 36: 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 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
- 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
- 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 for floats, and for structs/unions a pointer to them is used as a hidden stack parameter (see below)

Stack layout

Stack directly after function prolog:

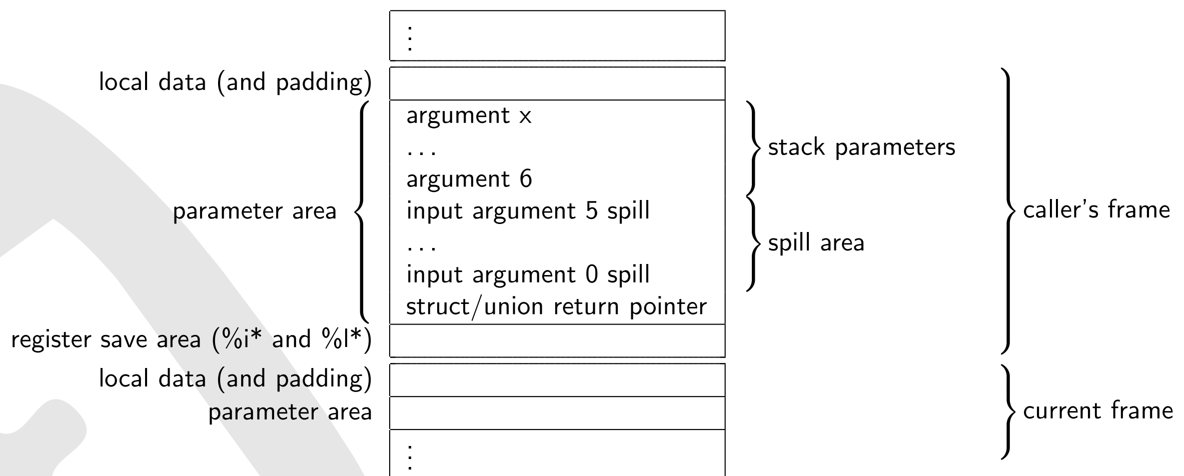


Figure 22: Stack layout on sparc32 calling convention

D.10 SPARC64 Calling Convention

Overview

The SPARC family of processors is based on the SPARC instruction set architecture, which comes in basically tree revisions, V7, V8[28][26] and V9[29][27]. 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 probosals, 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,...g60, 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, value is pointing to stac
%o7 and %i7	%r15 and %r31	Return address (caller writes to o7, callee uses i7)
%l0-%l7	%r16-%r23	preserve
%d0,%d2,%d4,%d6		Floating point arguments, return value
%d8,%d10,...,%d30		Floating point arguments
%d32,%d36,...,%d62		scratch (but, according do Hal, %d16,...,%d46 are preserv

Table 37: 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-%f32, 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 doube 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 (for integer arguments passed via %o0-%o5) is adjacent to parameters
- 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 for floats
- structs/unions up to 32b, the fields are returned via the respective registers mentioned in the previous bullet point
- for structs/unions >= 32b, the caller allocates the space and a pointer to it is passed as hidden first parameter to the function called (meaning in %o0)

Stack layout

Stack directly after function prolog:

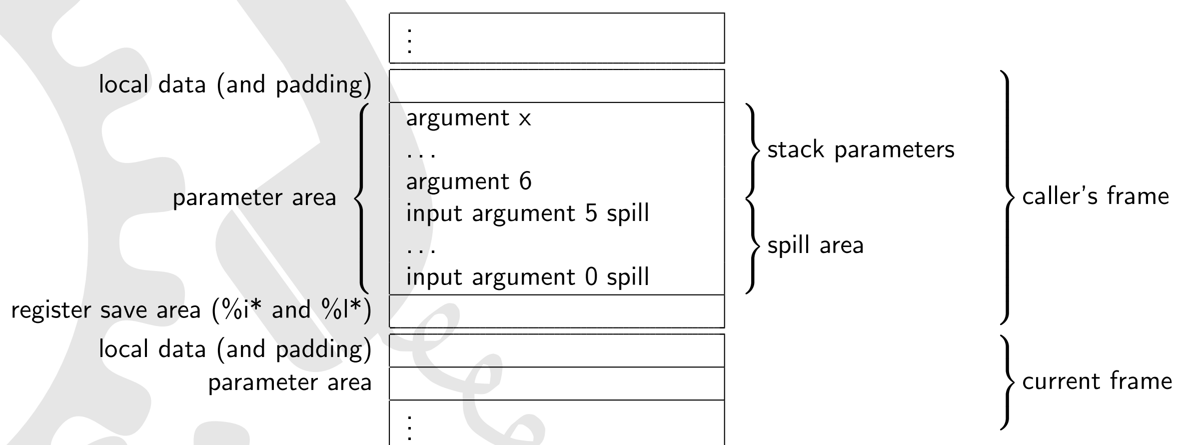


Figure 23: Stack layout on sparc64 calling convention

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

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

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

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

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

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

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

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

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

[31] 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

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

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

[34] 64-bit PowerPC ELF Application Binary Interface Supplement 1.9
http://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi.html

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

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

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

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

[39] 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

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

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

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

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

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

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

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

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