dyncall Library

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

July 6, 2009

Contents

List of Tables

List of Figures

Listings

1 Motivation

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

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

1.1 Static function calls in C

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

1.2 Anatomy of machine-level calls

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

1. The target function's calling convention dictates how the stack is prepared, arguments are passed, results are returned and how to clean up afterwards.

2. Function call arguments are loaded in registers and on the stack according to the calling convention that take alignment constraints into account.

3. Control flow transfer from caller to callee.

4. Process return value, if any. Some calling conventions specify that the caller is responsible for cleaning up the argument stack.

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

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

Listing 1: C function call

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

Listing 2: Assembly X86 32-bit function call

2 Overview

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

bind argument parameters from left to right and then call

interface allowing programmers to call C functions in a completely dynamic manner. In other words, instead of calling a function directly, the *dyncall* library provides a mechanism to push the function parameters manually and to issue the call afterwards.
Since the idea behind this concept is similar to call dispatching mechanisms of virtual machines, the object that can be dynamically loaded with arguments, and then used to actually invoke the call, is called CallVM. It is possible to change the calling convention used by the CallVM at run-time. Due to the fact that nearly every platform comes with one or more calling conventions, the *dyncall* library project intends to be an 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, Playstation Portable and Nintendo DS.
- Add-on language bindings to Python,R,Ruby.
- High-level state machine design using C to model calling convention parameter transfer.
- One assembly *hybrid* call routine per calling convention.
- Formatted call, ellipsis 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);
  dcArgDouble(vm, x);
  r = dcCallDouble(vm, funptr);
  dcFreeCallVM(vm);
  return r;
}
```

Listing 4: Dyncall C library example

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

```
library(rdc)
call.as.sqrt <- function(funptr,x)
  rdc.call(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.
Please note that a green cell doesn't imply that all existing calling conventions/features/build tools are supported for that platform (but the most important). For details about the support consult the appendix.

	ARM	MIPS	SuperH	PowerPC (32)	PowerPC (64)	x86	x64	Itanium	SPARC	SPARC64
Windows/Windows CE	🟨	🟨	🟥	⬜	⬜	🟩	🟩	🟥	⬜	⬜
Linux	🟨	🟨	🟥	🟩	🟥	🟩	🟩	🟥	🟥	🟥
Darwin	🟨	⬜	⬜	🟩	🟥	🟩	🟩	⬜	⬜	⬜
FreeBSD	⬜	⬜	🟥	🟨	🟥	🟩	🟩	🟥	🟥	🟥
NetBSD	🟨	🟨	🟥	🟩	🟥	🟩	🟩	🟥	🟥	🟥
OpenBSD	🟨	🟨	🟥	🟨	🟥	🟩	🟩	🟥	🟥	🟥
DragonFlyBSD	⬜	⬜	⬜	⬜	⬜	🟩	⬜	⬜	⬜	⬜
Solaris	⬜	⬜	⬜	⬜	⬜	🟩	🟩	⬜	🟥	🟥
Playstation Portable	⬜	🟩	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
Nintendo DS	🟩	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜

Table 1: Supported platforms

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 or Microsoft C/C++ compiler)
- C++ compiler to build the optional test cases (GCC or Microsoft C/C++ compiler)
- Python (optional - for generation of some test cases)
- BSD make, GNU make, or Microsoft nmake as automated build tools

3.2 Supported/tested platforms and build systems

Although it is possible to build the *dyncall* library on more platforms than the ones outlined here, this section doesn't list operating systems or architectures the authors didn't test. However, untested platforms using the same build tools (e.g. the BSD family of operating systems using similar flavors of the BSD make utility along with GCC, etc.) should work without modification. If you have problems building the *dyncall* library on one of the platforms mentioned below, or if you successfully built it on a yet unlisted one, please let us know.

x86	
Windows	nmake, GNU make (via MinGW)
Darwin	GNU make, BSD make
Linux	GNU make
Solaris	GNU make (Sun's make tool isn't supported)
FreeBSD	BSD make
NetBSD	BSD make
OpenBSD	BSD make
DragonFlyBSD	BSD make

x64	
Windows	nmake
Darwin	GNU make, BSD make
Linux	GNU make
Solaris	GNU make (Sun's make tool isn't supported)
FreeBSD	BSD make
NetBSD	BSD make
OpenBSD	BSD make

PowerPC (32bit)	
Darwin	GNU make, BSD make
Linux	GNU make
NetBSD	BSD make

ARM9E	
Nintendo DS	nmake (and devkitPro[12] tools)

MIPS32	
Playstation Portable	GNU make (and psptoolchain[13] tools)

3.3 Build instructions

1. Configure the source

   ***nix flavour**

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

   windows flavour

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

   Available options:

`prefix=`*path*	specify installation prefix (Unix shell)
`prefix` *path*	specify installation prefix (Windows batch)
`target-x86`	build for x86 architecture
`target-x64`	build for x64 architecture
`target-ppc32`	build for ppc 32-bit architecture (not on windows batch)
`target-psp`	cross-compile build for Playstation Portable (homebrew SDK)
`target-nds-arm`	cross-compile build for Nintendo DS (using ARM mode)
`target-nds-thumb`	cross-compile build for Nintendo DS (using THUMB mode)
`tool-gcc`	use GNU Compiler Collection tool-chain
`tool-msvc`	use Microsoft Visual C++
`asm-as`	use the GNU Assembler
`asm-nasm`	use NASM Assembler
`asm-ml`	use Microsoft Macro Assembler
`config-release`	build release version (default)
`config-debug`	build debug version

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

   ```
   make                        # when using {GNU,BSD} Make
   bsdmake                     # when using BSD Make on Darwin
   make -f BSDmakefile         # when using BSD Make on NetBSD
   nmake /f Nmakefile          # when using NMake on Windows
   ```

3. Install libraries and includes (not supported for nmake based builds)

   ```
   make install
   ```

4. Optionally, build the test suites

   ```
   make test                   # when using {GNU,BSD} Make
   bsdmake test                # when using BSD Make on Darwin
   make -f BSDmakefile test    # when using BSD Make on NetBSD
   nmake /f Nmakefile test     # when using NMake on Windows
   ```

5. Optionally, build the manual (Latex required)

   ```
   make doc                    # when using {GNU,BSD} Make
   bsdmake doc                 # when using BSD Make on Darwin
   make -f BSDmakefile doc     # when using BSD Make on NetBSD
   nmake /f Nmakefile doc      # when using NMake on Windows
   ```

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 Java[1], Lua[2], Python[3], R[4] and Ruby[5].
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 interface 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 a code module.

4.1.2 Functions

All bindings are based on a common interface convention providing a common set of the following 4 functions:

load loading a module of compiled code

free unloading a module of compiled code

find finding function pointer by symbolic names

call invoking 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. The high-level C interface functions `dcCallF()` and `dcCallFV()` also make use of the *dyncall* signature string.

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

While the size and encoding scheme (integer, float or double) is an important property for the *dyncall* library to establish the function in a correct way, the distinction between `signed` and `unsigned` integer data types (`char`, `short`, `int`, `long`, `long long`) in C is not of importance for the library itself. as these types share the same machine storage semantics in regard to register usage, size and alignment.

On a higher level, such as in the binding of a scripting environment, it is vital to have a correct conversion between different storage schemas among scripting languages. Therefore we define also the unsigned/signed variants for a later use in the language binding part.

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"
double	f4(int, bool, char, double, const char*);	"iBcdZ)d"

Table 3: Type signature examples of C function prototypes

4.2 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 4: Type signature encoding for Python bindings

4.3 R language bindings

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

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 5: Type signature encoding for R bindings

4.4 Ruby language bindings

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

Signature character	accepted Ruby data types
'B'	TrueClass, FalseClass, NilCalss, Fixnum casted to bool
'c'	Fixnum cast to char
's'	Fixnum cast to short
'i'	Fixnum cast to int
'j'	Fixnum cast to long
'l'	Fixnum cast to long long
'f'	Float cast to float
'd'	Float cast to double
'p'	String cast to void*
'v'	no return type

Table 6: Type signature encoding for Ruby bindings

5 Library Design

5.1 Design considerations

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

6 Developers

6.1 Project root

```
configure     -- configuration tool (unix-shell)
configure.bat -- configuration tool (windows batch)
ConfigVars    -- configuration tool output
BSDmakefile   -- BSD makefile
GNUmakefile   -- GNU makefile
Nmakefile     -- MS nmake makefile
LICENSE       -- license information
README.txt    -- general information
buildsys/     -- build systems ({BSD,GNU,N}make)
doc/          -- manual
dyncall/      -- dyncall library source code
dyncallback/  -- dyncallback library source code
dynload/      -- dynload library source code
test/         -- test suites
```

6.2 Test suites

plain Identity function calls for all supported return types and calling conventions, plus this C++ calls (GNU and MS).

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_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.

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.

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. Thus in order to avoid crashes, data corruption, etc., the user is urged 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 command line arguments to the call itself, or even worse, by indirectly choosing a library and a function to call. Such unchecked input data can be quite easily used to intentionally crash the program , or to hijack it and take control of the program flow.
To put it in a nutshell, if not used with care, programs depending on the *dyncall*, *dyncallback* and *dynload* libraries, can become arbitrary function call dispatchers by manipulating their input data. Successful exploits of programs like the one outlined above can be misused as very powerful tools for a wide variety of malicious attacks, . . .

7.2 Embedding

The *dyncall* library has a very low dependency to system facilities. The library uses some heap-memory to store the Call VM and uses per default `malloc()` and `free()` calls. This behaviour can be changed by providing custom `dcAllocMem()` and `dcFreeMem()` functions. 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 a CallVM (invoking the call is always thread-safe, though). However, since there's no limitation on the number of created CallVM objects, it is advised to keep a copy for each thread.

