Dynamic Language Bindings for C Libraries with emphasis on their application to R

Dissertation
presented for the degree of Doctor rerum politicarum
at the Faculty of Business and Economics
of the Georg-August-Universität Göttingen

by

Daniel Adler

from Göttingen, Germany

Göttingen, 2012

First Examiner:	Prof. Dr. W. Zucchini
Second Examiner:	Prof. Dr. T. Kneib
Third Examiner:	Prof. Dr. C. Damm
Thesis defense:	23.05.2012

in memory of Gisela

Contents

List of Tables

List of Figures

List of listings

Typographic Conventions

Emphasized passages of text are printed in “italic” font style. Software `packages`, `libraries`, `filepaths` and internet links are printed in “type-writer” font style. Elements of computer languages are printed in “type-writer” font where the background color indicates on a given context and dialect such as document markup elements of `XML` and `XSLT`, programming language elements of `R`, `C` and `Assembly` and other plain `text` format. Further, code snippets and listings are printed with syntax highlighting using the `minted` LaTeX package and `pygmentize`[1]. Figure 1 gives the color scheme and examples of listings.

Language — Code example

Plain Text

```
:fun
setWindowRect(*<Window>*<Rect>)v;
newWindow(IB)*<Window>;
.
:struct
Window{:}
```

XSLT

```
  <template match="//Function">
    <value-of select="@name"/>
    <text>(</text>
    <apply-templates select="Argument"/>
    <text>)</text>
```

XML

```
<GCC_XML>
  <PointerType id="_24" type="_5"/>
  <FundamentalType id="_25" name="int"/>
  <Function id="_3" name="SDL_SetVideoMode" returns="_24">
    <Argument type="_25"/>
```

R

```
draw <- function(sim) {
  glClear(GL_COLOR_BUFFER_BIT)
  drawTexCirclesVertexArray(sim$x,sim$y,sim$r)
  glFinish()
  SDL_GL_SwapBuffers()
}
```

C

```
double tiny_dyncall(void (*fun)(), char const * fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  while ( ch = *ptr++ != ')' ) {
    switch(ch) {
      case 'd': dcArgDouble(vm, va_arg(ap, double)); break;
```

Assembly

```
amd64_sysv:
  pushq %rbp           # Prolog
  pushq %rbx
  movq  %rsp,%rbp
  movq  %r8, %rbx       # RBX = function pointer
  movsd 0(%rcx),%xmm0  # Load FPU register
```

Figure 1: Color highlighting conventions for code in this thesis.

[1]Sources of `pygmentize` were slightly modified for enhanced support of `gas` ARM Assembly syntax; Patches are available from the thesis homepage `http://dyncall.org/thesis`.

Software Sources

Several software packages that are subject of this thesis are available under the open-source license ISC (Internet Systems Consortium). Copyright information, download links and license text are given below:

- `dyncall` is available from the project site at `http://dyncall.org`.

  ```
  Copyright (c) 2007-2012 Daniel Adler <dadler@uni-goettingen.de>,
                          Tassilo Philipp <tphilipp@potion-studios.com>
  ```

  Parts of `dynload` and `dyncall` related to `dynload_mach-o.*` and `dyncall_struct.*`:

  ```
  Copyright (c) 2010-2011 Olivier Chafik <olivier.chafik@centraliens.net>
  ```

- `rdyncall` is distributed on CRAN at `http://cran.r-project.org/package=rdyncall`.

  ```
  Copyright (c) 2007-2012 Daniel Adler <dadler@uni-goettingen.de>
  ```

The license text is given below.

Acronyms

ABI Application Binary Interface. 2–4, 26, 39, 42, 66, 83, 139, 175

API Application Programming Interface. 1, 2, 4, 12, 13, 20, 21, 23, 26, 29, 31, 36, 39, 56, 66, 69, 91, 100, 102, 110, 117–119, 121, 123, 175, 178, 179, 181, 185, 195, 231, 245

ARM Advanced Risc Machines. 149

BNF Backus-Naur Form. 48

Call VM Call Virtual Machine. 178, 179, 181–183, 186, 187, 193–198, 202, 205, 207, 210, 212, 215, 220, 222, 223, 230, 242, 248, 249

CHRP Common Hardware Reference Platform. 156

COM Component Object Model. 3, 45, 46

CPU Central Processing Unit. 132

CRAN Comprehensive R Archive Network. 8, 69, 174, 259, 261

CUDA Compute Unified Device Architecture. 17

DLL Dynamically Linked Library. 83–85, 89, 148, 246

DSC Distributed Statistical Computing. 18

DTD Document Type Definition. 59

ELF Execution and Link Format. 83, 85, 88, 141, 239, 246

FDI Foreign Data Interface. 30

FFI Foreign Function Interface. 2–4, 23, 30, 33, 39, 46, 54, 55, 66, 72, 91, 92, 95, 100, 101, 107, 112, 117, 131, 175, 196

FFT Fast Fourier Transformation. 261

FP Frame Pointer. 136

FPR floating-point register. 143

FPS Frames Per Second. 122, 124, 127

FPU Floating-point unit. 132, 145, 152

GCC GNU Compiler Collection. 35, 56, 65

GLEW OpenGL Extension Wrangler library. 13

GLU OpenGL Utility library. 13

GPGPU General-Purpose Graphics Processing Unit. 16

GPL GNU Public License. 9

GPR general-purpose register. 143

GPU Graphics Processing Unit. 16, 122, 124

IDL Interface Definition Language. 46

ISA Instruction Set Architecture. 132, 203

ISC Internet Systems Consortium. XV

JIT Just-In-Time. 261

JNI Java Native Interface. 30

JSM Joint Statistical Meetings. 18

LR Link Register. 136

Mach-O Mach object file format. 84, 239, 246

MMU Memory Management Unit. 133, 238

MSVCRT Microsoft Visual C Run-Time. 80

OpenCL Open Computing Language. 17

OpenGL Open Graphics Library. 12, 13, 15

OpenGL ES Open Graphics Library for Embedded Systems. 13

PC Program Counter. 132, 136

PE Portable Execution Format. 83, 84, 239, 246

PReP PowerPC Reference Platform. 156

SDL Simple DirectMedia Layer. 15

SIMD Single Instruction Multiple Data. 132, 145, 152

soname Shared Object Name. 85–89

SP Stack Pointer. 136

SPARC Scalable Processor ARChitecture. 162

SSE SIMD Streaming Extension. 145

SWIG Simplified Wrapper and Interface Generator. 28

URI Uniform Resource Identifier. 60

XML Extensible Markup Language. 56, 58, 60

XPath XML Path language. 61

XPCOM Cross-Platform Component Object Model. 45, 46

XSD XML Schema. 59

XSLT Extensible Stylesheet Language Transformations. 56, 58, 60

Acknowledgements

My first and sincere appreciation goes to my supervisor Professor Dr. Walter Zucchini for the encouragement, patient guidance and advice he has provided throughout my time as his student, and since his retirement. I am deeply grateful for his support and commitment.

I also would like to thank Professor Dr. Thomas Kneib and Professor Dr. Carsten Damm for their spontaneous willingness to act as my second and third examiner.

This project would be impossible without the help and support other persons and organizations. Special thanks goes to Tassilo Philipp for the fruitful collaboration with him. I also want to thank the "GCC Compile Farm Project" project, the GWDG and Frank Scholl for providing access to hardware resources.

I am very grateful to Juliane Manitz for promptly proofreading. I also want to thank Jörg Evers, Cordula Kunkel, Frank Scholl and Friday for their support during my defense.

Lastly, I want to thank Cecy and my family for their support and love.

Chapter 1

Introduction

The use of scripting languages is becoming increasingly prevalent. Ever more users, from non-technical areas, are attracted by the low hurdles of *scripting* and the *rapid* development environments. The "R Project for Statistical Computing" gives a prime example of such an *open* platform for computational statistics and graphics which connects users world-wide.

Whether a specific language is suitable to solve a given task often depends on the quality and quantity of available extensions. Numerous efficient "best-practice" software solutions are available in the form of reusable software program libraries with a C API (Application Programming Interface). Often they are ported across hardware platforms and can be accessed as a dynamically loadable and linkable shared C library, such as OpenGL to utilize accelerated graphics hardware for real-time graphics rendering. Scripting users can benefit indirectly from these external C libraries if a *Language Bindings* extension exists for their language to provide for a scripting interface to the C API.
However, the development of *Language Bindings to C Libraries* can be expensive. Depending on the library the API can comprise hundreds of functions; for each of them a corresponding scripting wrapper needs to be implemented. Moreover, if callbacks and complex data structures are part of the C interface, the development becomes even more challenging. After all, the development is a *language-and-library* specific task; developers require detailed knowledge of both sides of an adapter, the C extension interface of the scripting language and the C API of the library. Since a language bindings extension comprises hybrid implementations, written in the scripting language and C, its development includes porting, compilation, testing and deployment for each supported target platform of the language; effectively, development tasks are needed for each and every *language-library-platform* combination.

In this thesis we discuss an extension model for scripting languages that provides for seamless scripting access to the C API of native libraries *without* the need for compilation of wrapper modules and thus significantly decreases the complexity of language bindings development. We give a detailed description of the model and an implementation for the R language. The model makes use of *dynamic* operations for interoperability with native code and data that are carried out at the machine level and that need to conform with the ABI (Application Binary Interface) and *Calling Conventions* of the processor hardware platform. We give an overview of ABIs across five processor-architecture families and then present a portable abstraction layer for making foreign function calls and handling of callbacks.

1.1 Overview

Whereas language bindings are usually implemented by means of C wrapper code which necessitates compilation for a specific *language-library-platform* combination, we suggest an alternative bindings model, named the *Dynamic Bindings Model*, that superseeds compilation to reduce the complexity where a C library can be made accessible across supported scripting languages and platforms by a *per-library* development task. The model describes a *single* extension module that needs to be developed *once* for a specific scripting language. This extension takes the role of an interface hub to C libraries; linkage and set up of scripting interfaces is carried out dynamically at run time, driven by a platform- and language-portable "C API bindings" specification format, named *DynPort*. The format serves as a *portable* text-based interface for type-safe interoperability with native C code and data across hardware platforms; abstract C type information of a library's API is encoded in a compact, intuitive text-based format that is suitable for bindings automation but also as a low-level scripting interface for native operations. Bindings for C APIs are collected in a repository that is extendable, and we give the design of an automation parser tool, based on open-source compiler, scripting and XML tools, that translates C API header files to *DynPort* files.

We then discuss an implementation of this model for the R programming language that is contributed as the `rdyncall` R package. The model requires the existence of a *Dynamic FFI (Foreign Function Interface)* to overcome compilation. However, the existing FFI of R has very limited support for calling native C code; only a small subset of C function types are supported and type-safety needs to be implemented manually. `rdyncall` provides a *Dynamic FFI* facility for R to be used as a flexible alternative for making calls to native C code of arbitrary C function type with only small limitations. It includes support for handling native C data objects and wrapping of R functions as callbacks. The implementation of this model is based on these services. We show that the C interfaces are mapped to R as-is and we illustrate this resemblance with a comparison of C API user code in C and R. We also give complex examples that emphasis platform-portable scripting of 3D multimedia applications in R using powerful libraries, such as `OpenGL` and `SDL`.

Basically we overcome compilation in this bindings model by utilizing dynamic operations of a *Dynamic FFI*, such as foreign function calls, handling and manipulation of C run-time data objects, and also wrapping of scripting functions as C callback pointers. They are usually implemented by using a software abstraction layer with a portable C interface, named *Generic Dynamic FFI*, that abstracts the low-level operations to carry out machine-level function calls that conform to a specific calling convention.

Little has been published about the *implementation* of a *Generic Dynamic FFI*, although this layer is substantial for *portable* implementation of FFIs and low-level middleware. This layer of abstraction is difficult to implement because it has to take details of the processor architecture, the ABI and calling convention of compilers into account.

In this thesis we give an overview of different ABI standards across five common processor-architecture families, namely `x86`, `powerpc`, `arm`, `mips` and `sparc`. We then discuss the design and implementation of a software abstraction layer for dynamic interoperability with precompiled code, contributed as open-source libraries of the *DynCall* project. The discussion of this abstraction layer completes the implementation design of the *Dynamic Bindings Model* for platform-specific issues. Table 1.1 gives an overview of the development history of the R package `rdyncall` and the C libraries of the *DynCall* project (`dyncall`, `dyncallback` and `dynload`).

Year	Details
2007	Adler developed prototyp of COM-based middleware for C++ components with R bindings on `x86-32` platform using type signature strings and call kernel. Adler and Philipp began to create an open-source library that abstracts calling conventions via a dynamic C interface for making calls to precompiled functions.
2008	Public release of `dyncall` version 0.1 comprising two libraries `dynload` and `dyncall`, and ports to `x86-32` and `x86-64` for Microsoft Windows and Unix-based SVR4 Platforms, Apple Mac OS X `ppc32` , `arm32` for Nintendo DS and `mips32 EABI` for Sony Playstation Portable. A number of low-level bindings to programming languages were contributed on the Subversion repository for R, Python, Lua, Ruby. `dyncall` version 0.2 contains an improved documentation.
2009	Support for `ppc32` on SVR4 and ARM `thumb` mode was added in version 0.3 of `dyncall`. Initial release of `dyncallback` library with support for `x86-32`, `x86-64` and Apple Mac OS X `ppc32` as of `dyncall` version 0.4. Lua package `luadyncall` was started that uses DynPort 2.0 format. On useR! 2009 `rdyncall` including DynPort was presented to the public.
2010	DynCall version 0.5 added more ports for `dyncallback` and improved `arm32` port. Chafik contributed improvements for `dyncallback` (Microsoft Windows `x64` port) `dynload` and initial support for passing composite data types in `dyncall` that is used in the Java/C++ middleware `BridJ` which uses `dyncall` for the implementation at low level.
2011	Public release of R package `rdyncall` version 0.7 on CRAN with dynamic R bindings to legacy/modern `OpenGL`, `SDL` library family and others.
2012	`dyncall` version 0.7 adds port for `sparc32`, `sparc64` and `mips64` platforms, abstraction of assembly sources (`portasm`) and hybrid GNU/BSD/Sun Make build system.

Table 1.1: History of `dyncall` and related projects.

1.2 Outline of Thesis

Chapter 2 gives background information on programming languages, such as C and R, and portable C libraries, such as `OpenGL` and `SDL`, followed by the motivation for this thesis. Chapter 3 describes the software architecture for an extension to dynamic programming languages that gives scripting access to C libraries without the need for compilation of language bindings wrapper and which can work across platforms. We introduce the model with a general overview on loading and linking shared libraries, and also interoperability with precompiled code. Then follows a discussion on the creation of language bindings to C libraries, outlining shortcomings of compiled language bindings and static foreign function interfaces, such as given by the built-in FFI of R. Our *Dynamic Bindings Model* was developed to overcome these shortcomings. There follows a description of language-neutral and platform-portable components of the model; we give a brief overview of the abstract C type system and then present a compact encoding format for C APIs and a parser framework for automation. An implementation of the model for the R programming language, contributed by the R package `rdyncall`, is discussed in Chapter 4. First, we describe the foundation layer comprising components of a *Dynamic FFI for R.* At appropriate points in the course of the description, we give side-by-side examples of R and C user code for C libraries, such as `SDL`, `OpenGL` and `Expat`, to compare the syntactic structure and to illustrate their similarity. The chapter closes with an example of an R user application that makes use of `OpenGL` and `SDL` via `rdyncall`; this serves to illustrate platform-portable R scripting of system-level applications without compilation. The platform-specific abstraction layer and its implementation across processor architectures that is needed by *Dynamic FFIs* is discussed in Chapter 5. The anatomy of function calls at native machine level is discussed, followed by a survey of platform-specific ABIs and Calling Conventions across five processor-architecture families. The three libraries of *the DynCall project*, namely `dyncall`, `dyncallback` and `dynload`, are described in detail, which provides the abstraction layer to platform-specific details for the implementation of *Dynamic FFIs* and the *Dynamic Bindings Model.* We discuss the chosen software design, including the portable C API, and we compare implementation ports across a range of processor-architecture familes and operating systems. Our conclusions are given in Chapter 6.

Chapter 2

Background and Motivation

In this chapter we discuss the difference between compiled and interpreted languages and emphasise their different strengths. In particular, we focus on the R programming language and on powerful libraries with a C interface, such as `OpenGL` and `SDL` for the development of platform-portable real-time graphics application. We then describe the design of an existing R extension, named `rgl`, that contributes a 3D real-time graphics visualization device system with interactive navigation to R that was written in C++ using OpenGL and platform-specific interfaces to the windowing system; we outline the software design and compare its current state with ongoing developments of OpenGL, including issues of porting the package across major R platforms. This marks the starting point for the motivation of this thesis because it suggests a different development model for language extensions based on dynamic bindings to make portable C libraries become *scriptable* components across platforms.

2.1 Programming Languages

Ada Lovelace is credited with having written the world's first computer program. In her article, written between 1842-43, on a translation of memoirs from Luigi Menabrea about Charles Babbage's *Analytic Engine*, she added specifications for the calculation of Bernoulli numbers using the Engine (Fuegi and Francis, 2003). The Engine was among the first mechanical computers, but Babbage did not manage to finish building it.

The first freely programmable "stored-program" computers, as we know them today, were developed no earlier than in the late 1940s. The fundamental programming interface to these machines has not changed since then; it is "machine code": a sequence of numbers in storage memory which encodes a sequence of primitive hardware operations including arithmetic, load/store operations, control-flow branches, subroutine calls and conditional blocks of execution. Assembly language is one of the first programming languages that gives text-based access to the binary machine interface. As computers evolved, ever new machine-level interfaces were invented. Thereby, programs written in assembly language were architecture-specific and needed to be rewritten in a different dialect for new hardware platforms. In addition to the incompatibility of assembly languages between machines, programmers realized that this language is not convenient for writing complex software as it addresses a programming style at too low a level. A competition of language designs began. In the 1950s FORTRAN (later renamed Fortran) was one of the first programming languages which offered a high-level programming interface to the machine. At that time languages were compiled. A year later LISP was introduced and quickly became one of the first interpreted programming languages. At the end of the 1960s the UNIX operating system (later renamed Unix) and the C programming language were invented; both are getting on in years but still provide the software backbone for platforms, services, components, applications, and the Internet of today. By 1969 about 120 "important" languages existed but most of them are no longer in use (Mashey, 2004).

In the last two decades a wave of programming languages and environments has evolved; it has initiated a new era of "programming" and has brought users from different fields in academia, industry and the general public closer together.

2.1.1 Scripting

Ousterhout's 1998 paper "Scripting: Higher-Level Programming in the 21th Century" emphasises the importance of *scripting* languages. He dicotomizes the landscape of programming languages roughly into *system programming languages* and *scripting programming languages*, and compares these language categories to explain why scripting languages handle many of the programming tasks better than system programming languages. Table 2.1 gives a summary of his comparison.

Aspect	System Programming	Scripting
Languages	C, C++, Java, Objective-C	Tcl, Perl, Python, R, Lua
Purpose	Data Structures and Algorithms	Connecting components
Typing	Static	Dynamic
Performance	Fast execution	Rapid development
Learning Curve	steep, weeks to months	flat, hours to days
User	Experienced Software Developer	Casual Programmer and Developers

Table 2.1: Comparison of system programming languages and scripting languages based on Ousterhout (1998).

He considers both language groups to be in a complementary and symbiotic relationship and suggests combining them in a "gluing component framework". System programming languages, such as C and C++, should be used for the implementation of reusable core *components*. The strongly typed nature of such languages helps to manage complexity of data structures and algorithms, and the compilation of code increases execution speed. Scripting languages are used for gluing components as applications. Ousterhout argues that scripting languages allow *rapid* development of gluing-oriented applications. He supports his argumentation by an empirical study which compares the number of code lines and the development time for a variety of software projects, differing by size (three weeks to 180 month) and application domain (Database, Security Scanner, Simulation and Graphical User-Interfaces). Each application was implemented in two rounds using a system programming language (C++, C, Java) and a scripting language (Tcl, Perl). In every case applications written in the scripting language required far fewer lines of code (by a factor between 2 and 47) and less development time (by a factor between 3 and 60). He concludes that "the true difference between the two language types is more like a factor of five to 10" (Ousterhout, 1998).

The "scripting" approach that he suggests can be found in a variety of current software architectures. With the Netscape Browser release as open-source by Netscape Communications in 1998, a community of developers began to re-engineer the source base. A little later the Mozilla 1.0 platform was born, a platform that fosters general-purpose *Rapid Application Development*; applications are written in JavaScript which is connected with a component model (XPCOM) that gives access to core components implemented in C++. Example applications that run on the platform are the Firefox browser and the Thunderbird mail client. The architecture encourages extension writing, so it is not surprising that a large repository of add-ons exists for Firefox and Thunderbird. Futher details are given in McFarlane (2003).

Scripting has developed as is evident in the Web of today. Server-side scripting was available from early on for generating dynamic web content. But nowadays the Web is scripted at both ends. In 1995, the web browser Netscape Navigator 2.0 included a JavaScript interpreter to execute small scripts as part of a web page content. At that time client-side JavaScript had little connectivity with the components of a web browser. But the number of new interfaces has exploded in the past years. Today JavaScript is an integral part of almost all web browsers; it gives web developers fine-grained control over the browser-side user interface, network services as well as low-level graphics and audio devices. For a growing number of web sites JavaScript is a requirement; it is increasingly used to control the *rendering* of the complete web page in order to provide a responsive and flexible user interface.

2.1.2 R Language

A prime example of "scripting" in the academic field is the *R Project for Statistical Computing.* What started as a tool for statisticians has evolved into a general-purpose programming environment with many users from different research fields. The core of R is in fact a scripting language, and users who start to work with R as a *tool* often find themselves writing their own functions within hours or days. The R language (R Development Core Team, 2012a) is an interpreted, functional programming language with a single reference implementation. The software architecture is of a modular design for development of extensions and graphics output devices. The interpreter, graphics devices and core functions for computation are implemented in compiled languages, such as C and Fortran, while the "gluing" part is mostly written in the R programming language. R is delivered with a basic set of computational methods primarily focused in the area of statistics, but the core is extendable by packages. The CRAN (Comprehensive R Archive Network) is a repository of extension packages, mirrored across continents with currently 85 servers that offer users a convenient and quick way to download and install R packages. The user can choose among the currently available 3700 packages that can be installed within seconds. Researchers have recognized the power of this platform and so it is not surprising that many of current *state-of-the-art* computational methods - in particular in statistics - are available as R packages from CRAN. Package authors upload their package source code and the CRAN build server precompiles these for Windows, Mac OS X and Linux and then distributes them in both source and prebuilt binary form on the mirror network.

2.1.2.1 History

The development was started in 1993 (Ihaka and Gentleman, 1996) as a free re-implementation and open-source alternative to the proprietary S language, a programming system for statistics invented at the AT&T Bell Laboratories in the 1970s by Chambers et al. "Our [Becker et al.] primary goal was to bring interactive computing to bear on statistics and data analysis problems."(Becker, 1994) The aim of the language, as expressed by Chambers, is "to turn ideas into software, quickly and faithfully"(Chambers, 1998). Table 2.2 gives an overview of the notable historical events relating to R and S.

2.1.2.2 Platforms

R is an open-source software that has been ported to many current platforms. The portable design is a key factor for R's success as a universal platform for scientific computing and exchange of software-based research methods; it does not force users to use a particular operating system; rather it unifies users of different desktop systems.

2.1.2.3 Extensions

R offers a large collection of packages that extend the core language with new functionality. A number of packages make services available to R from C libraries. Most of these packages make use of a two-sided wrapper implementation using R and C code, where the latter needs to be compiled and linked with external C libraries for each platform.

In general, R packages which contain C code and use external libraries are harder to implement and maintain. The package author has to understand the differences between operating systems, e.g. Mac OS X, Linux, BSD derivates, Solaris (Unix-based) and Windows systems, and either needs to incorporate the external library or to negotiate build strategies with administrators of the CRAN build server.

Date	Details
1975	Becker, Chambers and Wilks began the development of an interactive environment for data analysis and graphics.
1977	S 1.0 was released.
1984	Brown S Book "An Interactive Environment for Data Analysis and Graphics" (Becker and Chambers, 1984) was published, describing a Macro-based extension language.
1988	Blue S Book "The New S" (Becker et al., 1988) was published, featuring user-written extensions as first-class objects. Commercial implementation of S, named "S-Plus", was offered by a Seattle-based start-up company.
1990	Gentlemen and Ihaka decided to write a LISP-based interpreter to evaluate ideas and to "publish a paper or two".
1991	White S Book "Statistical Models in S" (Chambers and Hastie, 1991) was published, featuring Classes.
1992	Ihaka and Gentlemen decided to adopt the syntax of S. As a joke, the name "R" was coined for the language.
1993	Statistical Sciences merged with MathSoft and acquired the exclusive license to distribute S.
1994	Initial release of R under GPL license appeared on the Internet.
1997	R became an official part of the GNU project. CRAN went online. R Core Team was founded.
1998	Green S Book "Programming with data: a guide to the S language" (Chambers, 1998) was published, describing a more rigorous class system. Chambers received the ACM Award for his S language. Chambers and Lang started the OmegaHat project.
2000	R 1.0 was released. S-PLUS 6 was ported to Unix-based platforms.
2001	Newsletter "R News" started to publish short and medium length articles. S-PLUS was sold to Insightful.
2007	Revolution Analytics was founded offering commercial support via custom R distribution "Revolution R".
2008	R Journal was founded, superseding R News. TIBCO acquired Insightful Corporation.
2011	R version 2.14 was released.

Table 2.2: History of R, S and S-PLUS.

2.1.3 C Language

C is a statically typed and compiled programming language that is regarded as the *lingua franca* of system-level programming with a long history of development. Table 2.3 gives an overview of the past 40 years of its development in which the language has become a standard for software development and was refined through several ratification processes. The origin of C can be found within the history of the UNIX operating system: "MULTiplexed Information and Computing Service (MULTICS), which is considered the precursor of the UNIX operating systems, came about from a joint venture between MIT, Bell Laboratories, and the General Electric Company (GEC), which was involved in the computer-manufacturing business at that time. The development of MULTICS was born of the desire to introduce a machine to support numerous timesharing users. At the time of this joint venture in 1965, operating systems, although capable of *multiprogramming* (timesharing between jobs), were batch systems that supported only a single user. The response time between a user submitting a job and getting back the output was in the order of hours. The goal behind MULTICS was to create an operating system that allowed *multiuser timesharing* that provided each user access to his own terminal. Although Bell Labs and General Electric eventually abandoned the project, MULTICS eventually ran in production settings in numerous places. UNIX development began with the porting of a stripped-down version of MULTICS in an effort to develop an operating system to run in the

PDP-7 minicomputer that would support a new filesystem. The new filesystem was the first version of the UNIX filesystem. This operating system, developed by Ken Thompson, supported two users and had a command interpreter and programs that allowed file manipulation for the new filesystem. In 1970, UNIX was ported to the PDP-11 and updated to support more users. This was technically the first edition of UNIX. In 1973, for the release of the fourth edition of UNIX, Ken Thompson and Dennis Ritchie rewrote UNIX in C (a language then recently developed by Ritchie). This moved the operating system away from pure assembly and opened the doors to the portability of the operating system."(Rodriguez and Fischer, 2006, Section 1.1)

Date	Standard	Details
1966	BCPL	Richards designed BCPL at Cambridge University.
1969	B	Thompson developed "B" at Bell Labs with contributions from Ritchie. Ritchie started development of "C". UNIX development started by Thompson, Ritchie and others on the "little-used PDP-7 in a corner" at Bell Labs.
1971	UNIX V1	First edition of UNIX on PDP-11/20 written in assembly.
1973	C, UNIX V4	Ritchie designed the first version of the "C" language at Bell Labs. Fourth Edition of UNIX was finished, rewritten in C.
1975	UNIX V6, BSD	UNIX Sixth Edition (Version 6) left Bell Labs.
1978	K&R C	Kernighan and Ritchie published the first edition of "The C Programming Language", often called the 'white book' or "K&R".
1982	ANSI WG, System III	ANSI formed a committee to standardize C named ANSI X3J11.
1983	System V	AT&T announced the first supported release of UNIX.
1984	4.2BSD	University of California at Berkeley released 4.2BSD.
1989	ANSI C, C89, SVR4	First ratification of C as ANSI X3.159-1989 "Programming Language: C". UNIX System V Release 4 ships, unifying System V, BSD and Xenix.
1990	ISO C, C90	ISO adopted the ANSI C standard as ISO/IEC 9899:1990. (Working Group WG14 Commitee).
1995		Normative Amendment 1 was published and added support for international character sets.
1999	C99	The second official ISO standard (ISO/IEC 9899-1999) added support for `inline` functions, `_Bool` data type, `long long` integer qualifier and `_Complex` floating-point qualifiers and variadic function-like macros.
2008	Embedded C	A standard for embedded systems was published adding support for fixed-point arithmetic and named address spaces.
2011	C11 (C1X)	The third official ISO standard *ISO/IEC 9899:2011* (ISO, 2011) added support for generic type macros and multi-threading, published on 2011-12-08.

Table 2.3: History of C.

UNIX became widely popular: "There are good reasons for this popularity. One is portability: the operating system kernel and the applications programs are written in the programming language C, and thus can be moved from one type of computer to another with much less effort than would be in involved in recreating them in the assembly language of each machine. Essentially the same operating system therefore runs on a variety of computers, and users needn't learn a new system when new hardware comes along. Perhaps more important, vendors that sell the UNIX system needn't provide

new software for each new machine; instead, their software can be compiled and run without change on any hardware, which makes the system commercially attractive."(Pike and Kernighan, 1984, p.1)
C specifies an elaborate type system in which a number of built-in data types is defined without giving concrete specification about its implementation on the machine. In that the compiler defines the exact mapping to machine data types for a particular target platform, the C language can be utilized on a broad spectrum of hardware architectures. As a consequence, C compilers exist for a large number of hardware platform families, such as for 32- and 64-bit server- and desktop systems and also for 8-, 16- and 32-bit microcontroller boards, and mobile- and embedded platforms.
A number of modern general-purpose programming languages, such as C++ and Objective-C, are largely based on C; they adopt, in particular, the fundamental type system of C, including the syntactic structure and semantics of expressions for fundamental data types. The integration of C is not a coincidence; languages which are compatible with C can be linked with external C libraries, and thus enable users to make use of the large pool of available C libraries that has been created over the past decades and are still growing. This makes modern languages applicable for a wide range of applications from the very beginning.
Software developers often choose C as the lowest common denominator among languages for a portable and efficient reference implementation of algorithms and protocols. Also scripting language interpreters are often implemented in C, such as in the case of the reference implementation of Lua (Ierusalimschy et al., 1996), Perl (Wall, 2000), Python (Van Rossum and Drake Jr, 1995), R (Ihaka and Gentleman, 1996), Ruby (Matsumoto, 2002) and Tcl (Ousterhout, 1990). As a side effect such languages offer a low-level extension interface accessible from C that can be used to link existing C libraries with scripting languages.

2.1.4 Current Trends

We close this brief overview of programming languages with excerpts of two popularity indices of programming languages. The web-site *the Transparent Language Popularity Index* (Language Popularity Index Project, 2012) publishes a ranking index on a monthly basis. The index is created by an open-source tool, described transparently on the website; it aggregates the monthly activities of several large online sources, such as Freecode (Freshmeat), Ohloh, Craigslist, Google Code, Powells, Delicious and Yahoo Search. Table 2.4 gives an excerpt of the *normalized* comparison of the top 20 most popular languages, and the top 10 languages classified as either *general-purpose* (including system level) and *script* languages.

Another online index opened with the "April Headline: Java and C swap places at the top of the TIOBE index"(TIOBE, 2012). We give the first six positions of the total rank briefly: C (17.555%), Java (17.026%), C++ (8.895%), Objective-C (8.236%), C# (7.348%) and PHP (ranked as the first scripting language) (5.288%); R is ranked 31st (0.380%). The TIOBE index "counts the hits of the most popular search engines [..] from the top 9 websites."(TIOBE, 2012, in "TIOBE Programming

TOP 20	**All Categories**	
Rank	Language	Share
1	C	17.175
2	Java	16.500
3	Objective-C	9.682
4	Basic	8.636
5	C++	6.564
6	PHP	5.353
7	C#	3.938
8	Python	3.643
9	Perl	3.351
10	JavaScript	1.913
11	Delphi	1.625
12	Ruby	1.528
13	R	1.248
14	D	1.214
15	Pascal	1.079
16	CL (OS/400)	1.051
17	Ada	0.862
18	Go	0.719
19	Logo	0.671
20	Fortran	0.594

TOP 10	**General-Purpose**	
Rank	Language	Share
1	C	23.789
2	Java	22.854
3	Objective-C	13.410
4	Basic	11.962
5	C++	9.092
6	C#	5.455
7	Delphi	2.251
8	D	1.681
9	Pascal	1.495
10	Ada	1.194

TOP 10	**Script**	
Rank	Language	Share
1	PHP	22.854
2	Python	15.553
3	Perl	14.306
4	JavaScript	8.169
5	Ruby	6.525
6	R	5.327
7	CL (OS/400)	4.486
8	Lisp/Scheme	2.319
9	MATLAB	2.157
10	Lua	2.087

Table 2.4: The Transparent Language Popularity Index for April 2012.

Community Index Definition")

2.2 Software Libraries

The complexity of large software projects is often countered by breaking down large tasks or problems into smaller ones. For recuring tasks and solutions a generalized and reusable software library is often developed in the open-source world. Countless libraries with a portable C API now exist that provide a comprehensive solution framework for a specific problem domain. In the following we give a brief overview of two popular C libraries that provide an abstraction layer to high-performance graphics hardware, desktop windowing-systems and multimedia devices.

2.2.1 OpenGL

OpenGL (Open Graphics Library) offers a cross-platform programming interface to dedicated graphics hardware for the implementation of 3D graphics applications including computer games, scientific visualization software or sophisticated user-interfaces. A comprehensive guide for programming with the OpenGL C API is given in OpenGL Architecture Review Board and D. Shreiner and et al (2005).

The development began in 1992 by Silicon Graphics Inc., a company that focused on hard- and software solutions for real-time graphics. Today almost all desktop systems support the OpenGL interface to graphics accelerators, such as Microsoft Windows, Apple Mac OS X, and the X Window System of Solaris, Linux and BSDs, while graphics card vendors, such as Nvidia and ATI, provide the implementation in form of graphics card drivers for their hardware products. The latter often extend the OpenGL interface with new extensions that make use of hardware improvements. In addition to the core OpenGL API, a number of helper libraries exists, such as GLU (OpenGL Utility library) for rendering of high-level graphics primitives, or GLEW (OpenGL Extension Wrangler library) for providing a portable interface to load specific versions of OpenGL and extensions.

Over the last two decades the library experienced several revisions, specialized branches and vendor-specific extensions. For example, OpenGL ES (Open Graphics Library for Embedded Systems) is a variant of OpenGL designed for mobile devices such as game consoles, smart phones and tablets.

Altogether, OpenGL provides an efficient cross-platform interface to a high-performance graphics processing pipeline model; it does not offer a portable interface for setting up an OpenGL context and corresponding display output surface. These two initial steps are left to platform-specific APIs that need be provided by the respective windowing system.

2.2.1.1 From the Fixed-Function Graphics Pipeline to freely Programmable GPUs

Table 2.5 gives an overview of the development from the original OpenGL 1 API in 1992 to the current OpenGL 4 API. After the initial release of OpenGL a large number of extensions were added to accommodate the rapid progress in graphics hardware design (see Lengyel (2003) for a comprehensive overview). New C functions and symbolic constants were incorporated to the original OpenGL 1.0 API as part of extensions that address new graphics effects and their parametrization (e.g. new blending and texture-environment modes, improved lighting, shadow mapping, point sprites), optimizations of the data flow within the graphics processing pipeline (e.g. vertex pointers and buffer objects) and other issues mostly related to improvements in graphics hardware capabilities (e.g. compressed texture data formats, vertex and fragment programs, stencil operations).

The most significant change came with OpenGL 2.0 and the introduction of the OpenGL Shading Language which enables users to specify short programs that are executed per vertex (to freely define transformations and projections) and per pixel (to implement custom graphics shading effects). See Rost (2004) for a detailed guide of the language.

The graphics hardware has been transformed from specialized hardware designs for particular graphics effects to a freely programmable co-processor that works on streams of vector data types. Over the years the OpenGL API has been tidied up to accommodate this development. The current version comprises the OpenGL Shading Language and a core set of C interface functions for resource managament (compiling and linkage of shader objects, and uploading/downloading of data to texture samplers, vertex buffers and framebuffers). Large parts of the original 1.0 API and numerous extensions have been declared deprecated, such as *immediate mode* rendering commands (in favor of using vertex buffers) and support for *display lists* (prerecording of sequences of API calls) were dropped. Other APIs based on OpenGL, such as OpenGL ES and WebGL, adopted this *mean* programming model[1]. The development process of the OpenGL API emphasizes the importance for scripting language bindings to address adaptability to the evolution of the low-level C programming interface.

Versions			
API	Shader Language	Year	Details
IrisGL	-	1982	Clark started one of the first Computer Graphics companies, named Silicon Graphics (later renamed SGI), which invented Irix OS and IrisGL (Integrated Raster Imaging System Graphics Library).
1.0	-	1992	Release of the open-source version of IrisGL API, named OpenGL, which became an industry standard. First version provided the basic fixed-function graphics pipeline model.
1.1	-	1997	Vertex arrays and texture mapping support was added.
1.2	-	1998	Texture mapping improvements (3D and multi-level filtering/mipmapping) were added. Separate specular color component was added for improved lighting model as well as improvements for image processing (e.g. convolution matrices and support for color space transformation).
1.3	-	2001	Extensions were incorporated to the standard API, such as cube textures, multi-sampling of texture data, multi-texturing and bump mapping.
1.4	-	2002	Further extensions were incorporated, such as depth textures for shadow mapping.
1.5	-	2003	Vertex Buffer Objects were added to improve transfer of vertex data.
2.0	1.10	2004	The OpenGL Shading Language was added as a standard, including compilation of vertex and fragment shader programs, as an alternative to the fixed-function graphics pipeline model.
2.1	1.20	2006	Data transfer of pixel data was improved via incorporation of Pixel Buffer Objects.
3.0	1.30	2008	With this verison a large subset of the API, that addresses the fixed-function graphics pipeline model, were declared as deprecated (but still supported). It was also added support for floating-point texture formats and framebuffer objects. The latter allows rendering output to offscreen and texture memory.
3.1	1.40	2009	The API incorporates a strict separation between the modern and the deprecated API functions. It also included support for 1D texture buffer objects and uniform variable buffer objects.
3.2	1.50	2009	Introduction of profiles for deprecated and modern API.
3.3	3.30	2010	Simultaneous release with 4.0.
4.0	4.00		Two new stages of the graphics pipeline were opened for shader-based tesselation of higher level geometry.
4.1	4.10	2010	Full compatibility with OpenGL ES 2.0 with support for precompiled shader programs.
4.2	4.20	2011	API included support for 32-bit packed data types.

Table 2.5: Revisions of OpenGL.

[1]For a comparision of different sizes of the OpenGL API, see Table 4.10 and compare rows with column “Lib/DynPort” named “GL” (for Version 1.2), “GLEW” (for OpenGL 1.2 + Extensions) and “GL3” (for OpenGL 3.0 excluding deprecated API functions and constants).

2.2.2 Simple DirectMedia Layer

SDL (Simple DirectMedia Layer) is a portable and open-source C library providing an abstraction layer to the windowing system, including multimedia services and hardware. The library development was initiated in 1999 by Lantinga at "Loki Software", a company that was specialized in porting video games to the Linux desktop. SDL offers a coherent interface to core components such as video- and audio output, timer management, multi-threaded programming and CD-ROM drives. (See Lantinga and et al, 2012, and Pendleton, 2003 for an introduction to the C API.)

SDL also offers an abstraction to the windowing system for the creation of OpenGL graphics contexts. This makes SDL an attractive foundation layer for writing platform-portable OpenGL-based applications.

Based on `SDL`, there exist several extension libraries that provide further abstractions for more specific tasks. For example `SDL_image` offers a common interface for loading and saving of pixel-based images with support for a large list of graphics formats. `SDL_mixer` comprises music player routines for various codecs including *MP3*, *OggVorbis* and *MOD*. `SDL_ttf` enables the loading and rendering of fonts. Furthermore, `SDL_net` contributes networking facilities to the SDL framework.

The list of supported platforms is very large. Besides ports to all major desktop operating systems and windowing systems, ports exists to game consoles and embedded platforms. Today a large number of software packages exists that use `SDL` as a foundation layer for platform portability. Currently (as of September 2012), the user database of the SDL website (Lantinga and et al, 2012) lists 702 video games, 119 technical demonstrations, 182 applications and bindings to 27 programming languages. Bindings for the R language are not mentioned; in this thesis we discuss the framework that provides R bindings to several libraries of the SDL family, contributed with the `rdyncall` R package.

2.3 Motivation

In this section we make a brief digression to real-time graphics visualization in computational statistics, with focus on R. We emphasize the importance of interoperability between programming languages and software technologies, such as R and OpenGL, using the example of `rgl`, a contribution of a 3D real-time graphics visualization device system to R. We review the current states of `rgl` and `OpenGL` and discuss several design issues of the software architecture which is the ground work for the motivation of this thesis.

2.3.1 Graphics Plotting versus Real-time Graphics Rendering

One of R's strength is its graphics capabilities for plotting data and displaying results. While the graphics system of R offers a large range of output devices and formats, it is designed for high quality output of static images; it is not suitable for real-time graphics. In general, if data plots have more than two dimensions, it is challenging to produce a meaningful image without losing important details. 3D visualization techniques can be very helpful here. The `persp` function of R enables the drawing of a 3D surface of a height matrix but the underlying problem remains: since a 3D graph is projected onto a 2D image, the viewpoint of the projection is of importance. Significant regions at the *back* of a 3D plot might be obscured by data at the *front*. By using different colours and transparencies one can improve the ability to see through the front. However, real-time graphics visualization, coupled with interactive navigation, is much more valuable here, because it enables users to change the viewpoint and to get an instant update of the display; the user can interactively *explore* the data.

However, the R graphics system was designed for plotting images; it is too slow for real-time graphics output that requires rapid graphics clear and drawing performance.

In the past, real-time computer graphics was a reserved domain for owners of dedicated and expensive computer hardware available to professional engineers in product planning and construction, medical visualization, and for film and TV broadcasting companies. A second branch that influenced the development of computer graphics software techniques and hardware can be traced back to the early years of home computers and its computer game industry.

When the first commercial video games for home computers were sold in the early 1980s, a black market for 'cracked' versions of computer games evolved very rapidly. A 'crack' of a game was produced by skilled computer enthusiasts who reengineered the game code to remove the copy protection and finally to create a freely copyable version. Small programs, named "intros", were added to a cracked version, and got loaded before the actual game. Intros were used to advertise the so called 'cracker group' responsible for the crack. Although intro programs had strong size limitations, programmers constantly improved their skills to develop visually rich *real-time* graphics effects while playing computer-generated music. As more people became attracted to "intro" and "demo"[2] coding, a community, known as the "Demoscene" (Tasajrvi, 2004), diverted from its criminal origin. Game developers and demo coders started to push computer hardware to its limit in order to produce astonishing real-time graphics effects, but often with a limited scope of reuse.

At the end of the last century, hardware-accelerated 3D graphics chips had become very inexpensive due to the tremendous success in 3D graphics card sales volume, and the appetite of video game consumers for improved 3D graphics capabilities in terms of output resolution, quality and refresh rate. In the last decade the graphics hardware has evolved from circuit boards with a fixed function to freely programmable highly parallel vector-based GPU (Graphics Processing Unit). Today GPGPU (General-Purpose Graphics Processing Unit) computing is a field of applied computer science in research domains, such as molecular modeling in chemistry (Stone et al., 2007), Black-Scholes Options Pricing in finance (Kolb and Pharr, 2006) or Markov chain Monte Carlo methods in statistics (da Silva, 2011). The OpenGL library is a foundation of this development, now superseeded

[2]Demos are similar to intros but without size limitations and may contain several parts similar to a music video clip - but usually still employing real-time graphics effects.

by successor libraries designed for GPGPU programming, such as CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language).

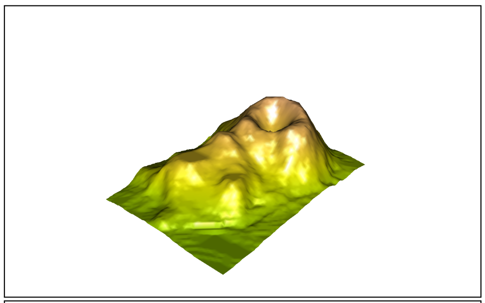
2.3.2 3D Visualization Device System for R

The `rgl` package contributes a 3D visualization device system for R using OpenGL for real-time graphics rendering.

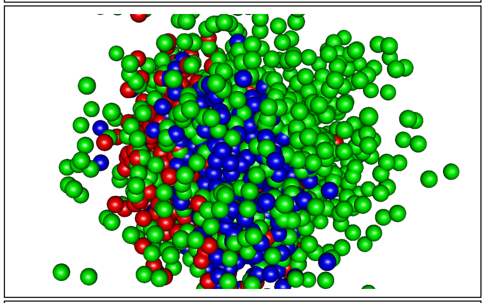
`rgl` offers a set of building block functions to define 3D shape geometry and apperance for *authoring* a 3D scene. 3D visualizations are created by passing data as parameters for geometry and appearance properties.

The design of the programming interface of `rgl` is close to that of the R graphics system and is optimized for displaying large amount of data in 3D space. The viewer window projects the data in 3D with automatic centering of the focus. The user can use the pointing device to rotate, zoom or to change the perspective distortion (up to an orthogonal projection).

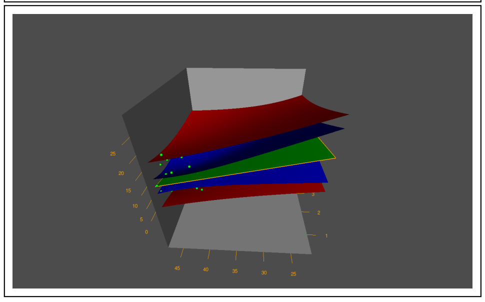
Some examples with code are given in Figure 2.1.



```
data(volcano)
x<-1:nrow(volcano)
y<-1:ncol(volcano)
z<-volcano*0.3
r<-range(volcano)
scaled<-volcano-r[[1]]/diff(r)
cols<-terrain.colors(256)
cmat<-cols[scaled]
surface3d(x,y,z,cmat)
```



```
n <- 1000
x1<-rnorm(n,2,3);y1<-rnorm(n,1,5);z1<-rnorm(n,-2,4)
x2<-rnorm(n,4,7);y2<-rnorm(n,2,7);z2<-rnorm(n, 2,7)
x3<-rnorm(n,3,5);y3<-rnorm(n,3,5);z3<-rnorm(n,-2,3)
spheres3d(x1,y1,z1,color="red")
spheres3d(x2,y2,z2,color="green")
spheres3d(x3,y3,z3,color="blue")
```



```
library(hotplots)
m1<-lm(sr~pop15*pop75+pop75,data=LifeCycleSavings)
plot3d(m1)
```

Figure 2.1: Examples of visualizations of `rgl`.

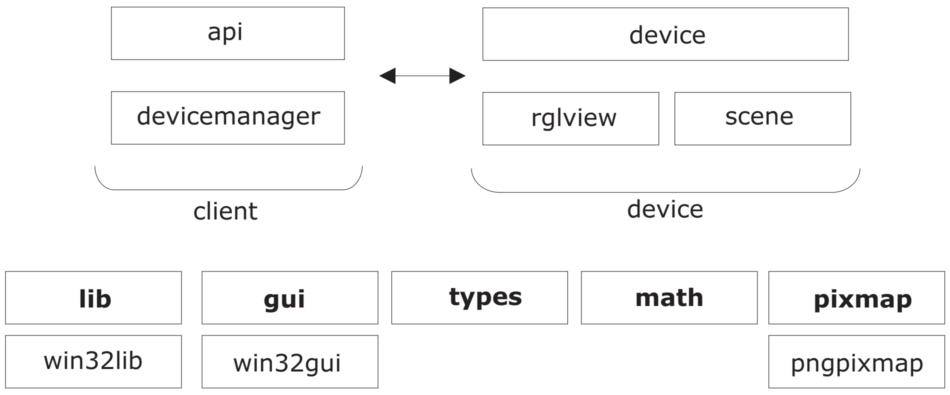


Figure 2.2: C++ software architecture of R package `rgl`.

2.3.2.1 History

The first public version of `rgl` appeared (Adler et al., 2003) in the context of a diploma thesis in Fall 2002. The first port was initially done for the Microsoft Windows operating system. The work was awarded with the John S. Chambers Software Award 2003 at the JSM (Joint Statistical Meetings) in San Francisco. At the DSC (Distributed Statistical Computing) 2003 a new version was presented (Adler and Nenadic, 2003) that included a second port to the X11 display system. Hence, the group of Unix-based platforms running Linux, BSDs and others were also supported. The package name inadvertently clashed with another package (Murdoch, 2001) that was based on a similar idea but written in Delphi for Microsoft Windows. That version was renamed to `djmrgl` and the authors joined efforts to merge e.g. `par3d` (a coherent 3D graphics parametrization interface analogous to `par` for R graphics) and to improve the code base. In a later version, a third port to native Mac OS X/Carbon was added. That step complemented ports to all three major windowing systems and can be considered to be one of the most important parts of the developments. In the meantime, a Mac OS X/Cocoa port has been added to support Mac OS X 64-bit platforms. CRAN Task Views, a topic-driven user guide to R packages on the web, classified `rgl` as a *core* package for *graphics*; About 60 packages on CRAN depend on `rgl` functionality and another 54 packages suggest the use of `rgl` (April 2012). `rgl` is mentioned as one of "the packages with the highest numbers of reverse strong dependencies"(Hornik, 2012, pg. 63).

2.3.2.2 Software Architecture

The most challenging part of the implementation of `rgl` was the integration of the platform-specific windowing system and user interface with ports to Microsoft Windows, Apple Mac OS X and Unix-based X11 desktop systems. The C++ software architecture is given in Figure 2.2 and is briefly outlined below.

Each box in the figure represents a certain module or class. At the lower part we see several modules that represent a foundation layer for platform-portable application development.

The modules "lib" and "gui" provide for an abstraction to the underlying operating system and windowing system, respectively; underneath the two modules the figure also depicts the *port* to the Microsoft Windows platform comprising of two modules with the common prefix "win32". Module "pngpixmap" implements support for reading/writing image data in the png image format offered as an abstract service "pixmap" for loading image textures and for saving screenshots.

`rgl` supports the simultaneous use of multiple open 3D "device" objects organized by the "device manager" that holds the currently active device, which is the target for action commands executed via the R programming interface depict as Module "api".

Module "device" represents a single 3D device that owns a window object. Module "rglview" represents the 3D rendering viewer component with interactive navigation. Module "scene" represents the heart of `rgl` because it comprises the basic building blocks for the composition of 3D scenes. We depict the C++ class hierarchy of the database in Figure 2.3. Leaf classes represent final building blocks for data visualization that are exposed to the R programming interface. E.g. `rgl.points()` or `points3d()` (for the creation of a point cloud in 3D) leads to the creation of C++ instance objects of class `PointSet` that are hooked into the database run-time model. The viewer component uses this database for the rendering of 3D graphics. If new objects are inserted into the database the viewer reflects that instantly on the screen.

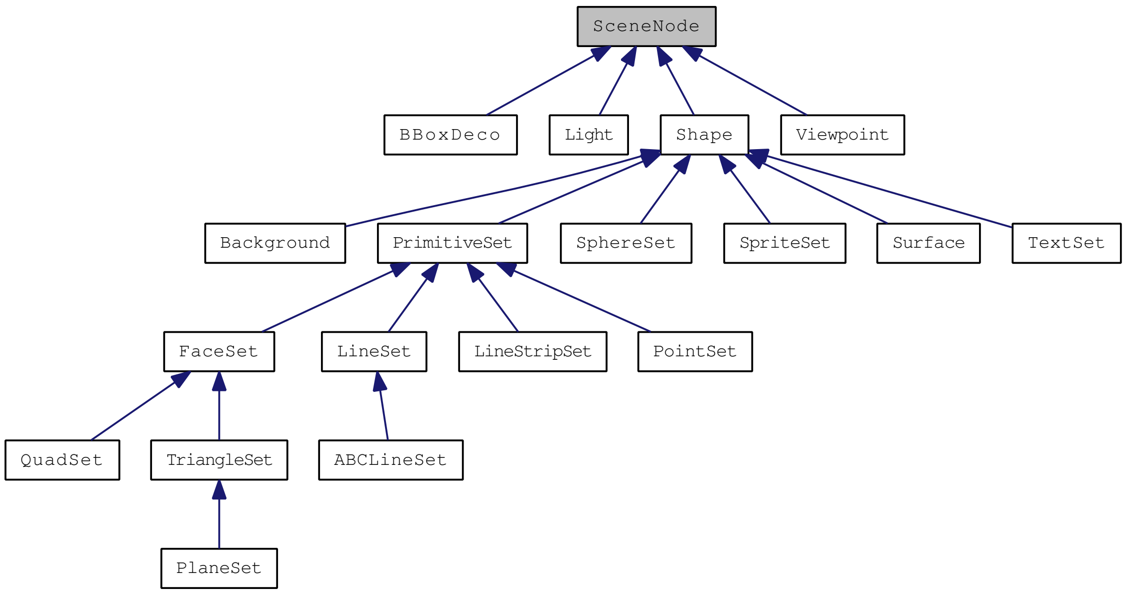


Figure 2.3: C++ class hierarchy of `rgl` scene graph database.