7.4 Supported types

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

7.5 Roadmap

The *dyncall* library should be extended by a wide variety of other calling conventions and ported to other, more esoteric platforms. With its low memory footprint it surely might come in handy on embedded systems. So far dyncall supports arm9e and mips32 (eabi) embedded systems processors. Furthermore, the authors plan to write some more scripting language bindings, examples, and other projects that are based on it.
Besides *dyncall* and *dyncallback*, the *dynload* library needs to be extended with .dylib, .so and other shared library format support (e.g. AmigaOS .library or GEM [14] 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 [15], libffcall [16] and C/Invoke [17].

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"` uses 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 7: 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 8: 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      dcFreeCallVM(DCCallVM* vm);
```

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

A.4 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).

Modes

Details

`DC_CALL_C_DEFAULT` is the default standard C call on the target platform. It uses the standard C calling convention and will also be used for variable argument ellipsis calls. On most platforms, there is only one C calling convention. Only the X86 platform provides a rich family of different calling conventions.

A.5 Machine state reset

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

Resets the internal stack of arguments. This function should be called prior to binding new arguments to the CallVM, because arguments don't get flushed automatically after a foreign function call invocation.

Constant	Description
`DC_CALL_C_DEFAULT`	C default function call
`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_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_ARM_ARM`	C arm call (arm mode)
`DC_CALL_C_ARM_THUMB`	C arm call (thumb mode)
`DC_CALL_C_MIPS32_EABI`	C mips32 eabi call
`DC_CALL_C_MIPS32_PSPSDK`	C mips32 default PSP Homebrew SDK call (uses eabi)

Table 9: CallVM calling convention modes

A.6 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.7 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

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.8 Formatted calls (ANSI C ellipsis interface)

Functions

```
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 issue a printf-style function call, using a signature string encoding the argument types and return type. The return value will be stored in `result`. For more information about the signature format, refer to 2.

B Dynload C library API

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

B.1 Loading code

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

B.2 Retrieving functions

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

C Calling Conventions

Before we go any further...

It is of **great** importance to be aware 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.

C.1 x86 Calling Conventions

Overview

There are numerous different calling conventions on the x86 processor architecture, like cdecl, MS fastcall, GNU fastcall, Borland fastcall, Watcom fastcall, Win32 stdcall, MS thiscall, GNU thiscall and the pascal calling convention, etc.

dyncall support

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

C.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 10: Register usage on x86 cdecl calling convention

Parameter passing

- stack parameter order: right-to-left
- caller 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 - this is required on darwin/i386 platforms)

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:

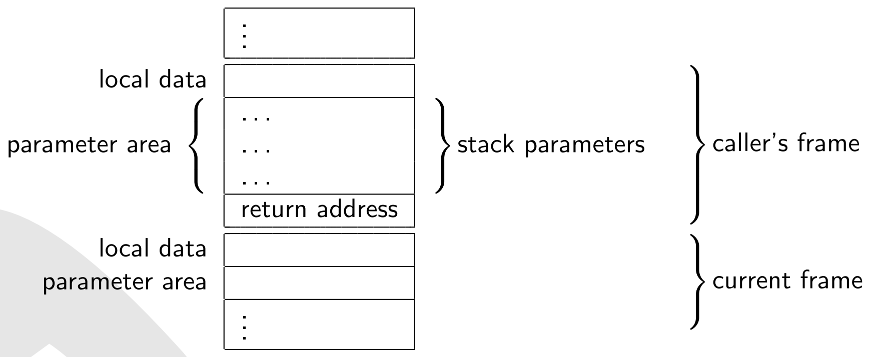

Figure 1: Stack layout on x86 cdecl calling convention

C.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 11: 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:

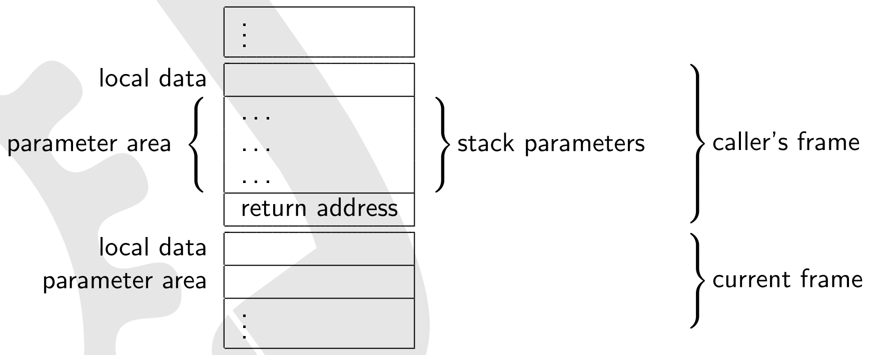

Figure 2: Stack layout on x86 fastcall (MS) calling convention

C.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 12: 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:

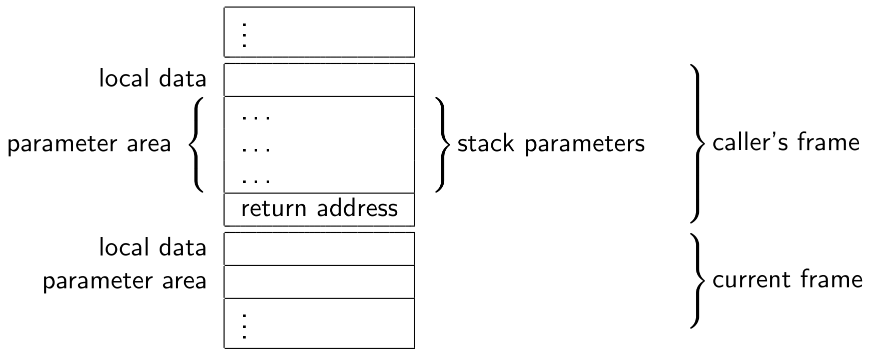

Figure 3: Stack layout on x86 fastcall (GNU) calling convention

C.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 13: 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:

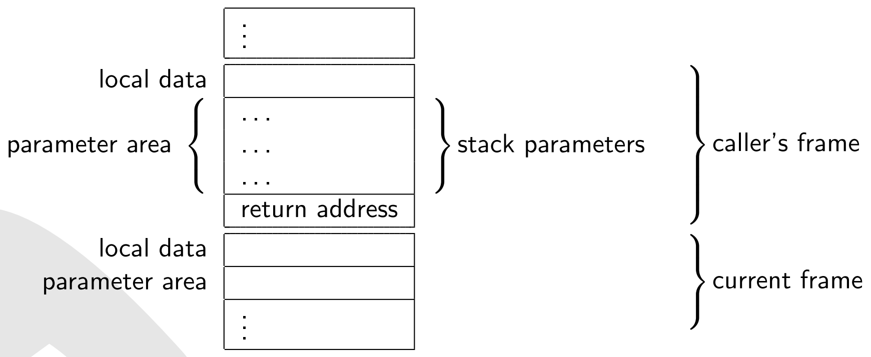

Figure 4: Stack layout on x86 fastcall (Borland) calling convention

C.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 14: 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:

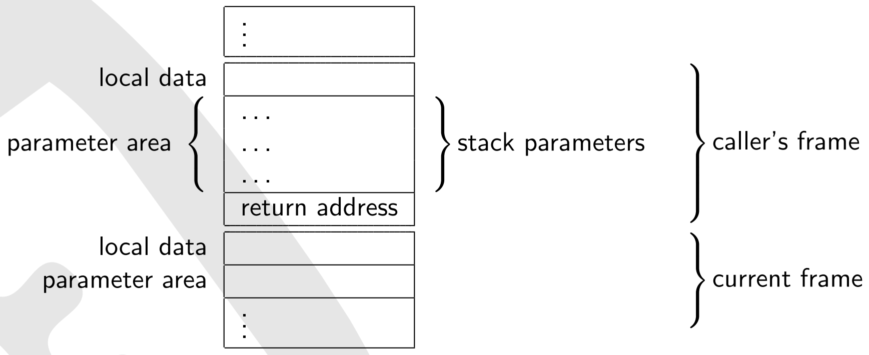

Figure 5: Stack layout on x86 fastcall (Watcom) calling convention

C.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 15: 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:

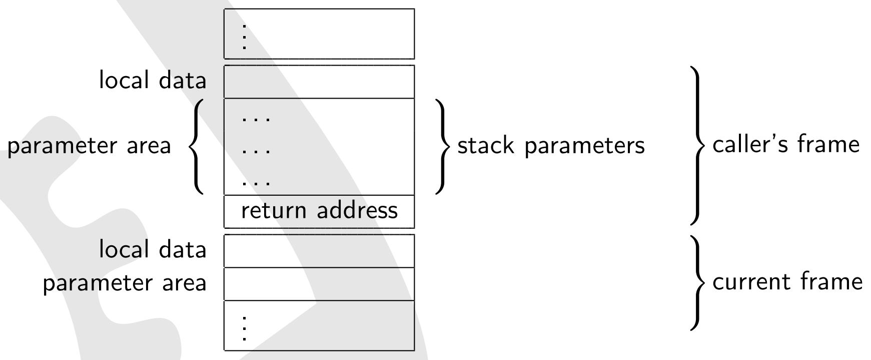

Figure 6: Stack layout on x86 stdcall calling convention

C.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 16: 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:

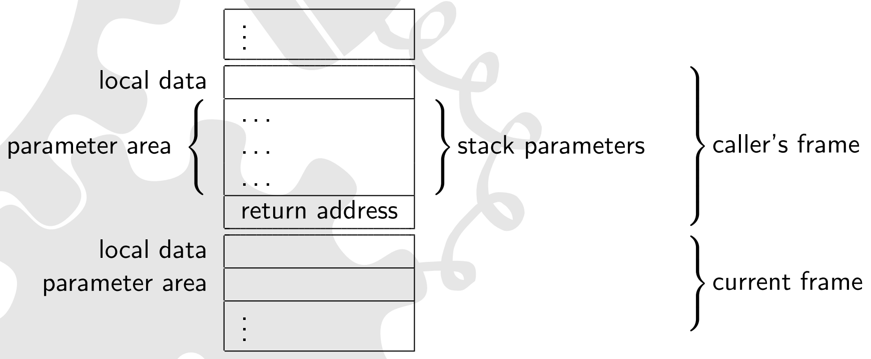

Figure 7: Stack layout on x86 thiscall (MS) calling convention

C.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 17: 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:

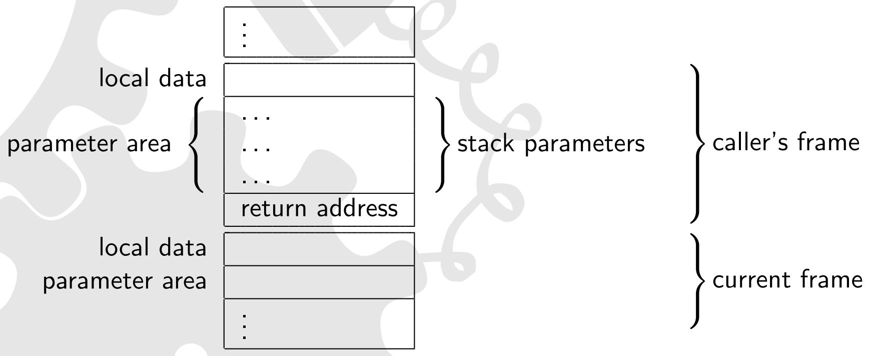

Figure 8: Stack layout on x86 thiscall (GNU) calling convention

C.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 18: 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:

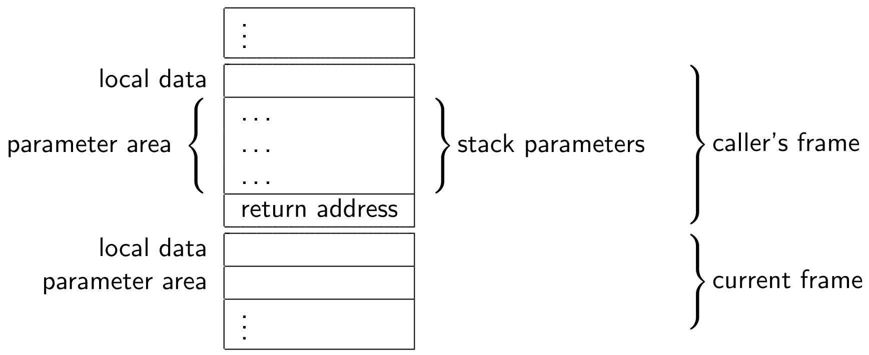

Figure 9: Stack layout on x86 pascal calling convention

C.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 [7] differs from the SystemV x64 calling convention [8] 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.

C.2.1 MS Windows

Registers and register usage

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.)

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 ase frame pointer
rsp	stack pointer
r8-r9	scratch, parameter 2 and 3 if integer or pointer
r10-r11	scratch, permanent if required by caller (used for syscall/sysret)
r12-r15	permanent
xmm0	scratch, floating point parameter 0, floating point return value
xmm1-xmm3	scratch, floating point parameters 1-3
xmm4-xmm5	scratch, permanent if required by caller
xmm6-xmm15	permanent

Table 19: Register usage on x64 MS Windows platform

- 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
- ellipse calls take floating point values in int and float registers (single precision floats are promoted to double precision as defined for ellipse 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:

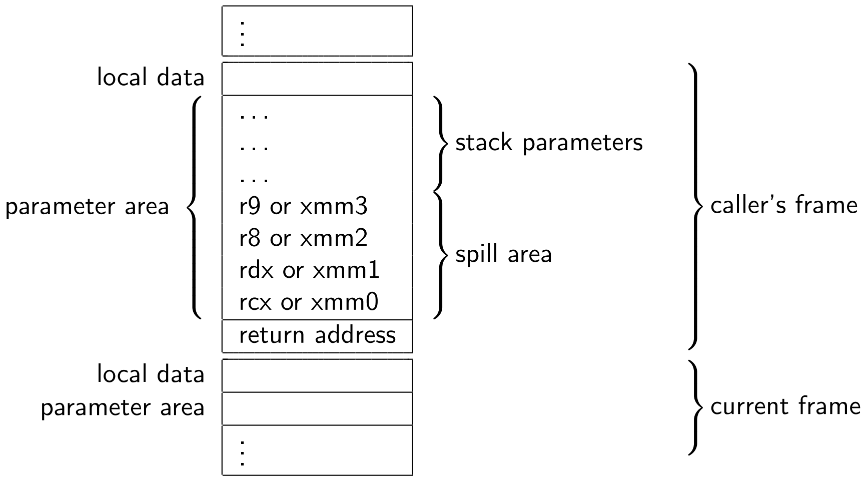

Figure 10: Stack layout on x64 Microsoft platform

C.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 ase 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 20: Register usage on x64 System V (Linux/*BSD)