2.3.2.3 Drawbacks of the Architecture for Extensions

From an R user's perspective, a major drawback of this architecture for extending `rgl` with new visualization methods is given by the use of a multi-layered software design comprising of code in C++ and R. The implementation for a building block of `rgl`'s scene graph is spread across three different layers:

1. The interface function, written in R, receives shape node parameters for appearance properties and the geometry data. The former are forwarded to a C interface function for handling material data, the latter are passed to the next layer.

2. A C++ function, using an R-to-C calling convention, receives data for geometry (e.g. coordinates, sphere radius, etc..). It creates an instance of a new shape node (implemented in the next layer) that is then inserted into the database of the currently active device.

3. A shape node, written in C++, prepares geometry data for rendering (e.g. generation of normal and texture coordinates for lighting computation and texture mapping, respectively) and implements the update method for graphics rendering.

In Section 2.1.1 we outlined advantages of *scripting* as a technique for a *rapid* application development. A hybrid implementation for the high-level parts in a system-level and scripting language significantly slows down development of new visualization methods. But the choice in favor of a hybrid software design in C/C++ and R for extensions is often inevitable if the necessary interfaces are not available for R. Since R interfaces to important components of `rgl` were missing, such as OpenGL and an portable abstraction to the windowing system, a *foreign* compiled language was needed here.
Furthermore, `rgl` is based on OpenGL Version 1.2, and, in the meantime, the graphics processing pipeline model and API of OpenGL have significantly changed from a fixed-function model to a freely programmable one for major stages of the pipeline. The new API offers considerable scope for advanced applications of real-time graphics visualization and high-performance computing. Although it is possible to upgrade `rgl` to use newer versions of OpenGL, we would have to introduce additional R interfaces and abstraction layers to take advantage of the new programming model of OpenGL. However, the C API of OpenGL already represents *the* interface for accessing graphics hardware resources in a platform-portable manner. So it makes sense to develope an approach that connects this interface directly to R.
If R bindings existed to all required low-level components of `rgl`, then the application could be written entirely in R and the distribution of the package would become significantly easier as compilation is eliminated.
If the creation of R bindings can be automated by using the official OpenGL C header file definitions, new versions of OpenGL could be incorporated promptly. Aside from `rgl` other applications for using OpenGL in the context of statistical computation with R are also becoming feasible, as R users do not have to figure out a complex C/C++-based software architecture in advance to start exploring new methods.

2.3.3 Suggested Architecture for Language Extensions

We propose an alternative software architecture for those language extensions that require significant support from external C libraries for their implementation. Instead of writing extension code in C/C++ to gain access to the C API of libraries, we suggest providing for language bindings to external C libraries in a first step and then to *script* the application or extension.

In that we shift the implementation from a foreign compiled language to the scripting language, we provide for a *rapid* application development model as discussed in Section 2.1.1. In the case of `rgl` this would reduce the number of implementation layers and the software design becomes more transparent to scripting users. For example, R users can extend `rgl` with advanced visualization techniques by using the core OpenGL API without the need to write and integrate C++ code across multiple layers of implementation.

In general, if C libraries can be used within scripting languages as language extensions, the pool of available language facilities instantly grows with high-quality components. Mature C libraries, such as `OpenGL` and `SDL`, provide an elaborated interface that has been developed over many years.

One prequisite of this software architecture design is the existence of a software technology that facilitates the bindings between the scripting language and C library. If possible, the bindings should be established in a dynamic manner so that compilation is not required which significantly eases the building and distribution of dependent application code.

In this thesis we propose a *Dynamic Bindings Model* for making arbitrary C APIs of external C libraries available as an extension to the scripting interpreter; our proposal emphasises the R language as a first implementation as well as platform-portability across processor architectures.

Chapter 3

Dynamic Bindings Model

In this chapter we discuss a middleware architecture for scripting languages and shared C libraries that provides scripting access to C APIs across platforms. In contrast to compiled language bindings, we aim to overcome compilation through the use of dynamic methods for run-time interoperability with precompiled shared library code, thus the name Dynamic Bindings model.

We begin with brief accounts of the background concepts, such as shared libraries, language bindings, interoperability and FFI, and then proceed to a description of the proposed model.

3.1 C Libraries

A basic idea of this thesis is to regard a C library as a *self-contained* component, and to elaborate techniques for C interface access from scripting languages. Therefore we give a brief overview of types of compiled C libraries and their distribution as software packages.

3.1.1 Binary Formats

In general, we distinguish two binary format types of C libraries:

- *Static libraries* represent the most basic type of a compiled library. After compilation of library sources the corresponding object files are collected in a single archive file. Static libraries are used for development of subsequent software. In simplified terms, static libraries are linked with user software by copying the archive's object files into the target executable or library file.
- *Shared libraries* comprise a rather complex binary form that resembles that of an executable image as shared libraries are *loaded* to a running process at load time or at run time:
  - *Load-time* linking: Shared libraries can be used for development of subsequent software, such as in the case of a static library. But instead of copying the library code into a target

executable, application code is linked symbolically. On loading an application, its executable file and referenced shared libraries are loaded into memory before the application's process *starts*.

  – *Run-time* linking: Shared libraries can also be loaded dynamically, on request, as self-contained binary components to a *running* process.

Shared libraries represent the most flexible binary format of a C library. After compilation as a shared library, the library can be used as a static dependency in other applications (load-time linkage) or also dynamically as a component. Dynamic run-time linkage is fundamental for plug-in architectures where the system is configured via loading components from a (possibly large) repository. For example, programming environments, such as R, use run-time linkage for loading extension packages that comprise compiled C/Fortran code.

The *Dynamic Linker*, a component of the operating system, is responsible for loading and load-time/run-time linking of shared libraries and executables. In order to reduce physical memory, it uses the virtual memory management system to load the read-only regions of shared library files into physical memory once and maps this memory to several user processes. Hence a significant amount of physical memory is *shared* if several application processes use the same shared library.

Since we consider dynamic loading of shared C libraries across platforms, we discuss platform-specific details in Section 4.2.3. In Section 5.5 we present a C library with a portable API for dynamic run-time loading of shared libraries including ports to major operating systems. Detailed information about binary formats and loading/linking methods is given in Levine (2000).

3.1.2 Software Distribution

Libraries are distributed as software packages in three different forms, depending on the chosen binary format, installation and deployment policy of the target platform:

- A *source package* comprises the source code, header files and miscellaneous files for building, deployment, documentation preparation and installation.
- A *run-time package* comprises the prebuilt binary file for a particular *operating system/processor architecture* combination.
- A *development package* comprises common platform-neutral files for development and compilation, such as C/C++ header files and documentation.

Package management systems, such as used in open-source Linux and BSD systems, as well as those provided by third-party system extensions, such as the *MacPorts* project (MacPorts Project, 2002) for Mac OS X and the *OpenCSW* project (OpenCSW project, 2002) for Solaris, are very convenient for distribution of source, run-time and development packages for libraries.

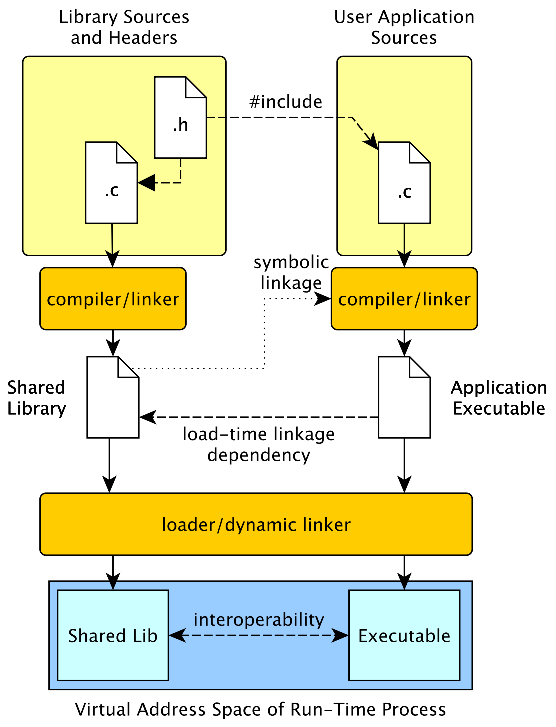


Figure 3.1: Dynamic load-time linkage of shared libaries. The build process (top) and the run-time environment (bottom).

If a shared library is used by a large number of applications, shared libraries and package managers become very efficient to the overall system. On current platforms, there are usually very *few* libraries that provide significant services to *virtually all* applications available. Due to the linkage model of shared libraries, package systems can arrange operating systems as a tree of dependent binary packages of applications and libraries. On the broad scale, the overall compilation time for building a comprehensive software repository is significantly reduced. Furthermore, software updates are available more promptly. Finally, end-users experience a rapid installation procedure since binary packages contain only the bare minimum of binary data. Dependencies to shared library packages are resolved at installation time and need to be installed only once. (For an overview of different installation methods across platforms, see Table 4.12 which uses the example of the `SDL` library.)

3.1.3 Binary Interface

Since libraries and dependent applications are compiled separately, possibly by different compilers on the same platform, the compiler plays a significant role in the realization of the binary interface.

Figure 3.1 illustrates the build processes for a shared library and a dependent user application that links with a shared library dynamically at load time. The shared library is linked symbolically to the executable. At load time of the executable, the loader and the dynamic linker (both are usually combined as a single component of the operating system) are responsible for setting up the virtual address space and mapping the executable including all dependent shared libraries into the virtual address space of the process.

Since there are no other run-time services involved except the loader and dynamic linker, run-time interoperability between application and library code needs to be facilitated in advance by compilers at compile time; they have to conform to a platform-specific implementation standard for the C language to make the run-time linkage and interoperability work. If for some reason two different compilers, one for the library and the other for the application, did not share a common C compilation standard and calling convention, then this would break compatibility, in the worst case, without notice at compile time.

The conformance of components at low level is given by the ABI, published for a specific platform. It contains machine-specific implementation details of C. Thus in effect the ABI and a conforming compiler implement the *interoperability* at compile time. (see Section 5.2 for a detailed discussion on the ABI.)

Shared C libraries can be regarded as a pure form of binary components within a system; they can be loaded as plug-ins dynamically at run time via dynamic linkers to a running process. However, a dynamic interface for interoperability with the components of their interface, such as functions and data objects, is not provided by operating systems. If we can carry out dynamic and generic methods that conform with the ABI, we can provide for *dynamic* interoperability between scripting languages and precompiled shared C libraries. Our considerations and a proposal for such a service is discussed in detail in Chapter 5.

3.2 Language Bindings

A *Bindings* extension for a particular scripting language gives scripting access to the C API of a specific external library. The scripting interface comprises proxy objects that mimick the C API via delegation of function calls and value preservation for symbolic constants. In this section we discuss common techniques for developing of language bindings extensions.

3.2.1 Compiled Language Extensions

Bindings are often implemented as *compiled* extension modules, written or auto-generated, in C. The source code comprises repetitive code sequences for wrapping C library function calls. In addition, depending on the design of the C API, methods for mapping symbolic constants, data structures and callbacks need to be taken into account. The details for passing control flow and for value conversion are carried out by *bridge* or *wrapper* C code that connects the two C interfaces, that of the interpreter and that of the library. The code strongly depends on the extension and embedding interface of the scripting language. Muhammad and Ierusalimschy (2007) give an overview and compare C interfaces

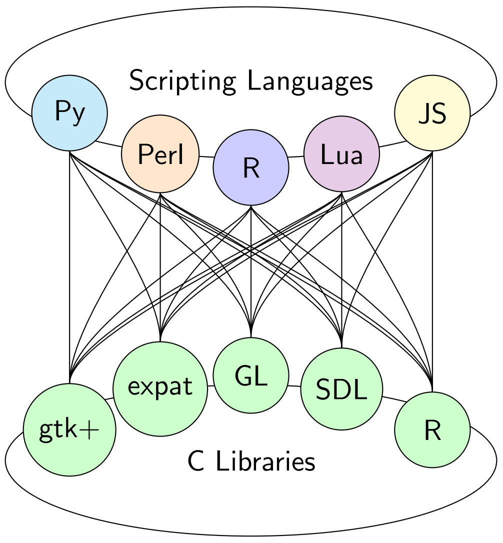


Figure 3.2: Illustration of scripting languages, C libraries and bindings on a single platform. In the compiled model each path between a language and library represents a compiled extension module.

for extension and embedding scripting languages.

The workload for making a single C library available to the set of popular languages, or for making a set of popular C libraries available to a new programming language, can be enormous. Figure 3.2 gives a graphical illustration. Note that taking account of the *platform* axis here as part of software build-and-release process for cross-platform languages multiplies the workload by the number of supported platforms.

Compilation of language bindings has an annoying side-effect. Scripting languages often suggest their own *build-and-deployment* tools for extension development, in particular when using C, for example see extension and embedding guides for R (R Development Core Team, 2012c), Perl (Jenness and Cozens, 2003), Tcl (Flynt, 2003), PHP (Golemon, 2006) or Python (van Rossum, 2012). Aside from learning the build tools, object code needs to be linked with external libraries. If a cross-platform bindings extension is required, the development task can be complicated and tedious due to platform-specific differences for linking. See Section 4.1 for an illustration of linking user code with `SDL` across platforms.

3.2.2 Handwritten Bindings

The development of language bindings requires detailed knowledge of both sides of the bridge; internals of the scripting language, as well as the C API and interface usage policy of the library need to be considered in order to achieve a comprehensive scripting interface to the C API. The development of bindings becomes a cumbersome process, in particular, if the C API comprises hundreds of components. In practice, work-arounds are common; for a given programming task, only the bare minimum of functionality is connected with the scripting language in the form of a custom bindings package. But in the long run this approach runs contrary to the objective of using scripting languages for rapid application development by gluing of 'components', such as shared C libraries with a well-defined API.

3.2.3 Automation Tools

Automation tools can significantly reduce the workload for developing a language bindings extension. Most notable is SWIG (Simplified Wrapper and Interface Generator), a software tool for automatic generation of language bindings with support for a large number of programming languages. `SWIG` (The SWIG Developers, 2010) generates language-specific extension C code according to a language neutral module description of the library. The format of the module description looks very similar to C header files. We illustrate the workflow for creating a `SWIG`-based binding of the `sqrt` C function to the Lua (Ierusalimschy et al., 1996) programming language in Listing 1.

```
%module sqrt
double sqrt(double x);
```

```
require"sqrt"
x = sqrt.sqrt(144)
-- result: x = 12
```

```
LUA_LDFLAGS =$(shell pkg-config lua --libs)
LDFLAGS     =${LUA_LDFLAGS}
all: sqrt.so
%_wrap.c: %.i
        swig -lua $<
sqrt.so: sqrt_wrap.o
        $(CC) -shared $(LDFLAGS) $^ $<
```

Listing 1: SWIG example to create a Lua binding to the `sqrt` function of the C standard `math` library. Listing in top-left corner gives the SWIG module specification, listing in bottom-left corner gives Lua user test code and the listing to the right gives a sample `Makefile` to trigger the SWIG code generator and then the C compiler to produce a Lua shared library module.

`SWIG` accepts interface files with the file extension `*.i` to be written manually, as stated in the manual: "Although SWIG can parse many header files, it is more common to write a special '.i' file defining the interface to a package. [...] It is rarely necessary to access every single function in a large package. Many C functions might have little or no use in a scripted environment."(SWIG Developers, 2009, Section 5.7.3) This argumentation appears to be pragmatic for the development of bindings for custom application software. However, in the context of scripting of applications by gluing *general-purpose* C library components, it becomes important that the complete C interface is made available to the scripting language.

Further the manual states, that "SWIG can't parse certain definitions that appear in header files. Having a separate file allows you to eliminate or work around these problems."(SWIG Developers, 2009, Section 5.7.3)

`SWIG` can be considered as a helpful development tool for creating compiled bindings but with limitations; it still requires further manual assistance for writing the interface file, as well as for building and linkage across platforms. In contrast, we aim to exchange the wrapper compilation layer with dynamic methods for interoperability between scripting language and C libraries, that serves as a foundation layer for a cross-platform bindings framework to shared C libraries and their API.

3.3 Interoperability

In the field of computer science the term "Interoperability" is used to characterise the ability of diverse systems and components to work together. The IEEE Glossary (Geraci et al., 1991) defines interoperability as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged."

In this thesis we consider interoperability within a single running process between the run-time environment of scripting languages and precompiled external libraries with a C API. In particular, we are interested to develop methods for interoperability that work at low level in a *dynamic* and *generic* manner, where the bindings are driven by *type information* at a meta level across native platforms. We consider two low-level forms of interoperability that are briefly outlined next.

3.3.1 Code-level Interoperability

Code-level interoperability has to take account of function calls and the orderly passing of control-flow from the interpreter's environment to the C run-time environment of the library.

Libraries often play an implicitly passive role as a collection of functions to be called in a uni-directional manner. But some libraries use "callbacks", i.e. user-defined functions are passed to library code which is called later from inside the library.

Therefore, code-level interoperability needs to be considered in both directions: from language to library (calls or call-out) and from library to language (callbacks or call-in).

3.3.2 Data-level Interoperability

Data-level interoperability deals with the synchronization of views on data objects that are shared by the scripting language and the C library. Data-level interoperability is intrinsically anchored with code-level interoperability at low level. During a call to a foreign function arguments are converted from a high-level form to a low-level representation. Conversely, return values received at low level need to be converted to newly allocated high-level language objects. This process is also named *marshalling* (from high to low level) and *unmarshalling* (from low to high level).

While parameter and return types of C functions are usually of scalar data type, which makes the conversion process between the interpreter and C manageable, the handling of scalar pointer objects represents a challenge for data-level interoperability, especially if the underlying pointee object is of a composite data type, such as user-defined `struct` and `union` data types. Many libraries provide an interface policy framework that specifies certain rules to follow (beyond calling API functions in a certain order).

A commonly used interface policy requires that data objects are allocated and initialized in user code and are passed as call-by-pointer arguments to a library's function; user code has to deal with the management of composite C structure and union data type, including allocation and destruction of memory objects and precise addressing of fields within the object. (See Section 4.5 for a comprehensive example.) The native representation of complex C data objects needs to be guaranteed when a pointer is passed as argument to a function of the library. Depending on the interface policy, the object also needs to remain at the address in memory subsequently.

A dynamic interface requires an appropriate carrier *within the scripting language* for C composite data objects with support for embedding C objects within scripting objects and referencing of C objects via pointers. By preserving the native representation and address of composite C data objects, and by providing a scripting interface for read and write access on field level, data-level interoperability is also feasible in this context.

Fisher et al. (2000) give a detailed discussion on data-level interoperability implemented for C and the Moby language and coin the term "Foreign Data Interface" for this approach. In Section 4.5 we discuss an FDI (Foreign Data Interface) for the R language.

3.4 Foreign Function Interfaces

A Foreign Function Interface (FFI) is a facility for programming languages to provide interoperability with external libraries written in a foreign language. The term was first coined in the context of Common LISP; users were given a set of functions to make function calls to foreign code of external libraries without leaving the language. A large number of extensions for various dynamic programming languages adopted this term. And even compiled languages and virtual machines, such as Java, use this term, for low-level interfaces to native C code, such as the JNI (Java Native Interface). Despite its name, FFIs often include extended functionality beyond calling *foreign* functions from external libraries.

The online work-in-progress report "Design Issues of FFIs" (Urban, 2004) gives an overview of existing FFIs and provides a detailed classification. In the reference manual of ECL, an implementation of Common Lisp, *"Two kinds of FFI"* are decribed (Garcia-Ripoll and Rosenberg, 2006, Sec. 3.2), namely "Static FFIs" and "Dynamic FFIs".

3.4.1 Static FFI

Essentially a static FFI is a language-extension interface to C; it provides functions to make calls to C. However, the support for making calls to foreign functions of external C libraries is rather limited. We discuss the limitation of static FFIs in the following section using the example of R.

3.4.2 Limitations of the Static FFI in R

The FFI of R comprises several R functions to make calls to compiled C and Fortran code; each interface function supports a different target language and calling convention:

Interface	Description
`.C()`	Foreign C function.
`.Fortran()`	Foreign Fortran function.
`.Call()`	C function with R calling convention passing each argument as an SEXP.
`.External()`	C function with R calling convention passing the call object as a single SEXP.

`.Call` and `.External` are used by package developers to call precompiled C functions that are written intentionally as an extension to R. C functions must be of a specific type in order to be compatible with R so that C code can access R arguments and return results as R objects. In general, such C code also accesses the low-level C API of R to create new R value objects.

`.C` was designed for the purpose of calling *foreign* C code from within the interpreter. It has the following functional interface:

```
.C(name, ..., NAOK = FALSE, DUP = TRUE, PACKAGE, ENCODING)
```

The first argument `name` specifies the C function to be called; it is followed by the arguments to be passed, denoted by the placeholder ‘`...`’. We notice that no additional parameter is reserved to specify the function type of the target C function; the C function type is derived from the R argument types that are passed on each foreign function call. Consequently the C argument types of the target C function need to be *compatible* with the R argument types and the conversion is driven by latter. Table 3.1 gives the mapping rules.

R Type	C Type
`integer`	`int *`
`numeric`	`double *`
`single`	`float *`
`complex`	`RComplex *`
`logical`	`int *`
`character`	`char **`
`raw`	`unsigned char*`
`list`	`SEXP *`
others	`SEXP`

Table 3.1: Static mapping of R to C argument types in R’s Static FFI `.C`.

Although the FFI works at a dynamic level without compilation, we classify the FFI as a static rather than a dynamic call interface to C. The reason is illustrated by the following simple example.

Example Suppose we wish to compute the square root of a positive number and plan to use the C function `sqrt` from the C Standard `math` library that is pre-installed on most platforms. An excerpt of the header file `math.h` that specifies the function prototype declaration is given here:

```
double sqrt(double x);
```

The function receives a single scalar argument of C type `double` denoted with `x`. Its implementation computes the square root of `x` and returns the result of C type `double`.

The `.C` function is suitable to "make calls to compiled code that has been loaded into R"(quote from the manual page of `.C` as of R Version 2.14):

```
# Load shared C library (Example on x86-32/Linux)
> libm <- dyn.load("/usr/lib/libm.so")
> addr <- libm$sqrt$address
> .C(addr, 144)               # UNDEFINED BEHAVIOUR
[[1]]
[1] 144
```

The expected result of 12 was not returned; it is also quite possible that the call will lead to an abnormal termination of the R session; also no warning or error is returned. Closer examination reveals: `.C` does not support passing of scalar objects. Instead, a pointer to that value is passed. Moreover it does not support C return values. The C function must return values via pointer arguments.

Thus, an intermediate C wrapper code is needed that serves as an adapter between R and `sqrt` to convert arguments and results as given in Listing 2. The left panel gives the C wrapper code. On the right panel of the listing an R front-end function is given. The R front-end insures that the R argument is coerced to a `numeric` value (i.e. to be passed as a C `double*` pointer to C) and the length of the argument vector is greater or equal to 1 before the call to the C wrapper code.

```
#include <math.h>
void wrap_sqrt(double* ptr)
{
  double x, result;
  x = ptr[0];
  result = sqrt(x);
  ptr[0] = result;
}
```

```
libwrap_sqrt <- dyn.load("libwrap_sqrt.dylib")
mysqrt <- function(x)
{
  x <- as.double(x)
  length(x) <- min(length(x), 1)
  .C(libwrapsqrt$wrap_sqrt$address, x )[[1]]
}
```

Listing 2: Hybrid C/R call wrapper for `sqrt`.

The code has to be compiled and the resulting object file has to be linked with external libraries to a shared library.

```
$ gcc -shared wrap_sqrt.c -lm -o libwrap_sqrt.dylib
```

Alternatively the wrapper can be prepared as part of an R package. In both cases the external wrapper library has to be linked with the C `math` library explicitly.

In most cases C library functions are not compatible with the given low-level *R-to-C* call interface of `.C`. We give a brief summary of the limitations of `.C`:

- Arguments can not be passed by scalar value, but are always passed by pointer, with support for `int*`, `float*`, `double*`, `char**` and `unsigned char*`.
- No return values are supported. Return values are passed by argument pointers.
- Non-standard C calling conventions are not supported.

In general, the FFI of R is so restrictive that no general-purpose C library code can be called *directly*. Hence, C wrapper code needs to be developed, so that it can be plugged in between R and the target C function of the library. Moreover we are forced to introduce hybrid wrappers consisting of R and C code:

- R wrapper "front-end" code converts input arguments and coerces a static type before calling the C wrapper via `.C`.
- C wrapper "adapter" code converts argument values and return values and calls the foreign C function of an external library.

Alternatively the `.External` and `.Call` FFI interfaces can be used, which gives C access to R internals, for single C wrapper implementations.

The FFI of R is a static variant that defines a strict R-specific calling convention with strong limitations. Support for handling of callbacks from C to R and foreign C data objects is also not addressed by this FFI.

We can summarize by saying that the FFI, offered by the base R package, is not an interface for *foreign* libraries; it is designed as an interface for writing extensions in C and Fortran code.

3.4.3 Dynamic FFI

Dynamic FFIs overcome the limitations of static FFIs in that a larger number of function types are supported without the need of compiled wrapper code. In contrast to the static FFI, the call interface function includes a parameter that gives the target C function type. Input arguments from the interpreter are converted according to the sequence of parameter types of the C function type; this ensures that the subsequent call to compiled code is *type-safe*. Notice that, whereas type-safety is supposed to be achieved by the C wrapper for static FFIs, it is largely unnecessary when using dynamic FFIs.

Language	Variant	Extension	Implementation
Forth	Factor	built-in	custom
Haskell	GHC	built-in	`libffi`
Java	JavaVM	`JNA`	`libffi`
		`BridJ`	`dyncall`
	`gcj`	built-in	`libffi`
	`OpenJDK`	built-in	`libffi`
JavaScript	WebKit	`JSCocoa`	`libffi-iphone`
	Mozilla	`js-ctypes`	`libffi`
	NodeJS	`node-ffi`	`libffi`
	jsh	built-in	`dyncall`
Lua	PUC-Rio	`alien`	`libffi`
		`luadyncall`	`dyncall`
	LuaJIT	`ffi`	LuaJIT/`DynASM`
ML	NJ	`ML-NLFFIGEN`	custom
OCaml	OCaml	`ocamlffi`	custom
Pawn	Pawn	built-in	`dyncall`
Perl	Perl 5	`FFI`	`ffcall`
	NQP	built-in	`dyncall`
	Rakudo Perl 6	`zavolaj`	`dyncall`
Python	CPython	`ctypes`	`libffi`
		`PyObjC`	`libffi`
R	R	`rdyncall`	`dyncall`
		`Rffi`	`libffi`
Rebol	Rebol3	`dyncall`	`dyncall`
Ruby	Ruby MRI	`ruby-ffi`	`libffi`
		`Typelib`	`dyncall`
		`fiddle`	`libffi`
		`RubyCocoa`	`libffi`
Scheme	Guile	built-in	`libffi`
	Racket	built-in	`libffi`
	PLT Scheme	built-in	`libffi`
Common Lisp	many	`cffi`	`CFFI-SYS`, `libffi`

Table 3.2: Overview of Dynamic FFIs.

Dynamic FFIs exist for a large number of languages, whether as built-in services or as extension libraries. Table 3.2 gives an overview of various dynamic FFIs. Many FFIs exist for Common Lisp, possibly due to the historical roots in that language. We introduce a dynamic FFI for the R language, named `rdyncall`, in Chapter 4.

3.4.4 Generic Dynamic FFI

Whereas static FFIs can be portably implemented in C (see Section 5.4.1 for a discussion of `.C`'s implementation), dynamic FFIs require an abstraction for function calls at machine level. The latter requires code to be written in assembly language, or incorporation of a machine-code generator. Given the complexity of developing a portable solution for the implementation of a dynamic FFI, it is not surprising that most dynamic FFIs do not include such an implementation themselves, but make use of an external library that offers this abstraction. Urban (2004) introduces a separate category, named *Language Independent Generic Dynamic FFI.* In the paper numerous FFI extensions for languages are given, but for their implementation only two C libraries with selective ports to specific processor architectures are given, namely `ffcall` (Haible, 2004) and `libffi` (Green and et al, 2011), as well as assembly language in general and "Minotaur", a Forth-based glue to C. Since 2004 (when that document was last updated), we can further extend this list with `C/Invoke` (Weisser, 2007) and `dyncall`.

Most dynamic FFIs regard the low-level implementation of dynamic FFIs as a black box and often details to be found only in assembly language sources. Given the fact that, for years now, the generic dynamic FFI layer plays the most significant role within any dynamic FFI and beyond, the shortage of comprehensive information on the design and implementation of dynamic interoperability techniques across processor architectures seems to be paradox. For example, the GCC (GNU Compiler Collection) uses `libffi` for handling calls between interpreted and natively compiled Java. The importance of these low-level packages is also given in Table 3.2 by column 'Implementation'; in summary (frequency is given parentheses) from our list of dynamic FFIs (32) either a custom implementation (4) or one of three Generic Dynamic FFI C libraries `libffi`(18) , `dyncall`(9) and `ffcall`(1) is used. We discuss the design and implementation of `dyncall` in Chapter 5.

3.5 Model Description

In this section we describe a model of a middleware architecture for scripting languages and shared C libraries to provide seamless scripting access to the C API across platforms without the use of compilation.

First we give an overview of the model and an outline of the architecture followed by a discussion on its characteristics for development of language bindings by comparision with the compilation-based model.

3.5.1 Overview

In general, C libraries are offered as binary components in the form of a shared library. They can be installed via package management systems on a broad range of platforms within seconds. Shared libraries represent the purest form of a binary component; they can be loaded dynamically as plug-ins and offer a static binary interface. The dynamic linker of the operating system provides an interface for loading shared libraries at run time, as well as for resolving symbolic names to loaded memory addresses of global function and data objects.

Further, a dynamic FFI provides the methods for run-time interoperability with foreign code and data, such as making calls to compiled and loaded C functions, but requires type information as a parameter. Thus, a wrapper layer is still required to inject type information. Scripting languages are ideal for automation tasks, including the creation of wrappers. The wrapper comprises proxies, one for each component of a C API, such as scripting functions for C functions to assure type-safe invocation of the foreign function call.

However, the essential information for using a C API, such as a functions symbolic name and type, is not contained in the binary shared library files but in C header files. A parser could be utilized to extract the required information from C header files. But this would demand that development files, including C API and system header files, are preinstalled on a target run-time platform. Furthermore, the inclusion of a C language parser would significantly increase the total memory footprint for the run-time middleware component.

As an alternative, we suggest to utilize an off-line preprocessing phase in which C header files are translated to a simple text-based format. Text formats can be parsed via standard text scanner routines, in the scripting language for text-based input to a dynamic FFI, as well as in C for processing text for operating a generic dynamic FFI. So in addition to the shared libary, the format encodes the corresponding type information, symbolic names and constant values of a particular C API as a cross-platform type library for C APIs.

3.5.2 Architecture

We now consider the implementation of the model for a specific scripting language by developing a single language extension, in C. The extension functions as a hub for the creation of dynamic bindings to arbitrary C APIs; the linkage between the scripting language and a pre-installed shared C library is created dynamically at run time.

The extension provides a function for loading of dynamic bindings. The name of the bindings is passed by a cross-platform name of the C API. On calling the function, the creation of a Dynamic Binding to the C API is triggered. The subsequent automation is driven by a platform-neutral description file that provides information about the components of the C API, including hints of the name for loading the platform-specific shared C library on different platforms. Description files are stored in a repository. Support for new C libraries is realised by extension of the repository.

An extendable repository of *portable* bindings information to C APIs is provided with the extension.

At first, the shared library is searched, based on hints of names. If the shared library is found, it is opened via the dynamic linker. Then, for each component of the C API the following proxy language objects are created:

- C API Functions are wrapped by a dynamically created scripting function which consists of a call object to the dynamic FFI with two parameters, namely code address and function type. Both are derived by the description file (for the symbolic name and type) and subsequent resolving the symbolic name to the loaded address.

- C API Symbolic constants are mapped as corresponding named constant value objects that preserve the constant value. Both, name and value, are given by the description file.
- C API Types are mapped as helper objects in order to support function pointer callbacks, data pointers to objects of composite data types, memory allocation of data objects and their manipulation. Type helper objects are initialized by type information provided by the description file.

We now consider the architecture of the model in order to reduce complexity of its implementation and to emphasize its portability across multiple languages and hardware platforms.

The *Dynamic Bindings* model comprises three layers of abstraction:

1. For each library, we encode the C API in a language- and platform-portable format, named *DynPort*. The format plays a central role as it is used as cross-platform text-based interface to the Dynamic FFI. We also give the implementation of a parser tool, named *DynPort* tool, that translates C header files to *DynPort* files. In Section 3.7 and Section 3.8 we discuss the *DynPort format* and the *DynPort tool*, respectively.
2. For each scripting language, a language extension needs to be developed. First, we need a Dynamic FFI as a foundation. Based on this FFI, we implement the automation process that parses DynPort files and creates corresponding proxy objects. We discuss this layer in Chapter 4 using the example of an implementation for the R programming language, named `rdyncall`.
3. For each platform, we need an implementation of the dynamic FFI at machine level or the use a Generic Dynamic FFI with a portable C interface. For the latter, we give a detailed discussion in Chapter 5 using the example of an implementation, named `dyncall`, comprising three C libraries that provide a portable C abstraction layer for dynamic linkers, function calls and callback handling, respectively. We give an overview of ABIs on five processor-architecture families, followed by a detailed discussion using the example of several platform-port implementations in C and assembly language of dyncall.

3.5.3 Comparison of Bindings Models

The *Dynamic Bindings* model is illustrated in Figure 3.3 which compares the different technical development workflows and run-time environment configurations of the “compiled” (left) and the “dynamic” (right) approaches.

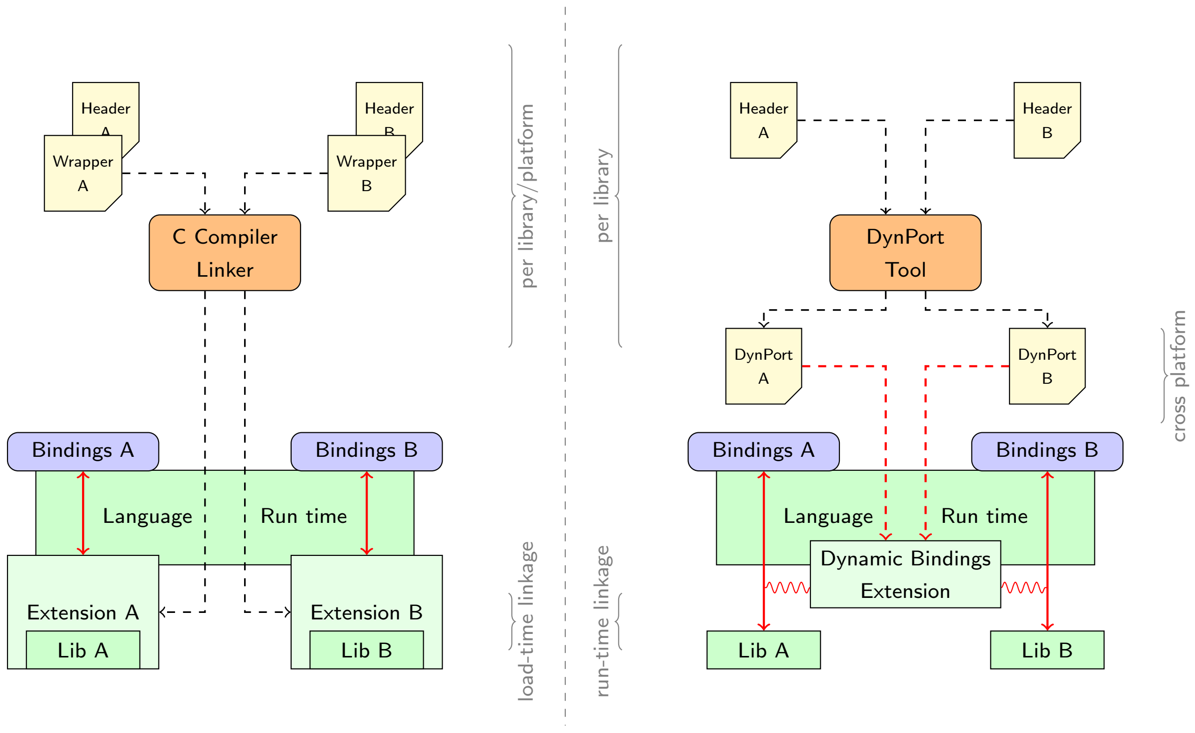


Figure 3.3: Comparison of the compiled (*left*) and dynamic bindings (*right*). See Section 3.5.2 for details.

In each case, two C libraries “A” and “B” are made available as bindings to a language run-time interpreter and environment, depicted in the lower part of the figure. The upper part gives the development tasks for creating language bindings.

The left-hand panel of Figure 3.3 gives the workflow and run-time environment for “compiled” bindings. For each C library a language extension is implemented in a compiled language (C/C++). The sources are depicted as document icons and labeled “Wrapper A” and “Wrapper B”. The term “Wrapper” is commonly used in the context of language bindings. Wrappers encapsulate existing functionality and *wrap* around a foreign interface. The wrapper code does include (via `#include` of the C preprocessor) the C API header files of the library as well as the header files of low-level extension development of the scripting language. After the modules are compiled, packaged and installed as standard language extension packages, Bindings “A” and “B” can be loaded on demand by users via language-specific standard extension loading mechanisms (e.g. loading compiled R Bindings to “A” via package “ExtA”: `library(ExtA)`).

The right-hand panel gives the workflow and run-time environment of our proposed “dynamic” bindings approach. Instead of a compilation phase, the C API header files are translated by the *DynPort* tool into a simplified language-neutral and platform-portable *DynPort* format that provides the information for automation of dynamic interfaces. After the *DynPort* files are created, and the shared libraries have been installed to the system (e.g via package managers), then as above, Bindings “A” and “B” can be loaded on demand as well but via a specific loader interface (e.g. loading dynamic R

Architecture	Year	Bits	`char`	`short`	`int`	`long`	`void*`
Unisys 1100	1962	36	9	18	36	36	72
PDP 11	1970	16	8	16	16	32	16
VAX/11	1977	32	8	16	32	32	32
Motorola 68000	1979	16/32	8	8/16	16/32	32	32
Harris H800	1982	48	8	24	24	48	24
Cray-2	1985	64	8	64(32)	64(32)	64	64(24)
Intel 80386	1985	32	8	8/16	16/32	32	16/32/48

Table 3.3: Data type bit-sizes of C types across architectures of the past.

bindings to "A" via DynPort "ExtA" and `rdyncall`: `dynport(ExtA)`).

We now compare the two bindings methods with regard to their implementation, also across multiple languages and platforms. In the "compiled" model sources for wrapper modules have to be written for each *language-and-library* combination. While the wrapper modules also need to be compiled for each *language-library-platform* combination, the *DynPort* file is generated *once per library*. The *Dynamic Bindings* extension has to be implemented *once per language*. Doing this for several languages makes one *DynPort* file applicable across those languages. The implementation of the model is based on a *Dynamic FFI* which needs to be implemented *once per language*; it is based on a *Generic Dynamic FFI*, such as `dyncall`, that has to be implemented *once per ABI platform* but can then be used for a wider spectrum. (See Table 5.11 for an overview of ABI/platform ports of the `dyncall` library and Table 3.2 gives an overview of dynamic FFIs and their implementation base.)

A key point of our model is the use of a language-neutral format that is used for shared library loading, resolving of addresses and interoperability. For the implementation we require the existence of a dynamic linker and a powerful dynamic FFI that can be driven by given information from *DynPort* files.

3.6 C Type System

Types classifies the objects and expressions of a programming language. In simplified terms, the data type of a C object gives information about the encoding and representation of data in memory, while the function type of a C function gives information about the data type of the return value and the parameters. This information is essential for dynamic FFIs in order to make foreign function calls, or to handle callbacks and foreign data objects. Therefore, we give an overview of the components of the C type system, for which we develop a text-based encoding scheme for C APIs in Section 3.7.

3.6.1 Overview

C was designed as a platform-independent language with emphasis on low-level programming and plaform-portability software development. Although both objectives seem to be rather contrary, C was successfully used for the development of cross-platform solutions as well as bit-level processing.

A key decision within the C standard was to omit implementation details, such as the representation of data types and the details of the implementation of function calls in machine code, to a large degree. The decision for the assignment between abstract C types and concrete machine-level data types is ascribed to vendors of hardware platforms, compilers and operating-systems. Table 3.3 gives an overview of the different data type representations (in bit sizes) of abstract integer-based C data types for a range of outdated hardware architectures. From a programmer's point of view, it may seem as a dilemma; the size of a data type, such as `int`, needs to be obtained from the builtin C function `sizeof(X)` at compile time, or, researched from ABI and compiler specifications. Fortunately, the mapping of data types has become more uniform across current platforms. For example, the size of an `int` is usually 32 bits on major architectures (but there are exceptions; see Section 5.3.8 for a detailed discussion on 64-bit systems and different mapping schemes).

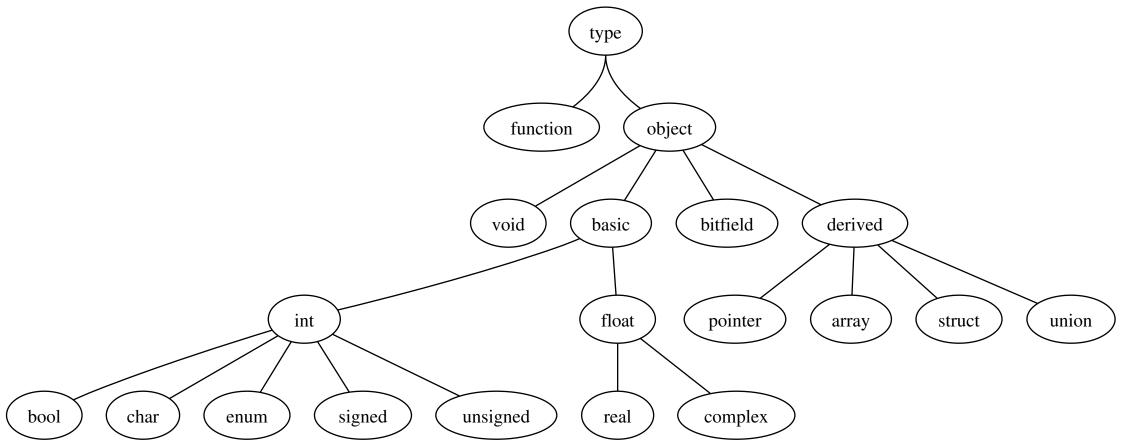


Figure 3.4: Overview of the C type system.

We now give a brief overview of the components of the C type system. Figure 3.4 gives a structured illustration of the C type system using a hierarchy of categories. At the highest level, C types can be divided into *function* and *object* types.

3.6.2 Object types

Object types describe the encoding format and storage properties of C data objects i.e. local and global variables but also arguments and results of a function call. From early on, the C standard reserved only a small number of keywords. A subset of keywords were used to refer to built-in *abstract* basic C data types.

A major milestone of the *C standard specification* was finished in 1989, often refered to as *ANSI C 89* or *C89*. It defines "only a few basic data types in C:

`char`	a single byte, capable of holding one character in the local character set.
`int`	an integer, typically reflecting the natural size of integers on the host machine.
`float`	single-precision floating point.
`double`	double-precision floating point."(Kernighan and Ritchie, 1988, Ch. 2,Pg. 36)

This classification is rather abstract and limited; it does not cover the full range of available machine-level data types, such as integral data types of different bit sizes and numeric value formats. C reserves a few more keywords in order to refine *size* and *value encoding* characteristics of basic data types. Besides the standard integer data type `int` there exist *shorter* and *longer* versions that are refered via qualifier keywords `short`, `long` and `long long`. While the C standard does not define the bit sizes for the five different `int`-based data types `char`, `short`, `int`, `long` and `long long`, a strict relation between their implementation size is implied:

`sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)` $\leq$ `sizeof(long long)`

Longer and shorter versions may refer to different sizes of its machine-level implementation type but this is not mandatory; it depends on the binary interface standards that are defined by the designers of a platform, and that are to be implemented by compiler vendors. Once a standard platform exists, one can assume that the mapping between C data types and machine-level data types with a specific size is fixed; otherwise, interoperability between applications and libraries is broken. `char` and `int` data types can be further divided into signed (the default) and unsigned value types. Unsigned data types are specified via the type qualifier `unsigned`. Signed integers may have different encoding scheme (as defined in the standard) but the two's complement representation is usually used as an implementation at hardware level across architectures.

Objects of the `char` data type are "large enough to store any member of the basic execution character set"(ISO, 2011, Section 6.2.5) i.e. `char` represents the smallest integer with a size of a single *byte* and, since hardware platforms converged at byte-level over the decades, we can assume it is of 8-bit size. No particular value encoding format is implied for a `char`; it can be implemented as `signed char` or `unsigned char`; this is decided by the ABI and/or compiler.

Usually, `float` and `double` are represented as standard floating-point data types defined by the IEEE 754 standard for single-precision and double-precision floating-point numbers; we also can assume they have a size of 32 and 64 bits, respectively. C defines a third standard floating-point data type, named `long double`; this data type is mainly used in applications that need very large precision. The implementation of this floating-point type is very different on several platforms.

Enumeration types represent an unsigned integer-based data type. In general, `enum` types are used in a declaration comprising a set of named constant unsigned integer values, as an alternative to traditional constant value definitions via C preprocessor macro definitions. In practice, the `enum` types are implemented as an `unsigned int` data types. Bit-field types appear only as members of structure types. They use a basic integer type as a carrier and specify a subset of reserved bits from that type. When bit field types of the same carrier type occur in sequence within a structure, the memory can be shared among bit-fields.

The `void` type is a unit type that refers to a non-data type. Usually it is used to denote the absence of a return value for functions or an incomplete pointer type (`void*`).

While the first two ratification processes, namely ANSI C 89 and ISO C 90, were equivalent technically, ISO C 99 expanded the set of standard basic types by a boolean type `_Bool` , a `long long` data type for very large integers and a `_Complex` floating-point qualifier for complex values. The most recent revision of the C standard, namely *C11 (ISO/IEC 9899:2011)*, mentions the data types `__int128` and `double double` as possible extensions for supporting larger integer- and floating-point data types in future.

Compilers often provide extensions to the C standard and its type system; this includes instrinsic functions and data types that make use of advanced features of the processor architecture. Other data types and extensions are often found in specifications of processor-architecture ABIs. In Table 3.4 we give an overview of standard basic data types and platform-specific extensions.

In general, cross-platform C APIs tend to use a portable subset of possible C data types. So if the support is limited to a subset of C data types, there is still a good chance that most general-purpose C libraries can be adopted.

3.6.3 Composite data types

`struct` data types are used to define complex data types or records comprising of members that are sequencially arranged. Each member field consists of an object type and a field name; the latter needs to be unique within this structure. The exact layout of this information depends on the size and the alignment properties of its members.

A `union` data type is a type unification of a sequence of field types. Fields of a `union` data object share the same block of memory. Fields access the shared memory in terms of the corresponding field type. The total size and alignment of a union is as large as the maximum size and alignment of its members, respectively.

3.6.4 Pointer types

Pointers are a very powerful component of the C language; as lightweight objects, comprising a reference on another object, they can be exchanged rapidly between functions and data structures, so that the referenced object need not to be duplicated for data exchange. Pointer types are implemented as unsigned integer data types that encode a memory address. Additional type information about a pointer's base type is available but not required for handling the raw data type of a pointer.

But, in the context of a dynamic FFI or a C compiler, the base type of a pointer becomes very

C Type	Standard or platform-specific details
incomplete types	
`void`	C89
Boolean	
`_Bool`	C99
character and standard integer types	
`char`	C89
`short`	C89
`int`	C89
`long`	C89
`enum`	C89
`long long`	C99
Pointer	
real floating types	
`float`	C89
`double`	C89
`long double`	C89
complex types	
`float _Complex`	C99
`double _Complex`	C99
`long double _Complex`	C99
extended integer types (implementation-specific)	
`signed __int128`	`x86-64`, `ppc64` (`GCC` 4.7)
`unsigned __int128`	`x86-64`, `ppc64` (`GCC` 4.7)
`_Decimal32`	32bit BID (IEEE-754R)
`_Decimal64`	64bit BID (IEEE-754R)
`_Decimal32`	128bit BID (IEEE-754R)
other implementation-specific types	
`__f16`	`arm` (16-bit half-precision floating-point)
`__float128`	`x86` (128-bit extended floating-point)
`__m64`	`x86` (64-bit packed vectors, MMX and 3DNow!)
`__m128`	`x86` (128-bit packed vectors, SSE and SSE-2)
`__m256`	`x86` (256-bit packed vectors, AVX)

Table 3.4: Overview of fundamental C types and some extensions.

important for type-safe interoperability with external C functions and C objects. “C is unusual in that it allows pointers to point to anything. Pointers are sharp tools, and like any such tool, used well they can be delightfully productive, but used badly they can do great damage.”(Pike, 1989)

Pointer types to a composite data type are often used in C API functions for the exchange of information records. As such, in a function as a parameter, virtually any memory address can be passed as pointers, also those which point to something else; this can damage other run-time data structures and also often leads to a crash of the running process.

Type information about a pointers base type can be used to reject dangerous calls; such as, if the base types of the argument object and the parameter type are not matching.

Even if base-type information is incomplete i.e. only the name of a structure is known, but nothing about its members. While incomplete information is useless for memory allocation and data field access, it can contributes to type-safety of foreign function calls.

3.6.5 Function types

Function types describe *callable* objects; the type gives the sequence of parameter types, the type of the return value and the calling convention.

Any object type except *bit fields* and `void` are valid as parameter types. While array types are implicitly passed by reference via pointers, other composite types, such as `struct` and `union`, are passed by value. The number of arguments can be large; *ISO C* gives recommendations for compiler designs to support at least up to 127 arguments, but in practice, C API functions usually use a much lower number of parameters.

Any object type except *bit fields* and *array types* are valid as return types. If no value should be returned the unit type `void` is used.

A function type implies a *Calling Convention* which specifies the details of the *Calling Sequence* for passing arguments and receiving results during a function call at machine level. C supports two kinds of calling conventions, one for functions with positional arguments and another for functions with variadic number of arguments. The arguments of variadic functions are divided into two parts. One part consists of a fixed number of positional arguments and the other part is variable in length, designated by the ellipsis symbol ’`...`’.

Furthermore, there exist platform-specific calling conventions that compilers support via non-standard extensions to C. Support for additional operating-system specific calling conventions is required, for example, on the Microsoft Windows 32-bit platform. The `stdcall` calling conventions is used for system libraries, such as `OpenGL`.

Aside from the indicator `...`, C offers no further keywords or syntax rules for the specification of calling conventions. It is rather common that compiler-specific syntax extensions to C are used; The `gcc` compiler expands the C language family with several extensions (see Stallman, 2003, Ch. 6). The statement `__attribute__( ( ) )` is used to annotate C objects, such as `struct`, `union`, `enum` as well as function declarations and definitions. A large number of Function Attributes are offered where a subset is reserved to specify the calling convention (see Stallman, 2003, Section 6.30). Function attributes can also be specified in a short form by prefixing `__` followed by the calling convention name, e.g. `__stdcall` to declare the `stdcall` calling convention. This syntax is compatible with that of Microsoft Visual C++ Compilers. The following table gives the short form for attribute and the corresponding calling convention.

Function Attribute	Calling Convention
`__cdecl`	Standard C
`__stdcall`	Windows System DLLs
`__thiscall`	C++ Member Function Calls
`__fastcall`	Passing via registers (non-standardized)

A few examples for declaring functions with different calling conventions are given below:

```
int f1(int x, int y) __fastcall;
int f2(int x, int y) __stdcall;
int f3(int x, int y) __cdecl;
typedef int (* __fastcall) fastcall_funptr_t)(int x, int y);
```

We close this brief overview on the abstract side of the C language which serves as the basis for a description of the encoding in the following section. Implementation details on C, such as data representation of basic data types and the implementation of calling convention across processor-architecture families are discussed in Section 5.3.

3.7 Text-based Encoding Format for C APIs

In this section we describe a compact text-based encoding for C API components, named *DynPort*. It is designed for language- and platform-independent creation of language bindings for C Libraries as outlined in 3.5.2. Furthermore, part of the syntax was incorporated in dynamic FFIs as text-based interfaces for the specification of type information.

3.7.1 Introduction

C API header files contain necessary information for run-time interoperability with their corresponding shared library. At compile-time, header files are parsed for compilation of user code so that later linking will function. However, they are not designed to be used as C API information resources at run-time, which is needed for dynamic creation of language bindings at run-time.

Object-oriented component middleware frameworks, such as Microsoft's COM (Box, 1998) and Mozilla's XPCOM (Cross-Platform Component Object Model) (McFarlane, 2003), connect scripting languages with C/C++ components. While a native component is implemented in C/C++, its interface is *explicitly* specified during development in IDL (Interface Definition Language), named *MIDL* and *XPIDL* for COM and XPCOM, respectively. An IDL specification is compiled to a "binary" *type library* file, with the suffix `.tlb` for COM and `.xpt` for XPCOM. "A type library (`.tlb`) is a binary file that stores information about a COM or DCOM object's properties and methods in a form that is accessible to other applications at runtime. Using a type library, an application or browser can determine which interfaces an object supports, and invoke an object's interface methods. This can occur even if the object and client applications were written in different programming languages."(Microsoft, 2012) (See also Box, 1998, p. 40, and Szyperski et al., 2002, pp. 330, 345-346 for further information on type libraries on COM.)

XPCOM was designed as an open alternative to COM and is available across platforms. These middleware frameworks are similar in their design, but they are not compatible. "Ideally, XPCOM typelibs would be binary-compatible with those generated by Microsoft's MIDL compiler, but the MS typelib format is proprietary." (Furman and Bandhauer, 2012)

In contrast to object-oriented C++ middleware frameworks, little was published about middleware frameworks for C libraries. `ctypeslib` (Kloss, 2008) is a framework for automatic creation of Python Bindings for C libraries. A parser translates C header files to Python code which, when executed, creates Python bindings for C libraries. However, the translated information is very specific to Python's `ctypes` FFI but not usable for other scripting languages.