Parameter passing

- stack parameter order: right-to-left
- caller cleans up the stack
- first 6 integer/pointer parameters are passed via rdi, rsi, rdx, rcx, r8, r9
- first 8 floating point parameters <= 64 bits are passed via xmm0l-xmm7l
- parameters in registers are right justified

- parameters that are not passed via registers are pushed onto the stack
- 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:

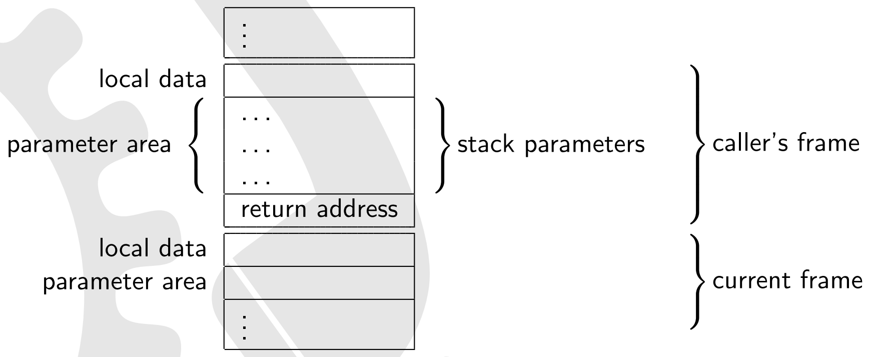

Figure 11: Stack layout on x64 System V (Linux/*BSD)

C.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". It uses Big Endian (MSB).
- Linux PPC 32-bit ABI is specified in "LSB for PPC 2.1" which is based on "System V ABI". It uses Big Endian (MSB).
- Ellipse calls do not work on Linux/System V PPC ABI currently.

dyncall support

Dyncall supports PowerPC (32bit) Big Endian (MSB) on Darwin (tested on Apple Mac OS X) and System V ABI systems (Linux, NetBSD, etc.).

C.3.1 Mac OS X/Darwin

Registers and register usage

Name	Brief description
gpr0	scratch
gpr1	stack pointer
gpr2	scratch
gpr3	return value, parameter 0 if integer or pointer
gpr4-gpr10	return value, parameter 1-7 for integer or pointer parameters
gpr11	permanent
gpr12	branch target for dynamic code generation
gpr13-31	permanent
fpr0	scratch
fpr1-fpr13	parameter 0-12 for floating point (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 21: 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)
- ellipse calls take floating point values in int and float registers (single precision floats are promoted to double precision as defined for ellipse calls)
- all nonvector parameters are aligned on 4-byte boundaries
- vector parameters are aligned on 16-byte boundaries
- integer parameters $<$ 32 bit occupy high-order bytes of their 4-byte area
- 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

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:

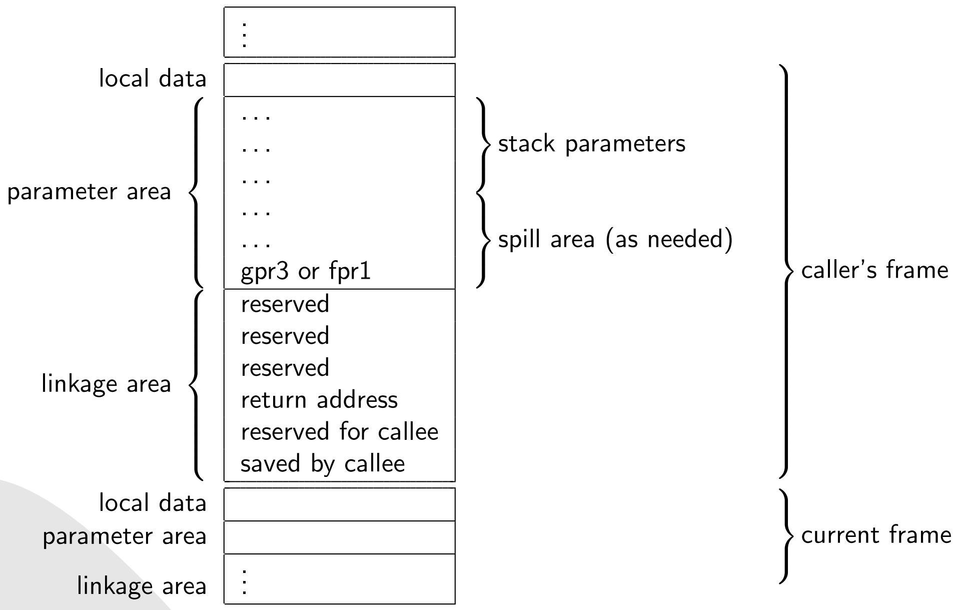

Figure 12: Stack layout on ppc32 Darwin

C.3.2 System V PPC 32-bit

Status

- Ellipse calls do not work.
- C++ this calls do not work.