3.7.2 Objectives

A major goal of DynPort is to offer a language-independent and platform-portable encoding format for C APIs. We propose a compact format with a simple structure, so that parser and scanner functions can be implemented with little effort in a variety of programming languages. Since scripting languages usually offer standard processing tools for text rather than for binary data, we use a text-based encoding format. For the text encoding, a small range of characters is used from the character set US-ASCII 7-bit. Encoded C API information can be embedded in character string objects of different languages, and thus type information can be transferred between language contexts.

C APIs are encoded by means of the abstract C type system for preserving the abstract platform-portable meaning. The mapping from high-level C data types to its low-level machine-specific data representation is implemented by a dynamic FFI.

In that the type information is transferable across languages and platforms, due to a text-based and abstract encoding, DynPort constitutes a platform-portable and language-neutral type library for C libraries; similar to type libraries of COM and XPCOM for C++, but with two differences.

Firstly, while COM is specific to a native platform, DynPort type information is used for cross-platform bindings to C libraries. Secondly, COM is an *intrusive* framework in that the components need to be written from the ground up i.e. a compiled wrapper is needed for making a C library a component. We consider C libraries as binary components that can be wrapped dynamically and without redefinition of their interface; our goal is to use existing C API header information as sources of information. This information is used for connecting shared libraries with scripting languages in a non-intrusive way with respect to the C library, across platforms.

3.7.2.1 Type signature

The term "type signature" is often used in the context of programming languages as the description for the type of a function object. Sometimes the term *signature of a function* is used instead and object-oriented languages, such as Java, use the term *method signature*. While type information is fundamental to statically typed programming languages, type signatures are also found in weak or dynamically typed languages for the external interface to native code.

Type signatures can be represented in different forms. In some languages type signatures are specified in the notation of the language for the definition of the function type, such as in Haskell, Erlang and C. In other languages, such as Python, Java and Objective-C, type signatures are encoded in a serialized text-based form. The latter was adopted for DynPort; we use text-based signature formats for function types, as well as for data type definitions.

3.7.2.2 Text-based Interface

Virtually every programming language supports the character string data type, which is used in functions as a parameter type for passing complex variable information. For example, the C output function `printf` uses a character string parameter for the specification of the output and formatting rules, which was widely adopted by other languages, such as Java, MATLAB, Python or Perl. Furthermore, the term "Regular Expressions" refers to a powerful language for pattern-based searching, matching and substitution in text processing. As a universal language, it is embedded across numerous scripting languages, C libraries, text editors and command-line tools.

Type signatures, encoded in character strings, are already used in FFIs for driving foreign function calls. The `FFI` module of Perl (Paul Moore, 2008) uses a text-based interface to denote the C function types as a serialized type signature string.

As a precursor to DynPort, we developed experimental FFI packages for various scripting languages, such as R, Lua, Python, Ruby and Shell. For each language we also utilized a common text-based interface and serialized type signature strings. In DynPort the core idea remains preserved; type signature strings are utilized to *drive* foreign function calls.

3.7.3 Encoding Format

We return now to a description of the *DynPort* encoding format. The notation is described first, then we describe the encoding for different components of the C API.

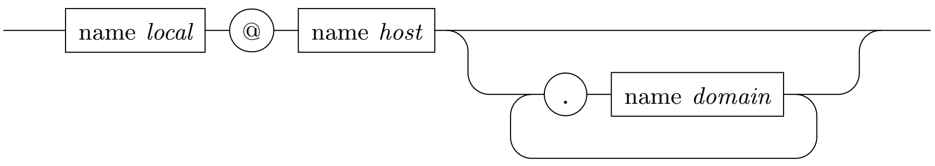
3.7.3.1 Notation of Syntax Diagrams

The syntax grammar of languages is often defined in BNF (Backus-Naur Form) notation. The notation comprises terminal elements, non-terminal elements and production rules for the specification of a syntax grammar. Terminal elements represent input tokens e.g. single characters, keywords or the *literal* text of a numeric value (for example, a sequence of digits). Non-terminal elements are defined by a single production rule that gives the sequence of terminal and non-terminal elements. In addition, the notation includes operators to denote repetition and optional elements. A grammar can be specified as a set of production rules and a top-level non-terminal element that represents the final element of a language, such as the "command-line" for an interpreted language or the "compilation unit" for a compiled language.

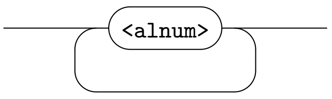
Syntax diagrams, also named Railroad diagrams, are a visual alternative to the formal BNF notation. Production rules are displayed as railroad network with a fixed start and end point at the left- and right-hand side, respectively. Terminal elements are represented by circles, labeled with single characters or a named token in angle brackets (e.g. '`<alpha>`' for any character of the alphabet). Non-terminal elements are represented as rectangles. Optionally, annotations are given following the names, in italic. Optional paths and repetitions are illustrated by outgoing branches (on track, or to the right-hand side of an element), and jointed incoming junctions (to the left-hand-side of an element), respectively.

Example We illustrate the use and interpretation of railroad diagrams by an example of the syntax for an electronic mail address format[1], such as `me@example.com`.

email



name



The syntax is given by a sequence of terminal and non-terminal elements. In the diagram the sequence is illustrated by traversing Railroad Diagram "email". The reader starts at the left-hand side and

[1]We give an example of a simplified syntax here. It is by no means the official format as specified in RFC 5322.

follows the railroad track in right-hand direction. By traversing possible paths, including repetitive or optional routes, the production is finished by reaching the right-hand side.

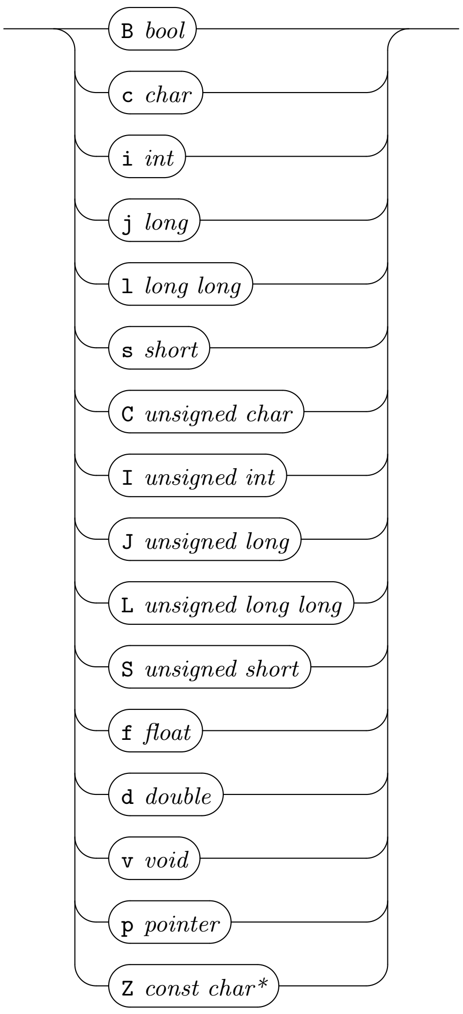
At first, Element “name” is reached, which is a non-terminal, depicted by a surrounding rectangle, so that the syntax is given by a different diagram, labeled ‘*name*’. As depicted in Diagram “name”, the syntax of a name comprises a non-empty sequence of terminal elements denoted with ‘`<alnum>`’. In general, terminal elements comprise simple lexical text patterns. We describe them in words rather than as a diagrams. We define ‘`<alnum>`’ as a single *alphanumeric* character in the range of ASCII characters ‘`a`’..‘`z`’, ‘`A`’..‘`Z`’ and ‘`0`’ .. ‘`9`’. Notice the ‘`<alnum>`’ is connected with a looping route that branches on the right-hand side and joins on the left-hand side. This denotes a non-empty sequence of consecutive alphanumeric characters, which also completes the syntax of a “name”.

We proceed back now to Diagram “name” and follow the route to the right of element “name” local. The next element that follows is the non-terminal ‘`@`’ followed by a “name” for the *host* part of an email address, as indicated by the note in italic. After that, if the next character in the input sequence is a ‘`.`’, we follow that route and reach another “name” for the *domain* part of an email address. As depicted in the diagram, this process is repeated until we finally reach the end of the rule indicated by an empty input sequence or a different character than ‘`.`’ or ‘`<alnum>`’ follows. If we want to ensure that the input is empty after the rule finishes, we would insert a terminal, for example ‘`<end>`’ (to indicate end of input), right before the right-hand side in the diagram. If the next input character of an sequence can not be used for moving along in the railroad network then the input sequence is considered to be invalid.

3.7.3.2 Basic types

The built-in basic data types of the C type system represent the fundamental building blocks. They are essential for the definition of function types and derived data types, such as pointers, arrays, `struct` and `union` types. We denote a basic C data types with a reserved character of the alphabet. Where possible, the character is derived from the initial character of the type name (excluding `unsigned`/`signed` prefixes), e.g. ‘`i`’ for `int` and ‘`s`’ for `short`. An exception to this is `long`, which is encoded with ‘`j`’ whereas `long long` is encoded with ‘`l`’. `unsigned` integer data types are encoded as upper case characters of corresponding character for `signed` data types, such as ’`I`’ for `unsigned int` and ‘`S`’ for `unsigned short`. The boolean C data type `_Bool` (`bool` in C++) is encoded as ‘`B`’. The `void` type is denoted by the ‘`v`’ character. Untyped pointers (pointers to incomplete data types) are expressed via ‘`p`’. Later we give a notation for typed pointers. ‘`Z`’ denotes the standard C string type `const char *`.

basic



3.7.3.3 User-defined structure and union types

User-defined record types are specified in terms of a sequence of field types and names. We give two examples for the definition of a user-defined `struct` and `union` and a corresponding type signature.

```
typedef struct
{
  int x, y;
  unsigned int w,h;
  const char * title;
} Window;
```

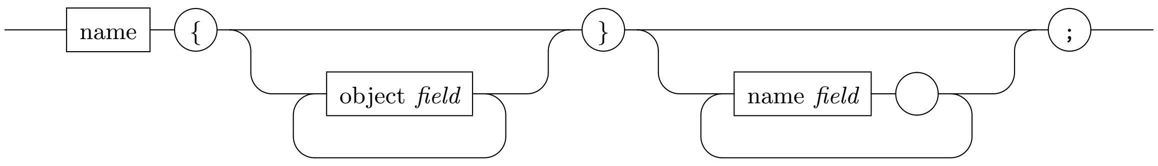
```
"Window{iiIIZ}x y w h title ;"
```

```
typedef union
{
  int          integer;
  double       real;
  const char * string;
} Val;
```

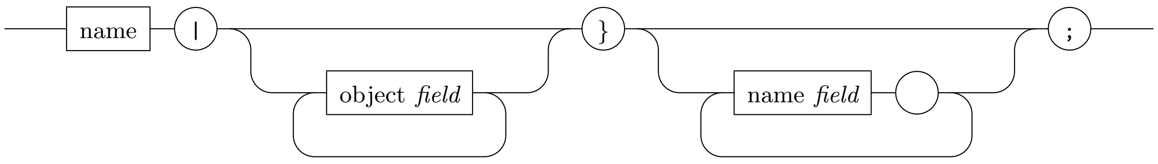
```
"Val|idZ}integer real string ;"
```

The syntax for the specification of the signature is given below:

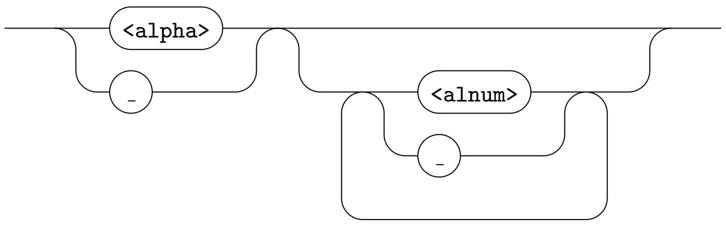
struct



union



name

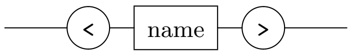


Signatures give the symbolic name (as defined by the last name of a `typedef` declaration), followed by a character to indicate the kind of type i.e. '`{`' for `struct` and '`|`' for `union` types. This is followed by a sequence of data type fields and terminated by a '`}`' character. Field data types are encoded with an object type signature; an extended version of the signature format for basic data types (see below for a description). This is followed by a corresponding sequence of field names. Field names are separated by a single whitespace character (ASCII code 32). The signature is terminated by a '`;`' character. A name begins with a single character of the alphabet or an underscore, followed by a (possibly empty) sequence of alphanumeric and underscore characters.

3.7.3.4 Object types

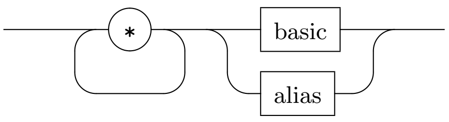
Object types are a combination of basic C data types, user-defined data types and also typed pointers. We encode user-defined data types by using an alias name that references a user-defined `struct` or `union` type by name. We enclose the name of a type name in angle brackets. The encoding of a type name is given next:

alias



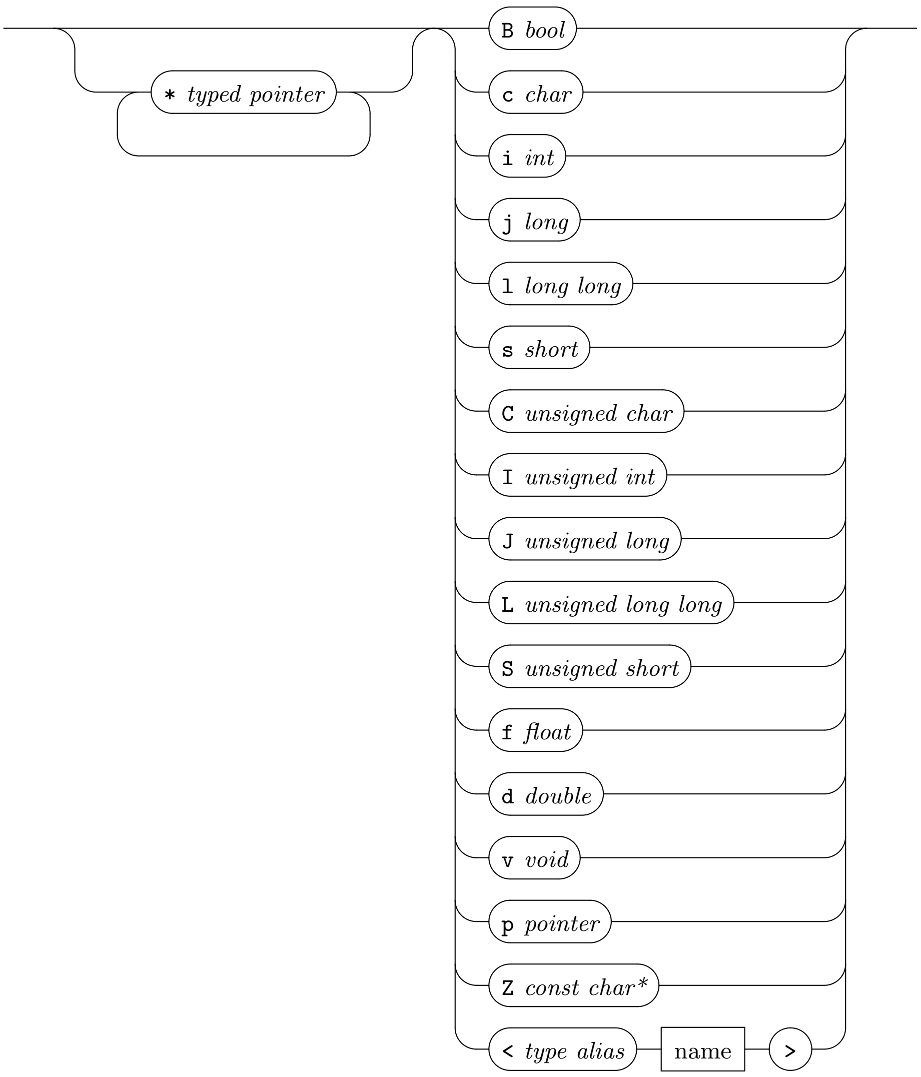
Aside from pointers to an incomplete types, denoted with '`p`', we also include support for typed pointers. We use an optional prefix sequence of '`*`', followed by it's base type, to denote typed pointers.

pointer



In combination, we give the syntax for object types, which integrates the two diagrams the above with the one for basic C data types:

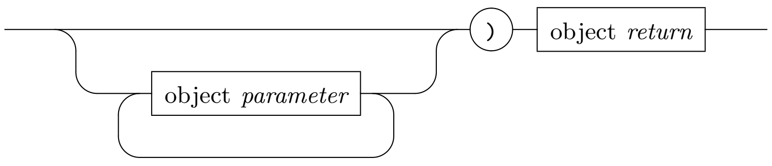
object



3.7.3.5 Function types

Function types are specified by a sequence of object signatures of the parameter types, followed by a '`)`' character and an object signature for the return type.

funtype

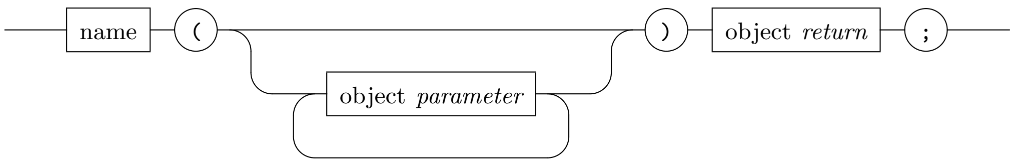


In the following, and also (for historical reason) in the R package `rdyncall`, we use the synonym *Call Signature* which refers to this encoding format.

3.7.3.6 Functions

We extend the above signature with a syntax for declaring public C functions, which includes the symbolic name of the function:

function



Below we give examples of signatures of C API functions:

```
int          SDL_Init        (Uint32 flags);
SDL_Surface* SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);
void         dpsifn          (double, int, int, int, double*, int*, int*);
```

```
SDL_Init(I)i;
SDL_VideoMode(iiiI)*<SDL_Surface>;
dpsifn(diii*d*i*i)v;
```

3.7.3.7 Symbolic Constants

A set of symbolic names for constant values, such as status return codes, enumeration constants or bit masks, are often defined as part of a C API. In C there are two alternative forms for their definition. Either the C `enum` type is used for declaring constant values or, in a more traditional way, constants are defined onto preprocessor level using the `#define` directive for macro definitions.

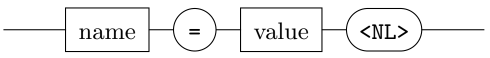
```
#define SDL_INIT_VIDEO    0x00000020
#define SDL_INIT_AUDIO    0x00000010
#define SDL_INIT_JOYSTICK 0x00000200
enum XML_Error {
  XML_ERROR_NONE,
  XML_ERROR_NO_MEMORY,
  XML_ERROR_NO_ELEMENTS = 4,
};
```

```
SDL_INIT_VIDEO=0x00000020
SDL_INIT_AUDIO=0x00000010
SDL_INIT_JOYSTICK=0x00000200

XML_ERROR_NONE=0
XML_ERROR_NO_MEMORY=1
XML_ERROR_NO_ELEMENTS=4
```

Symbolic constants are encoded using the following syntax:

constant



'`<NL>`' denotes a "new line"; it is encoded in ASCII using the *new line* character code (ASCII code 10). Optionally, for support of non-Unix platforms, it is also prefixed (Windows) or suffixed (Mac OS) by carriage return character code (ASCII code 13).

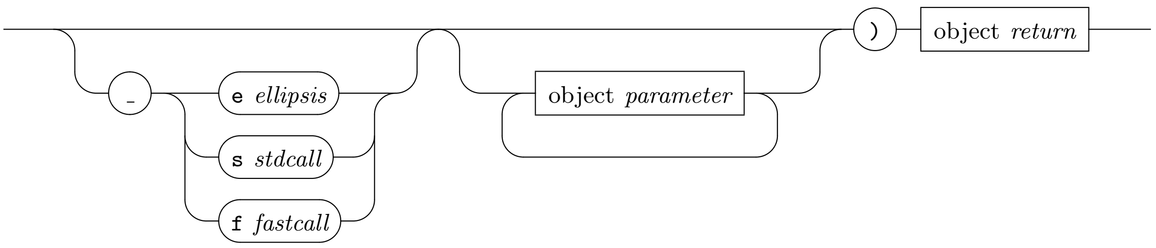
In principle, we encode a value as a line of text. For practical reason in `rdyncall`, values need to be encoded using a syntax that is compatible with R and C.

3.7.3.8 Function Types with Calling Convention

So far the encoding for function types does not provide information about the calling convention. As discussed in Section 3.6.5 we have to distinguish between at least two calling conventions for C; functions with a fixed number of arguments (the default), and those ending with a ellipsis "`...`" which denotes a variable number of arguments. Furthermore, specific to the `x86-32` platform, there are more calling conventions, that need to be considered here. For practical reason the `stdcall` is quite important in order to make use of functions of system libraries on Microsoft Windows 32-bit operating systems; for example, this is needed for using `OpenGL` functions on Windows. Other calling conventions, such as `fastcall`, are relatively seldom used in libraries.

We extended the encoding format for function types with an optional prefix notation, denoted with a '`_`' character and followed by a single alphabet character that encodes the calling convention. Otherwise, if the prefix is missing, the default C calling convention for functions with a fixed number of arguments is implied.

cccall



3.7.3.9 File Formats

DynPort type signatures are used in the R package `rdyncall` as an encoding format for text-based interface for various FFI services.

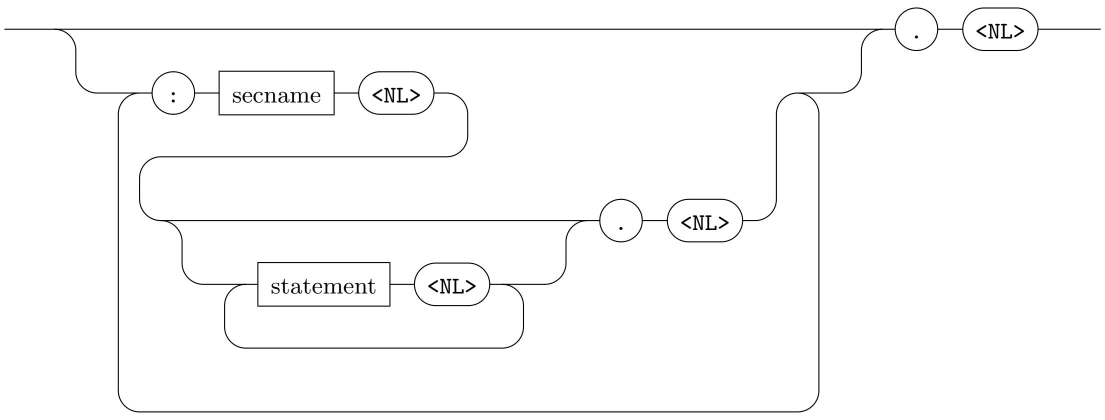
DynPort files specify cross-platform R bindings to C libraries, and they are also used in the package. The first experimental version of *DynPort* files, implemented in the `rdyncall` package, uses a representation as R script files. Listing 3 gives an example of a DynPort Version 1 file as it is used in the `rdyncall` package for cross-platform bindings to the `SDL` library. The R scripts comprise R expressions for the creation of wrappers, including service calls to `rdyncall`'s FFI services for creating function call wrappers and helper objects for foreign data objects. The *DynPort* type signature format is used for passing arguments to these service functions in terms of character strings. Notice that `rdyncall`'s FFI service functions work on batches of signatures. Hence, several signatures are processed in a loop. Thus several signatures can be packed in one large text block (e.g. for binding *all* shared C library functions at once).

```
# Shared Functions
dynbind( c("sdl","sdl-1.2","sdl-1.2.so.0"), "
SDL_Init(I)i;
SDL_SetVideoMode(iiiI)*<SDL_Surface>;
SDL_PollEvent(*<SDL_Event>)i;
")
# Aggregate Structure Informations
parseStructInfos("
SDL_Rect{ssSS}x y w h ;
SDL_MouseButtonEvent{CCCCSS}type which button state x y ;
SDL_KeyboardEvent{CCC<SDL_keysym>}type which state keysym ;
")
# Symbolic Constants
SDL_INIT_AUDIO=0x00000010
SDL_INIT_CDROM=0x00000100
SDL_INIT_VIDEO=0x00000020
```

Listing 3: Sample DynPort 1.0 file used by `rdyncall`

For a second implementation[2] of the Dynamic Bindings model for the Lua programming language, we designed a language-neutral *DynPort* file format. The file format is divided into sections. Each section addresses a different component type of C APIs, such as public functions, symbolic constants and data type definitions. Each element of a category is defined in a separate text line.

version2



Depending on the section name (*secname* the above), a different syntax is used for each line until the section terminates with a '`.`':

[2]Available at `http://dyncall.org/svn/dyncall/trunk/bindings/lua/luadyncall`.

Section	Description
`lib`	Short library names separated by '`|`'.
`fun`	Function signatures.
`const`	Symbolic contant signatures.
`struct`	`struct` type signatures.
`union`	`union` type signatures.

An example of a DynPort Version 2 file, as it is used in the Lua module `luadyncall`, is depicted in Listing 4

```
:lib
SDL|SDL-1.2|SDL-1.2.so.0
.
:fun
SDL_Init(I)i;
SDL_SetVideoMode(iiiI)*<SDL_Surface>;
SDL_PollEvent(*<SDL_Event>)i;
.
:struct
SDL_Rect{ssSS}x y w h ;
SDL_MouseButtonEvent{CCCCSS}type which button state x y ;
SDL_KeyboardEvent{CCC<SDL_keysym>}type which state keysym ;
.
:const
SDL_INIT_AUDIO=0x00000010
SDL_INIT_CDROM=0x00000100
SDL_INIT_VIDEO=0x00000020
.
.
```

Listing 4: Sample DynPort 2.0 file used by `luadyncall`

3.8 Parser Framework

In this section we describe a flexible framework for parsing C API header files to encode type information, as well as symbolic constant names and values in the *DynPort* format.

3.8.1 Overview

The framework is largely based on a modified version of the GCC compiler, named GCC-XML (King, 2004) (the command-line tool is named `gccxml`), which parses C/C++ sources and outputs a tree of definitions and declarations, encoded in the standard document markup format XML (Extensible Markup Language). Numerous data management and processing tools are provided for XML. Among them, XSLT (Extensible Stylesheet Language Transformations) provides a programming language for transformation of XML documents to user-defined output formats (XML, XHTML or raw text). (See Clark, 2001 for specifications.) An *XSLT processor*, such as `xsltproc` (Veillard et al., 2009), parses an input XML document as node tree, and outputs a transformed version, according to an XSLT stylesheet file. Aside from `gccxml` and XSLT/`xsltproc`, further tools are incorporated in the framework: a standard C preprocessor (e.g. `gcc`), the Unix program `grep` (Kernighan and Pike, 1983, Section 4.1) for filtering, a custom C/C++ preprocessor based on Boost `Wave` (Kaiser, 2011); the latter is superseeded by using `gcc` and `awk` (Kernighan and Pike, 1983, Section. 4.4).

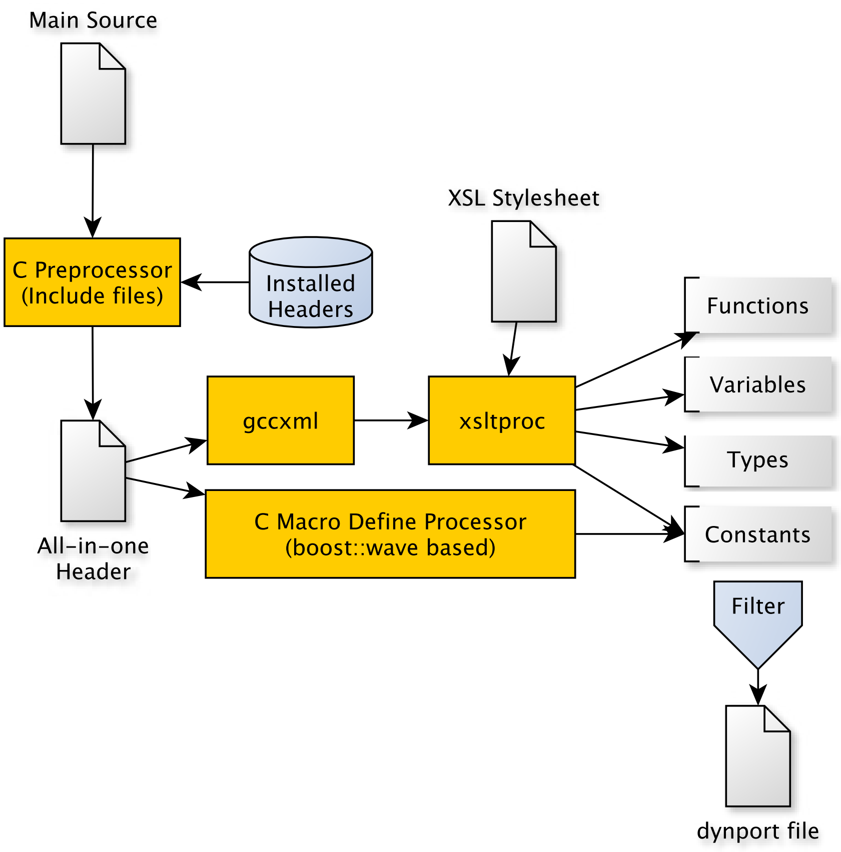


Figure 3.5: Design of the *DynPort Tool.*

Figure 3.5 illustrates the design of the *DynPort* tool. The tool comprises a chain of programs. As main input a simple C file (depicted as "Main Source") is created manually that references that C API header, which should be transformed to the DynPort format via an `#include` statement. Corresponding C API header files and system headers for C development should have been installed in advance on the build system.

In a first stage a temporary file (depict as "All-in-one") is created, comprising content of all referenced header files, after which it is passed as input to `gccxml`. As a result a node tree structure of the C sources is outputted, encoded in XML.

In a next step the XML file is processed by the XSLT processor `xsltproc`. The processor also reads an XSLT stylesheet file, which specifies *C to DynPort* transformation rules. Then the transformation is performed on the input XML document. As a result elements of the C API are outputted in the DynPort format, as is defined by the stylesheet file.

Since C API headers put reference to all kinds of external headers, the input to `gccxml` is rather a mix of C and System APIs. Thus we use a “Filter” to exclude elements of external header files. As depict in Figure 3.5, the output of `xsltproc` is split into separate passes; one for each category type. Each pass produces a separate fragment of text data encoded in the DynPort format. We use separated C API elements for a fine-grained filter.

Symbolic constants of a C API, which are defined as preprocessor macro definitions (in contrast to `enum`), need to be handled by a custom preprocessor. In the following section we describe details of implementation for the `gccxml/xslt` transformation from C to DynPort, the filter and the custom preprocessor.

3.8.2 Implementation

We illustrate the translation of C to custom file formats using XSLT and `gccxml/xsltproc` by an example. Listing 5 shows a translation of a sample C code (*left*) to the DynPort 2.0 file format (*right*).

C file ‘`example.c`’:

```c
/* Data structures: */
typedef struct Rect {
  int x, y;
  unsigned short width, height;
} Rect;
typedef struct Window Window;
/* Functions: */
Window* newWindow     (unsigned int flags);
void    setWindowRect (Window* w, Rect* r);
```

$\longrightarrow$

DynPort 2.0 file ‘`example.dynport`’:

```
:fun
newWindow(I)*<Window>
setWindowRect(*<Window>*<Rect>)v
.
:struct
Rect{iiSS}x y width height ;
Window{};
.
.
```

Listing 5: DynPort Tool implementation example: C to DynPort.

The translation is performed in two stages:

1. `gccxml` parses the C source `example.c` and outputs the XML document `example.xml` (see Listing 6).
```
$ gccxml example.c -fxml=example.xml
```
2. `xsltproc` transforms `example.xml` to DynPort `example.dynport` using transformation rules from `dynport.xsl`.
```
$ xsltproc dynport.xsl example.xml >example.dynport
```

Before we describe the XSLT language, we give a brief introduction to XML; a detailed description is given in Bray et al. (1997).

3.8.2.1 XML and `gccxml` XML output

XML is a text-based markup language for hierarchically structured document types. In XML, a document is logically structured as a tree of nodes with a single root. XML nodes have a type, given as a tag name, a number of attributes (pairs of name/value) and nested content, such as child nodes or text content blocks. Diverse kinds of document types can be encoded in XML. Document types can be defined in schema documents, such as encoded in a DTD (Document Type Definition) or XSD (XML Schema). For a given document type, the corresponding schema defines the set of types, mandatory or optional attributes, and the rules for hierarchical nesting structure of node types in a tree. Nodes are specified by opening and closing tags, beginning with '`<`' and '`</`', respectively, followed by the tag name and a '`>`' character. Text and child nodes can be declared between the tags, e.g. `<Function>` and `</Function>`. Opening tags of a node can have attributes after their tag name in the form ⟨*name*⟩'`=`'⟨*value*⟩ that are assigned to that node. There exist a short notation for leaf nodes where an opening tag is terminated by '`/>`', such as `<Argument type="_10"/>`.

```xml
<?xml version="1.0"?>
<GCC_XML>
  <Namespace id="_1" name="::" members="_3 _4 _6 _5 _8 _7 "/>
  <Namespace id="_2" name="std" context="_1" members=""/>
  <Function id="_3" name="setWindowRect" returns="_9" context="_1" extern="1">
    <Argument type="_10"/>
    <Argument type="_11"/>
  </Function>
  <Function id="_4" name="newWindow" returns="_10" context="_1" extern="1">
    <Argument type="_12"/>
  </Function>
  <Struct id="_5" name="Window" context="_1" incomplete="1"/>
  <Typedef id="_6" name="Window" type="_5" context="_1"/>
  <Struct id="_7" name="Rect" context="_1" members="_13 _14 _15 _16 _17 _18 " bases=""/>
  <Typedef id="_8" name="Rect" type="_7" context="_1"/>
  <FundamentalType id="_9" name="void"/>
  <PointerType id="_10" type="_6"/>
  <PointerType id="_11" type="_8"/>
  <FundamentalType id="_12" name="unsigned int"/>
  <Field id="_13" name="x" type="_19" context="_7"/>
  <Field id="_14" name="y" type="_19" context="_7"/>
  <Field id="_15" name="width" type="_20" context="_7"/>
  <Field id="_16" name="height" type="_20" context="_7"/>
  <Constructor id="_17" name="Rect" artificial="1" throw="" context="_7">
    <Argument name="_ctor_arg" type="_21"/>
  </Constructor>
  <Constructor id="_18" name="Rect" artificial="1" throw="" context="_7"/>
  <FundamentalType id="_19" name="int"/>
  <FundamentalType id="_20" name="short unsigned int"/>
  <ReferenceType id="_21" type="_7c"/>
  <CvQualifiedType id="_7c" type="_7" const="1"/>
  <File id="f0" name="test2.c"/>
</GCC_XML>
```

Listing 6: DynPort Tool implementation example: XML output of GCC-XML.

Listing 6 depicts the XML output file of `gccxml` after parsing the C source file `example.c`. The document begins with a conforming XML header, followed by a root XML node `<GCC_XML>`. Various entities of the C source `example.c` are nested as child of from `<GCC_XML>`. Since we consider transformations of this document, it is necessary to become familiar with its structure (compare with `example.c` in Listing 5). It is worth noting that the hierarchical structure of the `<GCC_XML>` tree is rather flat. For example, C `struct Rect`, encoded as `<Struct>`, and its members, encoded as `<Field>`, are inserted on the same nesting level of the XML node tree. All XML nodes are labeled using the `id` attribute. As an exception, argument types of functions, encoded as `<Argument>`, are nested as children of the `<Function>` XML node. Various relationships, including the link between composite types and their members, are realized by symbolic references using the `id` of the target, which is assigned for the source's attribute, such as `returns` and `type`.

Some of the nodes, depicted in Listing 6, are not considered in our example, such as `<Namespace>`, `<Constructor>` and `<File>`; they can be ignored in the following discussion.

3.8.2.2 Transformation of XML to Text via XSLT and XPath

We now discuss the implementation of XML transformations via XSLT using the example of stylesheet file `dynport.xsl`.

XSLT is written as a standard XML document. In contrast to our example outputting text, XSLT is usually used for output of "XML" documents. As a consequence, the XSLT stylesheet comprises two XML schemes or dialects i.e. tags of XSLT and of output format, in one file; so that ambiguous naming conflicts arise. Therefore XSLT tag names are usually prefixed with `xsl:` to distinguish them from tags of the XML output format. But in our case we use text output, and thus we use a default XML namespace for XSLT and omit tag prefixes[3] which increases readability of given XSLT listings.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<stylesheet version="1.0" xmlns="http://www.w3.org/1999/XSL/Transform">
  <output method="text"/>

  <template match="/">
    <!-- template for the root tree -->
  </template>

  <template match="//Function">
    <!-- template for all Function child nodes of root -->
  </template>

  <template match="//Struct">
    <!-- template for all Struct child nodes of root -->
  </template>

</stylesheet>
```

Listing 7: Outline of XSLT stylesheet files.

The basic structure of an XSL style sheet is illustrated in Listing 7. The file comprises a standard XML header followed by the document's root node `<stylesheet>`. Then the output mode is specified to as 'text' (otherwise defaults to XML), followed by several `<template>` tags.

In XSLT transformations are defined by a set of template rules, specified by the `<template>` tag. First of all the XSLT processor (`xsltproc`) parses the input XML document as a node tree representation,

[3] XML namespaces provide a means to solve ambiguity in XML; they also serve to identify the document type of an XML file. Namespaces represent XML dialects, such as XHTML or XSLT, which are specified by a namespace URI (Uniform Resource Identifier). For example, the official namespace URI for XSLT is `http://www.w3.org/1999/XSL/Transform`. Namespace prefixes are defined as pseudo-attributes of the root node using the syntax `xmlns:`⟨*prefix*⟩`=`⟨*URI*⟩ to declare a user-defined prefix with a link to the document type's namespace URI. Subsequent child tags that refer to that namespace need to be prefixed. A default namespace for tags without prefix, such as used in our example, is specified via the pseudo-attribute `xmlns=`⟨*URI*⟩.

as well as the XSLT stylesheet; the latter gives the templates. Then the actual transformation process begins by searching a template to proceed with.

A template comprises a sequence of operations, such as printing text content to the output via `<text>` tags. Templates are instantiated for a matching subset of the input document, given by a string value of the template's attribute `match`. The pattern matching process of XSLT is expressed in a string-based language, named XPath (XML Path language). XPath expressions can select nodes of a specific tag name, or those that are nested in a specific hierarchical configuration. For example, XPath expression `"//Function"` selects all `<Function>` nodes that are child nodes of the root node. Table 3.5 gives an overview of expressions that are used throughout this example. (See Clark et al., 1999 for specifications of XPath.)

XPath Expression	Description
`"/"`	root node
`"//Function"`	`<Function>` child nodes of root
`"//Function/Argument"`	`<Argument>` child nodes of `Function`, which are child nodes of root
`"Argument"`	`<Argument>` child nodes of the current node
`"@name"`	value of the attribute `name` of the current node
`"//*"`	all child nodes to the root node
`"//*[@id=_2]"`	all child nodes with an attribute `id` that equals to '`_2`'
`"//*[@id=$type]"`	all child nodes with an attribute `id` that equals to variable 'type'
`"//Typedef|//CvQualifiedType"`	`<Typedef>` *inclusive* or `<CvQualifiedType>` child nodes of root

Table 3.5: Examples of XPath expressions.

As an initial transformation step, the processor searches a template that matches the root node, and which then can be instantiated. So, we define a main template by using the XPath expression `"/"`:

```
<template match="/">
  <text>:fun&#10;</text>
  <apply-templates select="//Function"/>
  <text>.&#10;</text>
  <text>:struct&#10;</text>
  <apply-templates select="//Struct" mode="def"/>
  <text>.&#10;</text>
  <text>.&#10;</text>
</template>
```

By this template, we define the general outline of output for the DynPort format, including the order of sections. For our example, we plan the following: First we output function signatures, and then struct signatures. For each section a header is outputted, such as "`:fun`" and "`:struct`", terminated with a new line character (ASCII 10), encoded in XML encoding with the escape code '`&#10;`'.

After output of a header via `<text>...</text>`, further transformations are activated via:

`<apply-templates select="..."/>`

This tag prompts the processor to start a new template matching process; a subset of the input tree is selected via an XPath expression, given by the attribute value of `select`. Then, templates are searched that matches on the new input, which is then instantiated.

For the first section `fun:`, all C function declarations are selected from the `gccxml` input document, given by the XPath expression `"//Function"`. For the second section `struct:`, all C structure definitions are selected, given by the XPath expression `"//Struct"`. Notice the total transformation is implemented as an interaction between `<template>` and `<apply-templates>`.

This style of programming resembles the traditional style of function definitions and function calls. However, as a result of a single `<apply-templates>`, several templates can be instantiated, depending on the selected input and the matching templates.

In the following we give listings of `XSLT` template code as a the left-hand panel. At the right-hand side we give text fragments of `<GCC_XML>` input, `DynPort` output, and notes, depending on the context.

For the transformation of C function declarations from XML to DynPort function signatures, we define a template that matches `<Function>` child nodes of root, using the XPath expression `"//Function"`, in order to output the function signatures of C function declarations.

```
<template match="//Function">
  <value-of select="@name"/>
  <text>(</text>
  <apply-templates select="Argument"/>
  <text>)</text>
  <variable name="X" select="@returns"/>
  <apply-templates select="//*[@id=$X]"/>
  <text>;&#10;</text>
</template>
```

Input nodes (depict with children):

```
<Function id="_3" name="setWindowRect" returns="_9">
  <Argument type="_10"/>
  <Argument type="_11"/>
</Function>
```

```
<Function id="_4" name="newWindow" returns="_10">
  <Argument type="_12"/>
</Function>
```

Final output (with all template code from below):

```
newWindow(I)*<Window>;
setWindowRect(*<Window>*<Rect>)v;
```

In general, XSLT templates are instantiated in a context of a current node or node list, so that operations in a template can be executed relative to a matching node or list of nodes. The tag `<value-of select="@name"/>` evaluates an XPath expression `"@name"` within the current node's context and prints the result. The prefix `"@"` is used for referencing attribute names (and selecting their value) instead of tag names. Thus the XPath expression `"@name"` gives the value of the `name` attribute of the current node `<Function>`'. As a result the name of the C function (e.g. `setWindowRect` and `newWindow`) is outputted. In accordance with the format for function signatures, the name is separated by generating a '`(`' character. We then apply templates on all `<Argument>` children relative to the current node by using an XPath expression without a root prefix (`"//"`). As a result, a sequence of argument type signatures is generated; we discuss the definition of this template below. After all argument types are processed, we output a '`)`' character, followed by applying a template to generate the return type signature. In contrast to the arguments of a function, the return type is not given as a child node to `Function` but as an attribute value.

As we noted earlier, the hierarchical structure of `<GCC_XML>` documents is flat; associations between nodes are given by symbolic reference between attributes, such as `returns` and corresponding XML nodes with a matching `id`. Thus we need to follow the reference from `<Function>` to its return type via `returns` symbolically: We save the value of attribute `returns` of `<Function>` in a variable, named `X` using `<variable>` with the XPath expression `"@returns"`.

In a next step the template

```
<apply-templates select="//*[@id=$X]"/>
```

is executed; `"//*"` selects all nodes of a tree, while the expression `"[@id=$X]"` selects all subsequence nodes with an attribute `id` that equals to a value of the variable named `X`. In what follows we use this technique whenever we need to apply templates to symbolic relatives. Often we will also need to *delegate* to a different node symbolically.

Now we give the template for handling argument types.

```
<template match="//Function/Argument">
  <variable name="X" select="@type"/>
  <apply-templates select="//*[@id=$X]"/>
</template>
```

Sample context node:

```
<Argument type="_12"/>
```

Corresponding selection:

```
<FundamentalType id="_12" name="unsigned int"/>
```

We delegate to a node with the `id` that equals to `type` attribute.

We now give a template to generate DynPort's object type signatures for the C basic data types.

```
<template match="FundamentalType">
<variable name="name" select="@name"/>
<choose>
<when test="$name='void'">              <text>v</text></when>
<when test="$name='char'">              <text>c</text></when>
<when test="$name='unsigned char'">     <text>C</text></when>
<when test="$name='signed char'">       <text>c</text></when>
<when test="$name='short int'">         <text>s</text></when>
<when test="$name='short unsigned int'"><text>S</text></when>
<when test="$name='int'">               <text>i</text></when>
<when test="$name='unsigned int'">      <text>I</text></when>
<when test="$name='long int'">          <text>j</text></when>
<when test="$name='long unsigned int'"> <text>J</text></when>
<when test="$name='float'">             <text>f</text></when>
<when test="$name='double'">            <text>d</text></when>
</choose>
</template>
```

Input node:

```
<FundamentalType id="_9" name="void"/>
```

```
<FundamentalType id="_19" name="int"/>
```

Corresponding outputs: "`v`" and "`i`".

Basic C data types are encoded in `<GCC_XML>` as `<FundamentalType>` nodes where its `name` attribute gives the basic C type name. We store the attribute of its name in a variable first and then use a `<choose>` block of `<when>` conditions; each condition compares the variable with a standard C basic data type name. If a match occurs the corresponding basic type character signature is outputted.

We need to specify templates for handling pointer types as well, which are encoded in `<GCC_XML>` as `<PointerType>` tags. For the pointer type, we output a '`*`' prefix before we delegate symbolically to its base type using attribute `type`. Analogous to return types of functions and the pointer's base types, `<TypeDef>` and `<CvQualifiedType>` (which denote `const` and `volatile` C type qualifiers) also need delegate symbolically. Templates for `<Struct>` and `<Union>` types output their name, enclosed by angle brackets; '`<`' and '`>`' are encoding in XML as '`&lt;`' and '`&gt;`', respectively.

```
<template match="PointerType">
  <text>*</text>
  <variable name="type" select="@type"/>
  <apply-templates select="//*[@id=$type]"/>
</template>
<template match="//Typedef|//CvQualifiedType">
  <variable name="type" select="@type"/>
  <apply-templates select="//*[@id=$type]"/>
</template>
<template match="Struct|Union">
  <text>&lt;</text>
  <value-of select="@name"/>
  <text>&gt;</text>
</template>
```

A sample matching node:

```
<PointerType id="_10" type="_6"/>
```

After following the symbolic reference:

```
<Typedef id="_6" name="Window" type="_5"/>
```

After following the symbolic reference:

```
<Struct id="_5" name="Window" context="_1"/>
```

Final output: " `*<Window>` "

We have completed all templates for generating of the section for function signatures. Now, we give the implementation for data type signatures.

In order to process all C structure data types, we need a second template for `<Struct>` using a XPath selector `"//Struct"`. As we already defined a template with a similar select (see the last template above), we need to define a second template with a different `mode`. The `mode` is used as an attribute in `<template>` and `<apply-templates>`; it is an additional user-defined selector/match value for `<template>` and `<apply-templates>`, which needs to match.

```
<template match="//Struct" mode="def">
  <value-of select="@name"/>
  <text>{</text>
  <variable name="i" select="@id"/>
  <apply-templates select="//Field[@context=$i]"/>
  <text>}</text>
  <apply-templates
    select="//Field[@context=$i]" mode="n"/>
  <text>;&#10;</text>
</template>

<template match="//Field">
  <variable name="t" select="@type"/>
  <apply-templates select="//*[@id=$t]"/>
</template>

<template match="//Field" mode="n">
  <value-of select="@name"/>
  <text> </text>
</template>
```

Sample input document:

```
<Struct id="_7" name="Rect"/>
```

Subsequent sequence (for the above):

```
<Field id="_13" name="x" type="_19" context="_7"/>
<Field id="_14" name="y" type="_19" context="_7"/>
<Field id="_15" name="width" type="_20" context="_7"/>
<Field id="_16" name="height" type="_20" context="_7"/>
```

Final Output:

```
Rect{iiII}x y width height;
```

The name of the C `struct`'s is outputted, followed by a '`{`' character. Then we process all `<Field>` nodes via symbolic reference. Note that we need to do two rounds; first the types, and then the names of the field. So we use a `mode="n"` for the second pass. In between, we output a '`}`' character. Finally, we terminate the signature with a '`;`' character.

Our example is now complete. We illustrated the use of `gccxml` and open-source XML tools for the implementation of C/C++ code transformations.

3.8.2.3 Parsing C Preprocessor Defines

We presented a flexible framework for parsing C header files. But it is limited to C/C++ declarations. Symbolic constants, declared in C via `#define NAME VALUE`, are not *seen* by a C/C++ compiler, such as `gccxml`. The reason is that the preprocessing stage, including the definition and application of macros, is separated from the compilation phase in the translation model of C. (See ISO, 2011, Section 5.1.1.2 for details.) Hence, the C/C++ compiler does not even see macro names, but corresponding replacement parts on input performed by the preprocessor.

As a consequence we implemented a custom C/C++ preprocessor, using Boost `Wave` preprocessor parser C++ library (Kaiser, 2011). But the maintenance costs for this tool were large relative to the other programs used in this framework. As an alternative, we developed a lighter solution, based on GCC and `awk` (Kernighan and Pike, 1983, Section 4.4). The tool has two stages: First, `gcc` is invoked with the initial source file via `gcc -E -dM` to dump all macro definitions at the end of the preprocessing task.

```
# gcc -E -dM includeSDL.c
```

The following listing gives a sample output:

```
#define SDL_INIT_TIMER 0x00000001
#define SDL_INIT_AUDIO 0x00000010
#define SDL_INIT_VIDEO 0x00000020
#define SDL_INIT_CDROM 0x00000100
[... further output omitted ...]
```

Secondly `awk` was utilized for transformation into the right form; the program is given next:

```
#!/usr/bin/awk -f
/#define/ {
  printf("%s=",$2);
  for (i=3;i<NF;i++)
    print("%s ",$i);
  printf("%s\n",$1);
}
```

As a result, we get:

```
SDL_INIT_TIMER=0x00000001
SDL_INIT_AUDIO=0x00000010
SDL_INIT_VIDEO=0x00000020
SDL_INIT_CDROM=0x00000100
```

3.8.2.4 Filter

In general, C API headers refer to external header files of the C Standard Library, the operating system and third-party libraries. As a result, the DynPort toolchain generates a large number of signatures for C entities that are not part of the particular C API. Since C symbols are managed in a single namespace (except for tag names of `struct`, `union`, `enum`), name clashes are prevented by convention, in that C API symbols are prefixed with a short name (for example, `SDL` functions are prefixed with `SDL_`).

A filter, based on `grep`, was incorporated as a last step in order to filter the output fragments before assembly into a single *DynPort* file. Since all DynPort signatures for public functions, data types and constants are written in text lines, beginning with their symbolic name; all output fragments can be filtered by `grep` likewise.

3.9 Summary

We presented a middleware architecture for scripting languages and shared C libraries that provides scripting access to C APIs across platforms.

We introduced the topic with a brief overview of shared C libraries and emphasised their significance as reusable binary building blocks for virtually all applications on current platforms. We discussed binary interoperability and emphasized the role of C compiler as a *static glue* between binary components at the Application Binary Interface of platforms. We noticed that language bindings to C libraries are usually implemented by C wrapper code. However, the development of wrapper code for each library-language combination and subsequent compilation for each platform is obviously a disadvantage. We noted that the use of standard code generators also inevitably necessitates some manual development tasks. At the latest, when building for multiple platforms, subsequent work is needed for specification of linking. Instead of using wrappers in C and compilation for language bindings, we suggested using the dynamic methods, provided by Foreign Function Interfaces of scripting languages, for interoperability with binary components at run-time.

We identified two subcategories of FFIs, namely static and dynamic, and illustrated the limitations of the former using the example of the built-in FFI of R. While dynamic FFIs seem to be advantageous in this context, we noted that these require type information to provide dynamic bindings to C APIs but that this information can be extracted from C header files.

Based on our considerations, we developed the Dynamic Bindings model and outlined it's architecture that comprises three abstraction layers. The first layer provides a cross-platform portable *type library* format for C API information, and a corresponding C header parser tool. The second layer defines the language extension that implements automation of dynamic bindings using a dynamic FFI and data from the first layer. Finally, the third layer provides a platform-abstraction interface to the dynamic linker, ABIs and calling conventions of processor hardware architectures for implementation of dynamic FFIs.

We gave an outlook of the potential advantages of our proposed model for software development of language bindings in comparison with the compilation-based method. Whereas compiled bindings are developed on per language-and-library basis, and are built as an additional binary wrapper per-platform, dynamic bindings are created per library via automation tools and are then available across all supported platforms and languages. We concluded that the dynamic bindings model reduces development time, fosters rapid scripting of cross-platform applications, and can significantly decrease the gap between scripting languages and C library developments. In the last part of this chapter we proposed an implementation of the first layer of the architecture. Firstly, we gave an overview of the C type system, which provides the foundation for developing an encoding format for C APIs. We then presented the design of a text-based encoding format for C API information, named DynPort. Finally, we described a parser framework for translation from C header files to DynPort files. The framework is based on XML/XSLT- and Unix-tools, such as `gccxml`, `xsltproc`, `gcc`, `grep` and `awk`. Furthermore, we illustrated in detail the use of `gccxml` and XSLT/XPath for processing C type information and transformation to custom file formats using the example of DynPort.

In summary, this chapter gave an overview of the *Dynamic Bindings* model and a detailed description of the first of its three layers, named *DynPort*. For the remaining layers, we discuss an implementation for the R programming language in Chapter 4. Layer 3, an abstraction layer to the ABI across five processor architecture families, is covered in Chapter 5.

Chapter 4

Dynamic FFI for R

In this chapter we present a dynamic FFI for the R programming language, contributed as package `rdyncall` (Adler, 2012) on CRAN. The package comprises a toolkit of components for working with shared C libraries and provides interoperability with natively compiled C code and run-time C data objects. Collectively the components constitute an implementation of the *Dynamic Bindings Model*, as discussed in Chapter 3, for the R programming language; it provides a *simple cross-platform interface* for loading dynamic R bindings to C APIs, such as those offered by the `OpenGL` and `SDL` C library families.

4.1 Introduction

First we give a motivating example to use `rdyncall` as the base for rapid cross-platform development of application software followed by an outline of the package architecture. The first section closes with a brief overview of the R language and the anatomy of R language objects; it provides background information for subsequent discussion on the components of the package, interoperability between R and C, and the implementation of the *Dynamic Bindings Model*. We give listings of C API user code, side by side in C and R, to illustrate the syntactic resemblance of R and C when C APIs are used. The chapter closes with an example of a real-time visualization and simulation software, written in R, using `OpenGL` and `SDL` via `rdyncall`, and is followed by a summary.

Example We consider a visualization program that utilizes high-performance graphics output via `OpenGL` for a statistical application offered by packages of CRAN in R. Further we require that the software runs across platforms so that we also utilize the `SDL` library as a portable abstraction layer to the windowing system, user-input event processing and access to additional multimedia hardware such as audio output.

As a first step we develop a skeleton of the application that comprises basic setup methods for OpenGL graphics output and processing of user input events. As an initial test for using OpenGL, a rectangle is drawn within a loop, made of two triangles that are filled with interpolated colors across the four corners. Figure 4.1 gives screenshots of the application skeleton for three major R platforms. The

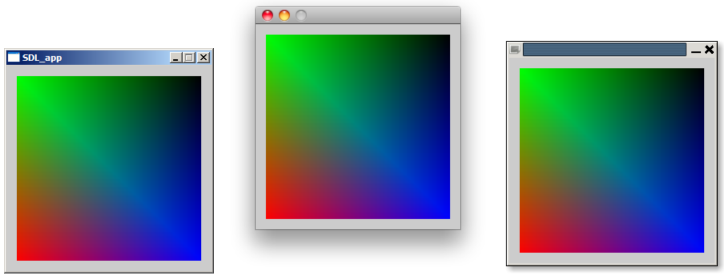


Figure 4.1: Screenshot: Cross-platform R application using SDL and OpenGL on Windows, Mac OS X and Linux.

application skeleton is implemented twice; the sources are given side by side in Listing 8. The left panel gives the C code and the right panel the R code using `OpenGL` and `SDL` via the `rdyncall` package.