Registers and register usage

Parameter passing

- Stack pointer (r1) is always 16-byte aligned.
- 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

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 22: 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:

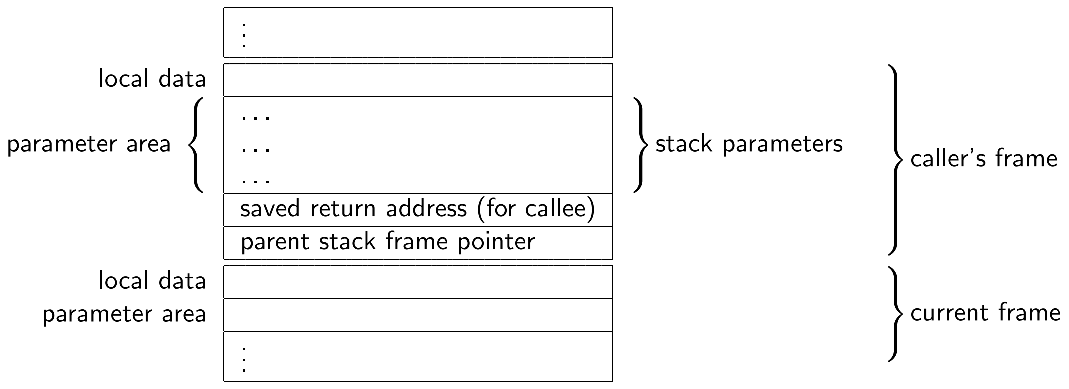

Figure 13: Stack layout on System V ABI for PowerPC 32-bit calling convention

C.4 ARM9E Calling Convention

Overview

The ARM9E family of processors is based on the 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

Take a look at the ARM-THUMB procedure call standard (ATPCS) [6] for more details.

dyncall support

Currently, the *dyncall* library supports the ARM and THUMB mode of the ARM9E family, excluding ARM-THUMB interworking. Although it's quite possible that the current implementation runs on other ARM processor families as well, please note that only the ARM9E 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.

C.4.1 ARM mode

Registers and register usage

In ARM mode, the ARM9E processor has sixteen 32 bit general purpose registers, namely r0-15:

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

Table 23: Register usage on arm9e

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 - GCC needs them to be aligned on 8 byte boundaries, 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 ARM9E 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:

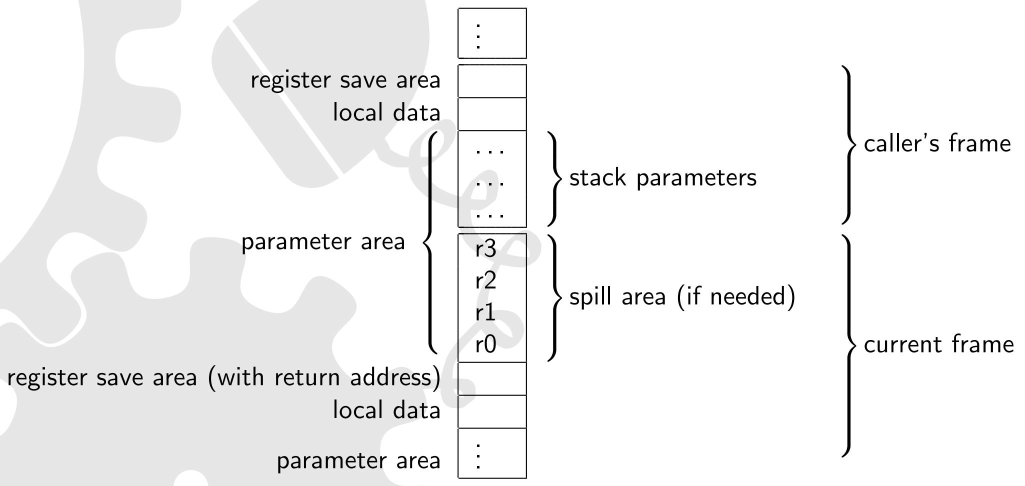

Figure 14: Stack layout on arm9e

C.4.2 THUMB mode

Status

- Ellipse calls do not work.
- C++ this calls do not work.

Registers and register usage

In THUMB mode, the ARM9E 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

Table 24: Register usage on arm9e 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 - GCC needs them to be aligned on 8 byte boundaries, 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 ARM9E 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:

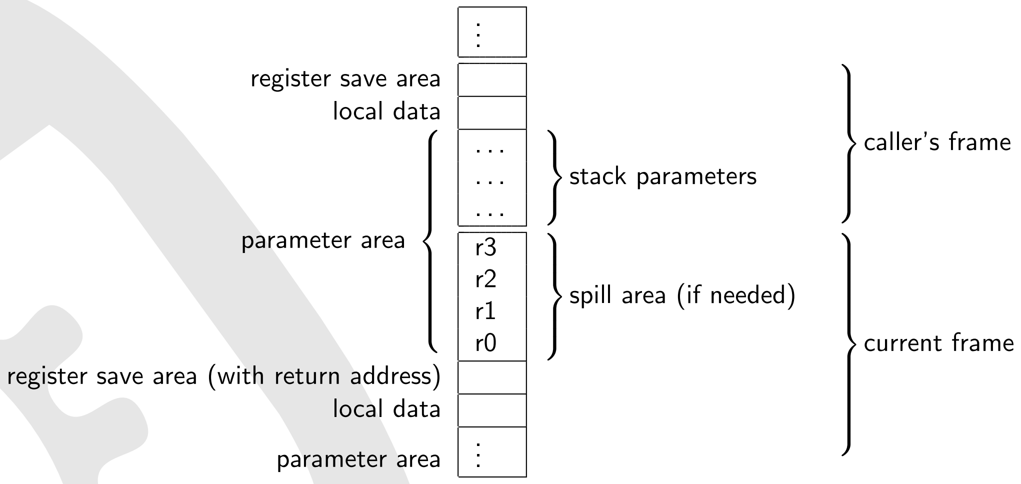

Figure 15: Stack layout on arm9e thumb mode

C.5 MIPS Calling Convention

Overview

The MIPS family of processors is based on the MIPS processor architecture. Multiple revisions of the MIPS Instruction sets, namely MIPS I, MIPS II, MIPS III, MIPS IV, MIPS32 and MIPS64. Today, MIPS32 and MIPS64 for 32-bit and 64-bit respectively.
Several add-on extensions exist for the MIPS family:

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 *32*, *O64*, *N32*, *64* and *EABI*.

dyncall support

Currently, dyncall supports the EABI calling convention which is used on the Homebrew SDK for the Playstation Portable. As documentation for this EABI is unofficial, this port is currently experimental.

C.5.1 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
$12-$15,$24,$25	**$t4-$t7,$8,$9**	Integer temporaries
$16-$23	**$s0-$s7**	Preserved
$26-$27	**$kt0-$kt1**	Reserved for kernel
$28	**$gp**	Global pointer
$29	**$sp**	Stack pointer
$30	**$s8**	Frame pointer
$31	**$ra**	Return address
hi, lo		Multiply/divide special registers
$f0,$f2		Float results
$f1,$f3,$f4-$f11,$f20-$f23		Float temporaries
$f12-$f19		Float arguments

Table 25: Register usage on mips32 eabi calling convention

Parameter passing

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

- Stack always aligned to 8 bytes.
- first 8 integers and floats are passed independently in registers using \$a0-\$a7 and \$f12-\$f19, respectively.
- if either integer or float registers are consumed up, the stack is used.
- 64-bit floats and integers are passed on two integer registers starting at an even register number, probably skipping one odd register.
- \$a0-\$a7 and \$f12-\$f19 are not required to be preserved.
- results are returned in \$v0 (32-bit integer), \$v0 and \$v1 (64-bit integer/float), \$f0 (32 bit float) and \$f0 and \$f2 (2 × 32 bit float e.g. complex).

Stack layout

Stack directly after function prolog:

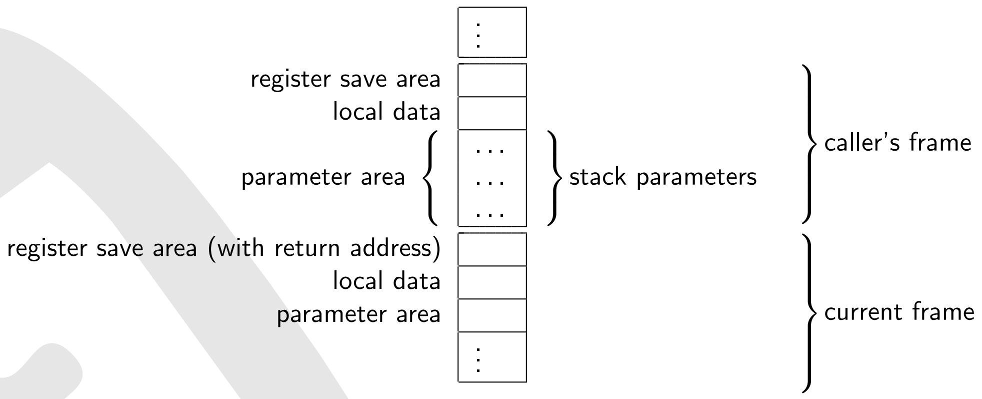

Figure 16: Stack layout on mips32 eabi calling convention

D Literature

References

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

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

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

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

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

[6] ARM-THUMB Procedure Call Standard
http://tinyurl.com/2rxb3a

[7] MSDN: x64 Software Conventions
http://tinyurl.com/2k3tfw

[8] System V Application Binary Interface - AMD64 Architecture Processor Supplement
http://tinyurl.com/2j5tex

[9] System V Application Binary Interface - SPARC Processor Supplement
http://www.sparc.com/standards/psABI3rd.pdf

[10] Introduction to Mac OS X ABI Function Call Guide
http://tinyurl.com/s7f85

[11] Linux Standard Base Core Specification for PPC32 3.2 - Chapter 8. Low Level System Information
http://tinyurl.com/9ttwcy

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

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

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

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

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

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