```c
#include "SDL.h"
#include "SDL_opengl.h"

int main(int argc char* argv[]) {
  int quit = 0;
  SDL_Event e;
  SDL_Init(SDL_INIT_VIDEO);
  SDL_SetVideoMode(640,480,32,
    SDL_OPENGL|SDL_DOUBLEBUF);
  while(!quit) {
    glClearColor(0.0,0.0,1.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLE_STRIP);
    glColor3f(1.0,0.0,0.0);
    glVertex2f(-0.9,-0.9);
    glColor3f(0.0,1.0,0.0);
    glVertex2f(-0.9, 0.9);
    glColor3f(0.0,0.0,1.0);
    glVertex2f( 0.9,-0.9);
    glColor3f(0.0,0.0,0.0);
    glVertex2f( 0.9, 0.9);
    glEnd();
    SDL_GL_SwapBuffers();
    while(SDL_PollEvent(&e)) {
      if (e.type == SDL_QUIT) {
        quit=1;
      }
    }
  }
  return 0;
}
```

```r
library(rdyncall)
dynport(SDL)
dynport(GL)
main <- function() {
  quit <- FALSE
  e <- new.struct(SDL_Event)
  SDL_Init(SDL_INIT_VIDEO)
  SDL_SetVideoMode(640,480,32,
    SDL_OPENGL+SDL_DOUBLEBUF)
  while(!quit) {
    glClearColor(0.0,0.0,1.0,0.0)
    glClear(GL_COLOR_BUFFER_BIT)
    glBegin(GL_TRIANGLE_STRIP)
    glColor3f(1.0,0.0,0.0)
    glVertex2f(-0.9,-0.9)
    glColor3f(0.0,1.0,0.0)
    glVertex2f(-0.9, 0.9)
    glColor3f(0.0,0.0,1.0)
    glVertex2f( 0.9,-0.9)
    glColor3f(0.0,0.0,0.0)
    glVertex2f( 0.9, 0.9)
    glEnd()
    SDL_GL_SwapBuffers()
    while(SDL_PollEvent(e)) {
      if (e$type == SDL_QUIT) {
        quit <- TRUE
      }
    }
  }
}
main()
```

Listing 8: C and R: Basic multimedia application skeleton based on `OpenGL` and `SDL`.

In both cases components of the two C APIs are distinguished by their prefix i.e. `SDL` components are prefixed by "`SDL_`" while `OpenGL` components are prefixed by "`gl`" and "`GL_`". Note that the R code is very similar to C; the differences are marginal: `SDL_OPENGL+SDL_DOUBEBUF` is used instead of `SDL_OPENGL|SDL_DOUBLEBUF` due to the lack of bit-wise operations in R[1]. The allocation of temporary C

[1]The value of symbolic constants, `SDL_OPENGL` and `SDL_DOUBLEBUF`, represent independent bit flag patterns that

`struct` data structures is done via `new.struct`; field member access to C structure objects is done in R via `$` operator.

The C version of the application needs to be compiled and linked with the shared libraries on each supported platform. Table 4.1 gives an overview for building the C application across platforms. In essence, platform- and library-specific installation paths of run-time and development files all need to be considered for each port. Some library development packages offer helper shell scripts to retrieve library-specific build options, such as `sdl-config` for SDL, but in general, platform-specific adaptions need to be taken into account for each port.

Platform/Compiler	Build instructions
Ubuntu Linux	$ gcc -c ‘sdl-config --cflags‘ main.c
	$ gcc -o example main.o ‘sdl-config --libs‘ -lGL
NetBSD 5.1	$ gcc -c ‘sdl-config --cflags‘ -I/usr/X11R7/include example.c -o example.o
	$ gcc -o example main.o ‘sdl-config --libs‘ -L/usr/X11R7/lib -lGL
Mac OS X 10.7	$ gcc -c -I/Library/Frameworks/SDL.framework/Headers SDLMain.m -o SDLMain.o
	$ gcc -c -I/Library/Frameworks/SDL.framework/Headers example.c -o example.o
	$ gcc -o example example.o SDLMain.o -framework SDL -framework Cocoa -framework OpenGL
Windows/MinGW32	> gcc -c -I/devel example.c -o example.o
	> gcc -o example.exe main.o -lmingw32 -L/devel/lib -lSDL -lSDLmain -lOPENGL32
Windows/Visual C	> cl /c /MD /IC:\devel\include main.c
	> link main.obj /OUT:example.exe /SUBSYSTEM:CONSOLE \
	/LIBPATH:C:\devel\lib\x86 SDL.lib SDLmain.lib OPENGL32.lib

Table 4.1: Compilation of C user applications based on OpenGL and SDL across platforms.

For example, on NetBSD, in contrast to Linux, a specific location for the OpenGL header files and libraries needs to be specified for the compiler toolchain. On Mac OS X the files `SDLMain.h` and `SDLMain.m` need to be copied from the SDL development package in advance. On Microsoft Windows the build instructions for using the Microsoft Visual C compiler are significantly different. It should be noted that build systems, such as `CMake` (Martin et al., 2003) and `GNU make` (Stallman and McGrath, 2002), substantially reduce the amout of work for building portable software, but such tools also require a steep learning curve. In summary, we can conclude that cross-platform application development using a compiled language, such as C/C++, requires platform-specific expertise and a pool of (virtualized) hardware and software tools for building multi-platform software releases. For the C version further work is still needed to incorporate the R interpreter; this is beyond the scope of this example; see R Development Core Team (2012c, Section 8.1) for details on embedding R and package `RInside` (Eddelbuettel and Francois, 2012), which provides a cross-platform C++ API for this purpose.

The R version in the example runs across platforms without compilation. Naturally the precompiled run-time packages of `OpenGL` and `SDL` need to be installed; see Section 4.7.3 for detailed installation instructions for major R platforms. In Section 2.3.2 we outlined the `rgl` package which offers a real-time visualization device system. However, `rgl` provides a scene graph using OpenGL and advanced

can be combined by "+" in R which is equivalent to bit-wise "or" in this case; alternatively this operation is provided by the package `bitops` (Dutky and Maechler, 2012).

methods need to be implemented in C++. Since the core C APIs of `OpenGL` and `SDL` are *directly* accessible from within R via `rdyncall`, a similar degree of freedom is achieved as it is the case for C but without considering platform-specific build and linking issues. Furthermore, the full range of R packages and the C APIs (given in Table 4.10) *both* become accessible.

4.1.1 Software Architecture

The software architecture of the `rdyncall` package is represented in Figure 4.2. The package comprises of several components that are depicted as stacked in order of dependency; components at a higher level are implemented by means of components below them. The language for the implementation of each component is indicated by the color background: R (blue), C (green) and assembly language (gray).

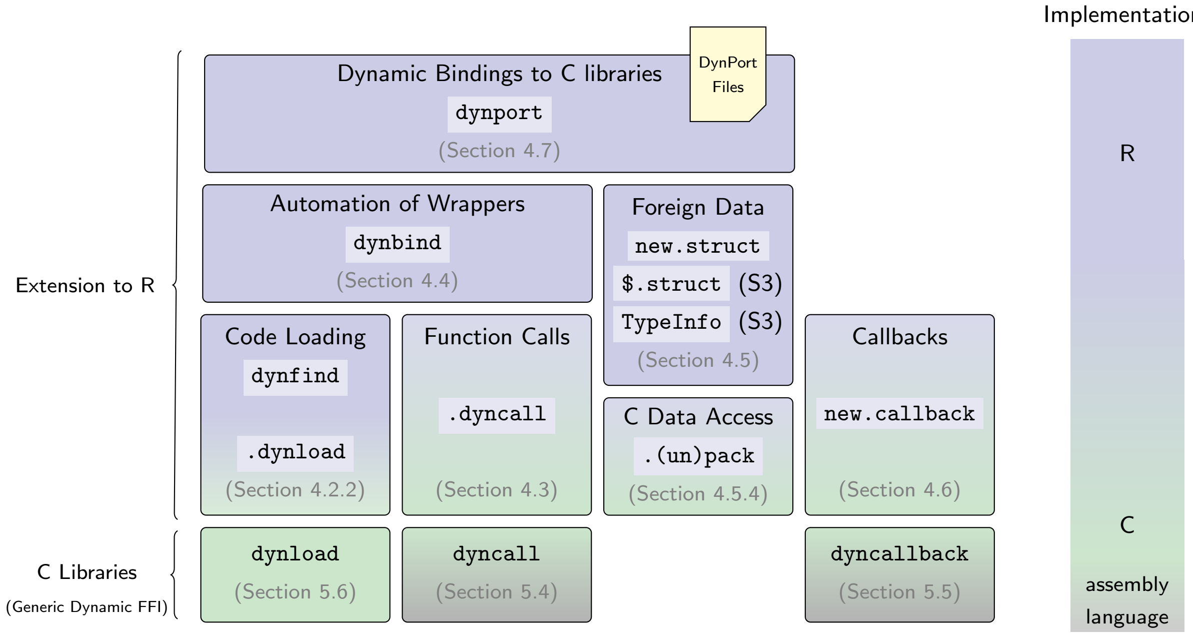


Figure 4.2: Software architecture of R package `rdyncall`.

`dynport` represents the main interface for loading dynamic R bindings to C libraries; the bindings creation process is data-driven using a repository of *DynPort* files, installed with the package, where each file specifies cross-platform R bindings to a specific C API. `dynbind` is an automation tool for the creation of a set of function call wrappers to shared library functions. `dynfind` provides a cross-platform interface for naming the shared libraries to be searched and loaded. `.dynload` is an alternative interface function to the dynamic linker of the operating system. `.dyncall` offers a dynamic function call facility for R and is the core component of the dynamic FFI. The component block in Figure 4.2, comprising `new.struct`, `$.struct` (S3) and `TypeInfo` (S3), represents a framework for data-level interoperability with C `struct` and `union` data types. `.pack` and `.unpack` offer a low-level interface for read/write operations with support for the basic set of C scalar data types. `new.callback` wraps

R functions as C closure objects so that R functions can be written and passed to shared libraries as C callback function pointers. The `rdyncall` package is based on a Generic Dynamic FFI, depicted by the bottom layer of Figure 4.2; it comprises the three C libraries, namely `dynload`, `dyncall` and `dyncallback`.

4.1.2 Usage of Type Signatures

Several facilities within the `rdyncall` package incorporate text-based type signatures as part of their interface, which uses the *DynPort* encoding format, as described in Section 3.7:

Component	Functions	Signature Type
Function Calls	`.dyncall`	Call Signature
	`.dyncall.<CALLMODE>`	Call Signature
Callbacks	`new.callback`	Call Signature
Foreign Data	`parseStructInfos`	Struct Signature
	`parseUnionInfos`	Union Signature
C Data Access	`.pack`	Object Signature
	`.unpack`	Object Signature
Automation of Wrappers	`dynbind`	Function Signature

The `.dyncall` FFI uses the type signature to specify the sequence of parameter types and the return type of a foreign C function. This information is used for type-safe value conversion between dynamic R objects and statically typed C argument values and return values. `new.callback` wraps a user-defined R function as a C function pointer object so that it is callable from C; it uses type signatures to specify the C function type. It is used in a similar context as in `.dyncall` but in the opposite direction; it converts the arguments from C to R and the results from R to C.

C `struct` and `union` data types are also specified using a signature to describe the sequence of field member types and their names in order to compute platform-specific memory requirements for allocation and member field offsets for symbolic member-field access. In order to read and write various C data types the `.pack` and `.unpack` functions use type signature character code for the specification of the desired C scalar basic type.

`dynbind` uses a sequence of function signatures for the creation of R wrapper functions to a set of C library functions, using `.dyncall` as implementation for generated wrappers.

4.1.3 Internals of R

As preparation to subsequent sections we give a brief overview of the internals of the R language. The core of R, including the interpreter and the representation of language objects, is implemented in C. In this chapter we consider details of the representation of R language objects to figure out efficient and type-safe techniques for data transfer between R and C, in particular when data need to be exchanged via C pointers.

Access to the internals of R is given by the C API header files of R (e.g. `Rinternals.h`) and details are described in R Development Core Team (2012c) and R Development Core Team (2012b).

4.1.3.1 Object Types

R is a dynamic programming language with a relatively large set of different object types. While most of the types are accessible via the interpreter, some of these types are internal. Table 4.2 summarizes all available language objects (of R 2.14); it gives the string output of `typeof` and sample expressions whose value is of such an object type.

Kind	`typeof()`	C Type ID	Examples/Comments
Atomic Vectors	`logical`	`LGLSXP`	`TRUE`, `FALSE`, `as.logical(0)`
	`integer`	`INTSXP`	`123L`, `as.integer(c(1,2,3))`
	`double`	`REALSXP`	`123`, `as.double(c(1L,2L,3L))`
	`complex`	`CPLXSXP`	`1+0i`, `as.complex(c(1,1,0))`
	`character`	`STRSXP`	`"hello"`, `as.character(c(1,2,3))`
	`raw`	`RAWSXP`	`as.raw(c(65,43,23))`
Linked Lists	`language`	`LANGSXP`	`quote(x+y)`
	`pairlist`	`LISTSXP`	`pairlist(1L,2,"3")`
Vectors	`list`	`VECSXP`	`list(1L,2,"3")`
	`expression`	`EXPRSXP`	`expression(x+y,a+b*rnorm(10))`
Special objects	`NULL`	`NILSXP`	`NULL`
	`externalptr`	`EXTPTRSXP`	`getLoadedDLLs()[["base"]][["handle"]]`
	`environment`	`ENVSXP`	`new.env()`
	`symbol`	`SYMSXP`	`as.symbol("hallo")`
	`closure`	`CLOSXP`	`function()NULL`
	`builtin`	`BUILTINSXP`	`get("+")`
	`special`	`SPECIALSXP`	`get("if")`
	`S4`	`S4SXP`	`typeof(getClass("NULL"))`
	`weakref`	`WEAKREFSXP`	key/value references
	`promise`	`PROMSXP`	argument slot for lazy evaluation
	`char`	`CHARSXP`	single character strings
	`...`	`DOTSXP`	argument placeholder
	`any`	`ANYSXP`	type signature marker
	`bytecode`	`BCODESXP`	byte code
		`NEWSXP`	internal
		`FREESXP`	internal

Table 4.2: Overview of R object types: R type name, the corresponding C symbolic constant S-Expression type ID and sample expressions that evaluate to that value type.

4.1.3.2 S-Expressions

In R all objects, whether they are atomic vectors, linked lists, vectors or special objects, such as environments, function closures, control-flow entities (`if` , `while` , `for`) or primitive language objects ('...'), are represented using a common data structure header. The common structure is named an *S-Expression*.

We now take a closer look at the internal structure of the representation of S-Expressions in R. The header contains vital information for the interpreter, the garbage collector, debugging facilities and also for the low-level C interface. The header is illustrated in Figure 4.3; the first field consists of a 32-bit wide bit field, followed by a dedicated pointer for attributes (which is a list with named entries), and a node header (double-linked list) for chaining objects for garbage collection.

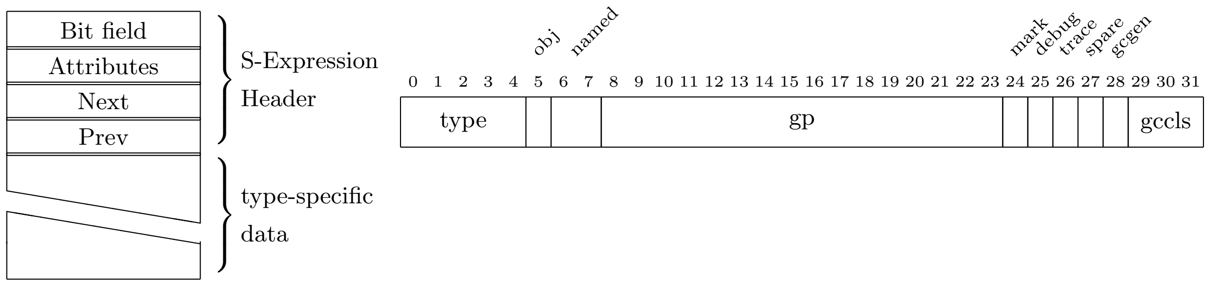


Figure 4.3: S-Expression header (*left*) and top-level bit field (*right*).

4.1.3.3 Type Identification

Since R is a dynamically typed language, interpreter and user-defined functions need to determine actions at run-time. The interpreter processes a sequence of R objects, and for each object in turn, it determines the action to perform, starting with the type id in the S-Expression, and whether it is an atomic vector or a list object, a variable name (that needs to be looked up) or a language list object representing a function call. Some user-defined functions check input argument objects and transform the data to a target form (e.g. `as.integer` to *coerce* an input vector to an `integer` atomic vector). While type checking is often not needed in high-level user-defined functions, where it is delegated to low-level functions, type checking becomes very important where basic operations are implemented in C, or where the control-flow is passed to a foreign C function. So the most primitive operation that can be applied uniformly to any R object is type identification. Currently there are 24 different R object types each having a designated number encoded in the first five bits of the header bit field. C symbolic constants for identification of the S-Expression type are given in Table 4.2. Depending on the type information, the rest of the S-Expression is expected to have a specific format.

We are primary interested in R objects that are used for data exchange with external C libraries. Thus we focus on objects that carry data values; these are atomic vector objects for data values, external pointer R objects as containers for C pointers, as well as special objects such as `NULL` . We also consider language list objects for processing arbitrary sequences of R arguments.

Figure 4.4 gives an outline of the data structure of atomic objects and language list objects.

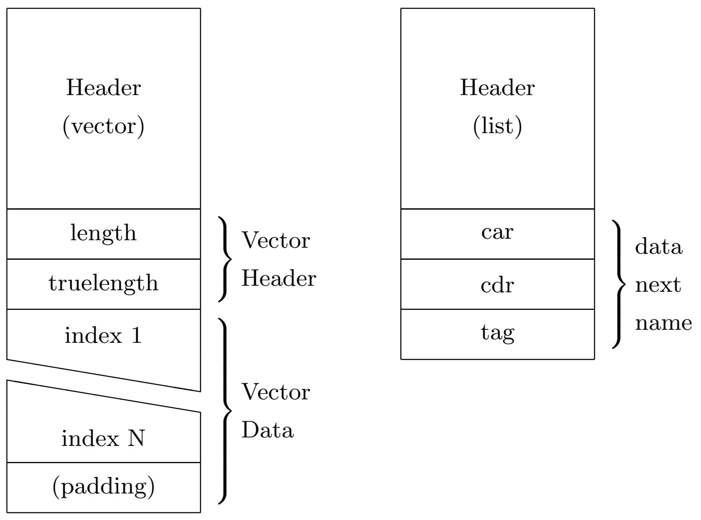


Figure 4.4: Atomic vectors and language lists in R.

4.1.3.4 Atomic Vectors

Atomic vectors are used as containers for data of homogeneous type, where data is tightly packed in a continuous region of memory. As with all R objects, they begin with an S-Expression header, followed by a common header for atomic vectors containing a length field. Elements are stored using the representation of C data types using the following mapping between R types and their internal C representation:

`logical`	$\longrightarrow$	`int[]`
`integer`	$\longrightarrow$	`int[]`
`double`	$\longrightarrow$	`double[]`
`complex`	$\longrightarrow$	`Rcomplex[]`
`raw`	$\longrightarrow$	`unsigned char[]`
`character`	$\longrightarrow$	`SEXP[]`

Note that `complex` is implemented as a `struct Rcomplex`, which consists of two `double`; atomic vectors of R, except for the `character`, are represented as a flat data structure of a C array of scalar C data types. Character strings are slightly more complex in general because in contrast to basic scalar data types, such as integer values, floating-point numbers and address pointers, because they have a variable length. As a consequence languages use their own data structure scheme for strings. For example, in Pascal strings consist of an array of characters prefixed with a length field of the string. In C strings are implemented as an array of characters with a terminating null character. In R the `character` R object is a vector of character strings using an array of pointers to S-Expression objects of type `CHARSXP`, each of which represents a single character string which is used for character data but also for `symbol` objects. Since R version 2.6.0 `CHARSXP` S-Expressions are partly managed in a cache for reuse; see R Development Core Team (2012b, Sec. 1.10) for details. Note that even a single scalar data value is held as a vector of length 1. In R the default representation for literal numeric values is `double` even if the value is representable as an `integer` object, which consumes less memory. Integer representation of literal values is enforced by giving a suffix `L`.

Example Values can be encoded in different data types in programming languages. In the following example we illustrate the representation of a numeric vector of four elements which is coerced to different R object types. Figure 4.5 gives an illustration of the run-time data structure of different R atomic types. The widths of the boxes are proportional to the memory usage.

```
x <- c(0,1,2,1)
```

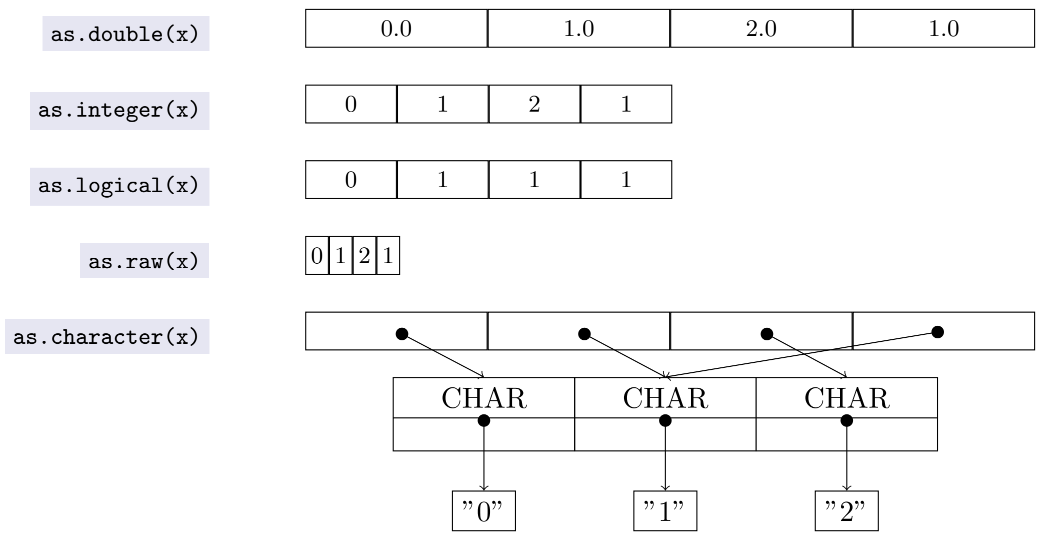


Figure 4.5: Different data representations of R numeric data.

Note the complex pointer structure that is used for the `character` vector.

4.1.3.5 Language lists

The implementation of R objects is strongly based on the Scheme language (a member of the LISP-based family of languages). In Scheme and LISP both code and data are represented by lists.

Language lists are small building blocks with a singly linked structure using three pointers, namely `car`, `cdr` and `tag`, as depicted in Figure 4.4. `car` points to data, such as an atomic vector, to refer to constant values, or to a symbolic name object to refer to a variable, or function name. `cdr` points to the next element in the chain of the list or to the singleton `NULL` object, to mark the end of the list. `tag` is used to specify the symbolic name of an argument in a function call.

Example Every function call in R is represented as a language list object. For example, the call expression `rnorm(100,mean=3,sd=4)` is represented as an S-Expression. Figure 4.6 illustrates its run-time data type structure.

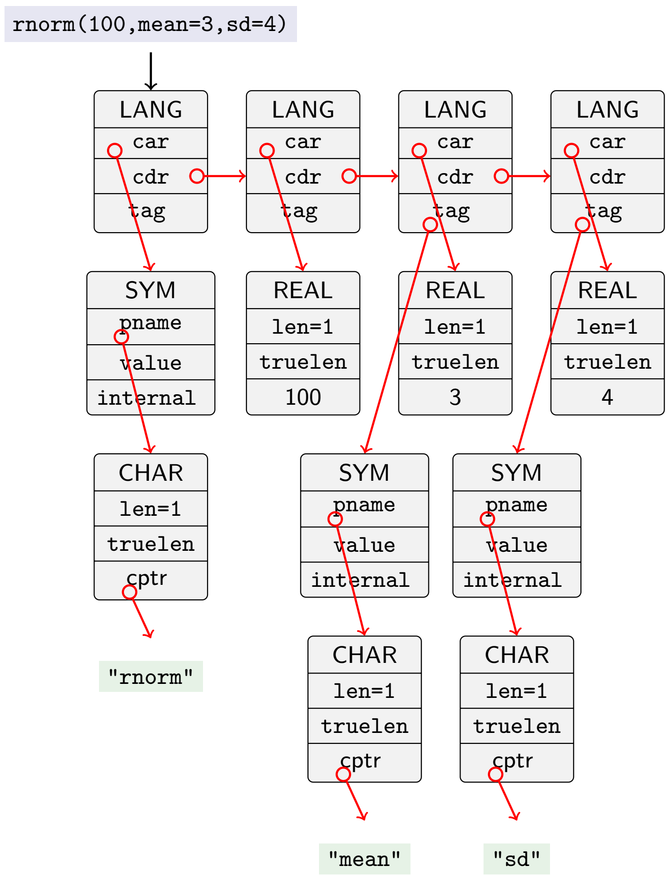


Figure 4.6: Internal representation of a call object in R.

The call expression is evaluated as a function call by the interpreter. Language list objects represent a chain of list objects to link other objects as a sequence. The head of the call expression points to the symbol object that represents name of the function to be called, followed by the first argument of the call, and so on. The last element of the call expression points to the *nil* object to indicate the end of the list (not depicted). Note that the arguments of the call expression are scalar values, each represented as a separate R atomic vector object of length 1.

4.1.3.6 Other language objects

The `NULL` object: In R `NULL` refers to a non-value type, sometimes refered to as *nil* in other languages to express non-value semantics. It does not have a state; it exists as a single entity within the R interpreter.

Vector-based lists Atomic objects are used for storing data in a uniform encoding scheme. In order to support structured data objects with varying data types, R offers a list object type for storing data objects (of possibly different atomic types) as a sequence. The internal representation resembles that of atomic vectors, but a pointer to S-Expressions is used as elementary scalar type of the vector. While this data type is flexible, it is very specific to the implementation of R and can not be used as-is in foreign language contexts. Thus we do not consider specific support for exposing this data type to C, in contrast to the atomic vector objects (with special care for `character`) which can be used for the exchange of C array data.

External Pointers External pointers are rarely used in R but they are important when working with C libraries. In Section 3.3.2 we discussed data-level interoperability and the need to determine the right *carrier* for transfer of pointer values; external pointers are exactly for this purpose. The value copying semantics for these objects is different. While R objects are usually copied when they are passed as arguments, external pointers (and also environment objects) are passed as references.

4.1.3.7 Memory Management

The R run-time system includes a generational garbage collector that provides for automatic memory management. Users allocate new objects; the garbage collector detects if data become *unreachable* and frees memory. R's garbage collector does not move objects; we can expose internal data to C APIs, but we need to make sure that the data are protected from garbage collection as long as they are needed by the C library. See R Development Core Team (2012b, sec 1.7) for details on the garbage collector of R.

4.2 Code loading

Before C functions and global data objects of *external* shared libraries can be used from within an interpreter, the library has to be loaded first. This is done dynamically using the dynamic linker of the operating system. Since the naming of a shared library is not standardized across platforms, the loading of a shared library represents a problem to cross-platform development. The naming scheme is specific to a particular operating-system platform and the location of the library can also depend on system configuration choices.

We illustrate this problem by comparing the file paths of shared libraries across a number of operating systems. Using the example of the standard C `math` library we show the limitation of the `base` R interface to the dynamic linker for searching and loading shared libraries across platforms. We then present an alternative facility for working with shared libraries in R that offers a *unified* naming scheme for searching and loading of shared libraries that works across platforms; it also offers automatic unloading of shared libraries, which is more suitable for working with *external* libraries. We also discuss the differences between the shared library formats and details of searching methods among dynamic linkers across operating systems.

4.2.1 Introduction

The `base` package of R offers the `dyn.load` function for loading shared libraries:

```
lib <- dyn.load(x)
```

If the library is found an S3 object of class `DLLInfo` is returned that represents an opened shared library in R so that the addresses of functions can be resolved by their symbolic names via the R function `getNativeSymbolInfo` or the extract operator `'$'`.

```
address <-lib$<SYMBOL>$address
```

However, the *name* of the shared library to be loaded, given by `x`, needs to be specified as a *platform-specific absolute file path.* It is this that makes cross-platform development with shared libraries rather impractical.

Consider, for example, the standard C `math` library that is usually installed on all systems. On Unix-based platforms it is commonly named "`m`". On Windows it is part of the C run-time library named "`MSVCRT`" (Microsoft Visual C Run-Time). The following table gives the file path of the shared C library `math` for this and a number of operating systems:

Operating system	File path of standard C `math` shared library
Mac OS X	`/usr/lib/libm.dylib`
Windows	`C:\WINDOWS\SYSTEM32\MSVCRT.DLL`
Debian 7 on `x86-32`	`/lib/i386-linux-gnu/libm.so.6`
Debian 7 on `x86-64`	`/lib/x86_64-linux-gnu/libm.so.6`
Fedora Linux 18 on `x86-64`	`/lib64/libm.so.6`
ArchLinux 2013.5	`/lib/libm.so`
Solaris 10	`/usr/lib/libm.so`
NetBSD 6.1	`/lib/libm.so`
FreeBSD 9.1	`/usr/lib/libm.so`
OpenBSD 5.3	`/usr/lib/libm.so.7.1`
DragonFly BSD 3.2	`/usr/lib/libm.so`

In order to write cross-platform R user code that loads the library, the platform needs to be identified to select a corresponding hard-coded platform-specific file path:

```
> path <- switch( Sys.info()[["sysname"]],
  Darwin    = "/usr/lib/libm.dylib",              # for Mac OS X
  Windows   = "/WINDOWS/SYSTEM32/MSVCRT.DLL",
  Linux     = "/usr/lib/i386-linux-gnu/libm.so.6",  # for Debian-based Distributions on x86-32
  SunOS     = "/usr/lib/libm.so",                 # for Solaris
  NetBSD    = "/usr/lib/libm.so",
  FreeBSD   = "/usr/lib/libm.so",
  OpenBSD   = "/usr/lib/libm.so.7.1",
  DragonFly = "/usr/lib/libm.so"
)
> libm <- dyn.load(path)
```

The number of platforms covered by above code is still incomplete; it would require extended code to handle cases where the file path includes versioning information, which may change between revisions of a library and can be different among platforms. In the case of Linux-based platforms, the above code does only support Debian-based Linux Distributions for `x86-32` architectures; other architectures and also other Linux Distribution families often have a different directory scheme. Furthermore, for other non-standard libraries, such as `SDL`, the path can also depend on the selected installation method or package management system, in particular on systems such as Mac OS X using MacPorts, Solaris using OpenCSW, and DragonFly BSD using DPorts. It is obvious that the `dyn.load` interface is inconvenient for *cross-platform* development with shared libraries; the absolute file paths need to be arranged and tested on a large number of systems including different distributions and processor architectures.

In general, the dynamic linker provides capabilities for *searching* a shared library by the library's file name *without given an absolute directory path*. However, `dyn.load` implicitly disables these search capabilities because the file path is *filtered*; if the character string, given by `x`, is not an absolute path, i.e. `x` does not begin with a '`/`', the current working directory is prepended, so that the dynamic linker *always* receives an absolute path.

Even when search capabilities of the dynamic linker are used, the name for a shared library significantly differs across platforms. On Windows, the `math` library is named "`MSVCRT.DLL`", on Mac OS X it is "`libm.dylib`" while on the others it can be looked up via "`libm.so`" or "`libm.so.6`" where the latter needs to be used on Linux platforms.

4.2.2 Interface

The `rdyncall` package offers the function `dynfind` for searching and loading shared libraries from within R. It provides a unified interface using a cross-platform naming schema; the implementation uses the direct interface of the dynamic linker in order to search libraries. The interface is as follows:

```
lib<-dynfind(libnames, auto.unload=TRUE)
```

`dynfind` attempts to load a shared library using a collection of platform-specific *hints*. The shared library is specified by `libnames` given as a vector of character strings. Each string represents a *short library name* that gives a hint about the name of the shared library. The library name is given by its distinguishing naming part without platform-specific prefixes and suffixes in which built-in platform-specific heuristics are used to derive possible file paths. For specific libraries and in order to support several platforms, multiple hints are often needed, such as a suffix form, which includes versioning information.

Example Suppose cross-platform R code is needed to load the `math` C library. `dynfind` can be used for this task by giving a list of hints to open the `math` library across all major platforms:

```
> libm <- dynfind(c("msvcrt","m","m.so.6"))
> libm
<pointer: 0x103813070>
attr(,"path")
[1] "/usr/lib/libm.dylib"
attr(,"auto.unload")
[1] TRUE
```

In the example three hints are required to cover cross-platform loading of the `math` library; the first hint `"msvcrt"` is used for loading the standard C library on Windows, the second hint `"m"` usually works across all other platforms except for certain Linux installations, which are handled by the third hint `"m.so.6"` that includes a major version number. As a result of this call an external pointer R object is returned that represents a *handle* to a shared library. Furthermore the *handle* can be used for operations such as resolving of symbols via the function `.dynsym(handle, symbol)`.

```
> sqrtAddress <- .dynsym(libm, "sqrt")
> sqrtAddress
<pointer: 0x7fff89e872d0>
```

The list of short library names depends on the naming characteristics of a particular shared library and its installation across platforms. Table 4.3 gives the file paths of four shared libraries, namely `C`, `expat`(Clark, 2007), `OpenGL` and `SDL`, on several operating systems. The full file path is given in `gray`; the name required for searching via the dynamic linker is given in `black`. The short library name, needed by `dynfind`, is `underlined`. We also give the list of short library names for each of the four shared libraries that is needed to load the library via `dynfind`.

Common platform-specific prefixes and suffixes are automatically appended to a short library name internally. For most cases the distinguished name part of the library is sufficient for `dynfind` to search and open a shared library. But there are platform-specific issues of naming shared libraries that require to use a list of names. For example, on Mac OS X there are two forms of shared libraries, on Windows some libraries, such as C, are named completely differently, and on Linux and Solaris version numbers need to be identified.

4.2.3 Naming Schemes of Shared Libraries

The *file path* of a shared library can be partitioned into a *directory location* and a *file name* component, where the former depends on a particular operating system, processor architecture, operating-system

Platforms	Shared library `C`	Shared library `OpenGL`
Windows	`%WINDIR%\SYSTEM32\MSVCRT.DLL`	`%WINDIR%\SYSTEM32\OPENGL32.DLL`
Mac OS X 10.7	`/usr/lib/libc.dylib`	`/System/Library/Frameworks/OpenGL.framework/OpenGL`
Solaris 10	`/lib/libc.so`	`/usr/lib/libGL.so`
ArchLinux 2013.5	`/lib/libc.so`	`/usr/lib/libGL.so`
Debian 6	`/lib/libc.so.6`	`/usr/lib/libGL.so.1`
Ubuntu 12.04	`/lib/i386-linux-gnu/libc.so.6`	`/usr/lib/i386-linux-gnu/libGL.so.1`
FreeBSD 9.1	`/usr/lib/libc.so`	`/usr/local/lib/libGL.so`
OpenBSD 5.3	`/usr/lib/libc.so.66.2`	`/usr/X11R6/lib/libGL.so.10.0`
NetBSD 6.1	`/usr/lib/libc.so`	`/usr/X11R7/lib/libGL.so`
DragonFly 3.4	`/usr/lib/libc.so`	`/usr/pkg/lib/libGL.so`
cross-platform	`dynfind(c("MSVCRT","c","c.so.6"))`	`dynfind(c("OPENGL32","OpenGL","GL","GL.so.1"))`

Platforms	Shared library `expat`	Shared library `SDL`
Windows	`<custom>\libexpat.DLL`	`<custom>\SDL.DLL`
Mac OS X 10.7	`/usr/lib/libexpat.dylib`	`/Library/Frameworks/SDL.framework/SDL`
Solaris 10	`/opt/csw/lib/libexpat.so.1`	`/opt/csw/lib/libSDL-1.2.so.0`
ArchLinux 2013.5	`/usr/lib/libexpat.so`	`/usr/lib/libSDL.so`
Debian 6	`/usr/lib/libexpat.so.1`	`/usr/lib/libSDL-1.2.so.0`
Ubuntu 12.04	`/lib/i386-linux-gnu/libexpat.so.1`	`/usr/lib/i386-linux-gnu/libSDL-1.2.so.0`
FreeBSD 9.1	`/usr/local/lib/libexpat.so`	`/usr/local/lib/libSDL.so`
OpenBSD 5.3	`/usr/lib/libexpat.so.9.0`	`/usr/local/lib/libSDL.so.8.0`
NetBSD 6.1	`/usr/lib/libexpat.so`	`/usr/pkg/lib/libSDL.so`
DragonFly 3.4	`/usr/local/libexpat.so`	`/usr/pkg/lib/libSDL.so`
cross-platform	`dynfind(c("expat", "expat.so.1"))`	`dynfind(c("SDL","SDL-1.2.so.0"))`

Table 4.3: Overview of platform-specific file paths for specific C libraries and cross-platform loading via `dynfind` and short library names.

family or Linux distribution, package management system or custom installation location, while the latter contains a library-specific ⟨*name*⟩ and uses an ABI platform-specific file name or path pattern as shown in Table 4.4.

Operating system(s)	ABI platform	Shared library name scheme
Windows	PE	⟨*name*⟩`.DLL`
Linux, BSDs, Solaris	ELF	`lib`⟨*name*⟩`.so` [`.`⟨*major*⟩ [`.`⟨*minor*⟩]]
Mac OS X	Mach-O	`lib`⟨*name*⟩`.dylib` ⟨*name*⟩`.framework/`⟨*name*⟩

Table 4.4: Naming schemes of shared libraries across operating systems and ABI platforms.

Windows platforms use the binary image format standard PE (Portable Execution Format); a shared library is stored as a *DLL (Dynamically Linked Library)* with the file name extension "`.DLL`".

Unix-related platforms based on the System V ABI standard, such as Linux, BSDs and Solaris, typically incorporate *ELF (Execution and Link Format)* as binary format for executable files and shared libraries; the latter are stored as *Shared Objects* with a common file name prefix "`lib`" and the shared object extension "`.so`". ELF also supports versioning of libraries in which the file name is suffixed by a '`.`' and a ⟨*major*⟩ version number.

Although Mac OS X is very similar to Unix-based systems, it is based on the *Darwin* operating system, which uses a *Mach*-based Microkernel, named 'XNU', and the Mach-O (Mach object file format) binary image format. Shared libraries are represented as *Dynamic Libraries* with the file name prefix "`lib`" and the file extension "`.dylib`". In addition, Mac OS X also provides a second form of shared library components named *Frameworks.* A Mac OS X framework is a directory tree of files with a specific directory name of the form "⟨*name*⟩`.framework`" in which a dynamic library is placed as "⟨*name*⟩" without prefixes or suffixes.

Whether the binary files are named *Dynamically Linked Libraries* on Windows PE, *Shared Objects* on System V ELF platforms, *Dynamic Libraries* on Mach-O or *Framework Bundles* on Mac OS X, all these terms denote a *Shared Library* that can be loaded dynamically at run time via the C API of the platform's dynamic linker.

4.2.4 Searching of Shared Libraries

Each dynamic linker provides a C API, discussed in Section 5.6, for loading shared libraries via a file name at run time. The interface is simple; a character string needs to be given for naming the library. If the file name is given without an absolute path, the dynamic linker *searches* for the shared library. There are fine differences between platforms in naming of shared libraries and which directory locations are considered for searching. Furthermore, the arrangement of user-defined locations for third-party shared library installations also differs depending on the functioning of the dynamic linker. We therefore briefly describe the different search methods and naming schemes in more detail.

4.2.4.1 Dynamic Linker on Windows PE-based platforms

The dynamic linker of Windows searches for a specific DLL in several locations using the following order of directory locations:

- The directory of the application's executable,
- the current working directory,
- a number of system directories and
- a number of locations specified by the environment variable `%PATH%`, which consists of a list of absolute directory paths separated by '`;`' characters.

The file extension "`.DLL`" can be omitted when searching via the C API. DLLs of system components, such as `OpenGL`, are installed in a subfolder, named '`SYSTEM32`', of the Windows folder, which is often located at `C:\WINDOWS` but can be specified during installation of Windows; the exact location is given by the environment variable `%WINDIR%`.

Since Windows platforms lack a standard package management and distribution system there are no standard directory locations for third-party libraries in contrast to open-source platforms and Unix-related systems. Therefore users usually have to copy DLLs into the `SYSTEM32` directory or arrange for a directory for common shared libraries that needs to be added to the environment variable `%PATH%` in order to enable the dynamic linker to find the DLL by its name.

4.2.4.2 Dynamic Linker on ELF platforms

The dynamic linker of ELF systems searches for shared objects given by the full file name using the following pattern:

`lib`⟨*name*⟩`.so`

So the prefix "`lib`" and the file extension suffix "`.so`" need to be included when searching for a shared library ⟨*name*⟩ via the C API.

The dynamic linker searches in

- built-in system locations (e.g. "`/lib`" and "`/usr/lib`"),
- further system locations, which are specified in configuration files, such as defined in "`/etc/ld.so.conf`" and in files located in `/etc/ld.so.conf.d`, and/or configured via system tools such as `ldconfig`,
- directories that are specified by the environment variable `$LD_LIBRARY_PATH`, which comprises a list of directory locations separated by the '`:`' character, and
- directories that are specified within the application's executable for which the dynamic linker searches shared libraries given as a data field named `rpath`.

A number of dynamic linkers that support the ELF format make use of *versioning* of shared libraries. The latter are installed using a multi-level version numbering scheme for their file name, and shortened variants are also installed via symbolic file links. For example, the following pattern of symbolic linked files is used on several Linux platforms:

`lib`⟨*name*⟩`.so.`⟨*major*⟩ → `lib`⟨*name*⟩`.so.`⟨*major*⟩`.`⟨*minor*⟩
`lib`⟨*name*⟩`.so.`⟨*major*⟩`.`⟨*minor*⟩ → `lib`⟨*name*⟩`.so.`⟨*major*⟩`.`⟨*minor*⟩`.`⟨*micro*⟩

The version naming scheme with a single ⟨*major*⟩ version number has a special meaning in ELF; it is the soname (Shared Object Name), which is also often stored within the binary file of the shared library and also within dependent executable files as symbolic link information; it gives the library name, which includes a binary ABI version. This name is used by the dynamic linker for resolving dependencies of application executables to shared libraries during the loading of a dynamically linked application program. The file name scheme is as follows:

ELF System	Package name	Files and Symbolic Links		Name for loading
Solaris 10	`CSWlibexpat1`	`libexpat.so.1`	→`libexpat.so.1.6.0`	`libexpat.so.1`
		`libexpat.so.1.6.0`		
	`CSWlibexpat-dev`	`libexpat.so`	→`libexpat.so.1.6.0`	`libexpat.so`
Debian Linux 7	`libexpat1`	`libexpat.so.1`	→`libexpat.so.1.6.0`	`libexpat.so.1`
		`libexpat.so.1.6.0`		
	`libexpat1-dev`	`libexpat.so`	→`libexpat.so.1.6.0`	`libexpat.so`
Fedora Linux 18	`expat`	`libexapt.so.1`	→`libexpat.so.1.6.0`	`libexpat.so.1`
		`libexpat.so.1.6.0`		
	`expat-devel`	`libexpat.so`	→`libexpat.so.1.6.0`	`libexpat.so`
ArchLinux 2013.5	`expat`	`libexpat.so`	→`libexpat.so.1.6.0`	
		`libexpat.so.1`	→`libexpat.so.1.6.0`	
		`libexpat.so.1.6.0`		
NetBSD 6.1	`expat`	`libexpat.so`	→`libexpat.so.2.0`	
		`libexpat.so.2`	→`libexpat.so.2.0`	
		`libexpat.so.2.0`		`libexpat.so`
FreeBSD 9.1	`expat-2.0.1_2`	`libexpat.so`	→`libexpat.so.6`	
		`libexpat.so.6`		
DragonFly BSD 3.4	`expat`	`libexpat.so`	→`libexpat.so.6`	
		`libexpat.so.6`		
OpenBSD 5.3	`base53.tgz`	`libexpat.so.10.0`		

Table 4.5: Loading `expat` shared library on different ELF-based operating systems.

$$\texttt{lib}\langle \textit{name}\rangle\texttt{.so.}\langle \textit{major}\rangle$$

The ⟨*major*⟩ version number usually starts with zero; it is not incremented until the interface of a library significantly changes or needs to break binary compatibility with a previous version. Whereas the soname "`lib`⟨*name*⟩`.so.`⟨*major*⟩" is used during load- and run-time dynamic linking, the linker of the compiler, which prepares symbolic links between application executables and shared libraries, usually uses the unversioned variant "`lib`⟨*name*⟩`.so`" for searching the shared library and then reads the actual soname from the found shared library as symbolic link information within the application's executable file.

As a consequence, run-time packages of shared libraries often do not install the unversioned file names since applications only require the existance of soname-based library file names. Table 4.5 gives an overview of different packages and ELF shared library names for the `expat` XML parser C library across ELF platforms. From this table we can see that the shared libraries on Solaris and on several Linux distributions are split into a run-time and a development package. The unversioned ELF shared library name of a symbolic file link is usually installed with the development package. Furthermore, we can see that the soname is not standardized across ELF platforms.

There are further fine differences among dynamic linkers on ELF platforms that we briefly describe in the following paragraph.

The GNU dynamic linker on Linux platforms comprises a cache of library names that can be configured via the `ldconfig` utility and the `/etc/ld.so.conf` configuration file, which contains additional search paths for shared libraries. In general the latter comprises the text line instruction "`include /etc/ld.so.conf.d/*`", which causes `ldconfig` to read further library search paths from additional text files in the directory in `/etc/ld.so.conf.d`. For example, the R installation on Fedora Linux 18 on `x86-64` installs the file `/etc/ld.so.conf.d/R-x86_64.conf`, which contributes the search path `/usr/lib64/R/lib` for searching custom R shared libraries such as the `R` shared library and R-specific BLAS libraries. `ldconfig` is usually executed during system startup and after installation of new packages in order to rebuild the library name cache of available shared libraries. When `ldconfig` is executed it scans directories (specified by the configuration) for shared library binary files, which usually have the form "`lib`⟨*name*⟩`.so.`⟨*major*⟩`.`⟨*minor*⟩" and creates symbolic file links of the soname form "`lib`⟨*name*⟩`.so.`⟨*major*⟩". BSD platforms and Solaris also offer a similar utility to configure a cache of shared library names (it is also named `ldconfig` on BSDs and `crle` on Solaris). But in contrast to the GNU-based utility, no symbolic links are created.

The example, shown in Table 4.5, illustrates different schemes for multi-level versioning of file names among Linux distributions. While FreeBSD and DragonFly BSD use a single symbolic link of the form "`lib`⟨*name*⟩`.so`" → "`lib`⟨*name*⟩`.so.`⟨*major*⟩", NetBSD uses a two-level symbol link. NetBSD also discourages the use of the `ldconfig` utility and configuration via `/etc/ld.so.conf` in favour of the `rpath` facility. `rpath` is a data field, similar to soname, which specifies a run-time search path that is hard-coded into an executable; it is used by the dynamic linker when searching for required shared libraries. See NetBSD (2013) for a detailed discussion on this topic.

OpenBSD does not use ELF standard methods for handling sonames. It has its own versioning system and search method. Every shared object is versioned using the file name pattern:

"`lib`⟨*name*⟩`.so.`⟨*major*⟩`.`⟨*minor*⟩"

Interestingly, there are no symbolic file links. The dynamic linker searches corresponding file names via unversioned library names so that "`lib`⟨*name*⟩`.so`" can be used to search a corresponding shared library *without* the use of symbolic file links.

In summary: Whether an unversioned file name is sufficient to load a shared library depends on the operating system; usually soname-based files are always available. However, the soname differs across operating-system platforms; in order to load a shared library dynamically a common naming scheme would be beneficial here.

Our experiments with several shared libraries on various platforms using the `rdyncall` package suggests that standard installations of shared libraries on the BSD-based operating-system family is supplied with a symbolic link to an unversioned file name, so that the short library name "⟨*name*⟩" is sufficient, as it is translated by `dynfind` to "`lib`⟨*name*⟩`.so`". On Linux and Solaris, there are cases where run-time packages of shared libraries do not include a symbolic link of the form "`lib`⟨*name*⟩`.so`" so that the soname needs to be used as a short library name, encoded without the prefix "`lib`" using the pattern "⟨*name*⟩`.so.`⟨*major*⟩". If the sonames differ among Linux and Solaris platforms, several variants need to be passed to `dynfind`. Note that major versions are usually very stable for a particular library and are generally incremented only in major revisions of a library.

See Levine (2000, Ch. 10) for details on ELF and soname, Hearn (2004) for details on versioning of ELF shared libraries, Wheeler (2003, Sec. 3) for details on ELF shared libraries on Linux, and OpenBSD (2013) for notes on handling shared libraries on OpenBSD.

4.2.4.3 Dynamic Linker on Mach-O / Mac OS X

The dynamic linker of the Mach-O platform offers an ELF-compatible C API for dynamic loading of a shared library. It requires the following file pattern to search for a corresponding shared library:

`lib`⟨*name*⟩`.dylib`

The dynamic linker searches for the shared library using a set of environment variables and the current working directory of the running process. The environment variables, if set, contain a sequence of directory paths separated by '`:`' characters. The order is as follows:

- `LD_LIBRARY_PATH`,
- `DYLD_LIBRARY_PATH`,
- working directory and
- `DYLD_FALLBACK_LIBRARY_PATH`, which defaults to `$HOME/lib;/usr/local/lib;/usr/lib`.

Similarly to Windows and in contrast to ELF-based platforms, there are no configuration files to control the searching.

Shared libraries, installed as Mac OS X frameworks, are searched by using the following file name pattern:

⟨*name*⟩`.framework/`⟨*name*⟩

Several standard directory locations are considered such as `/System/Library/Frameworks`, `/Library/Frameworks` and `$HOME/Library/Frameworks`. Further framework directories are considered as specified by the environment variable `$DYLD_FRAMEWORK_PATH` where directory paths are separated by '`:`' characters.

For shared libraries that are installed by package management systems, such as MacPorts, Fink or Homebrew, the environment variables need to be set appropriately in advance. For example, MacPorts installs dynamic libraries under the directory tree `/opt/local/lib`, which need to be adopted by setting the location in the environment variable `LD_LIBRARY_PATH`.

For information on the implementation of shared libraries within operation systems, see Tanenbaum (2009, Sec. 3.5.6).

4.2.5 Implementation

The initial version of `dynfind` incorporates platform-specific search heuristics where the file path is partitioned into four components:

⟨*location*⟩ ⟨*prefix*⟩ ⟨*name*⟩ ⟨*suffix*⟩

The components ⟨*location*⟩, ⟨*prefix*⟩ and ⟨*suffix*⟩ are platform-specific values, while the ⟨*name*⟩ component is passed by the user as a short library name.

For a given list of short library ⟨*name*⟩s, a large number of possible absolute file paths is derived by permutation of the four components. Each trial is passed *directly* to the dynamic linker until a library is successfully opened. The set of ⟨*location*⟩s comprises a hard-coded list of values that is extended by additional paths that are read from environment variables such as `$LD_LIBRARY_PATH` (ELF and Mach-O) and `%PATH%` (Windows PE). In addition, platform-specific characteristics, such as searching for frameworks on Mac OS X, were also incorporated. The initial version of `dynfind` works fine for a large number of platforms but has its draw-backs as new versions of operating systems and software distributions need to be adapted.

Recently an improved version of `dynfind` was developed. It makes use of the underlying search methods of the dynamic linkers and does not use any absolute file paths. Thus it significantly reduces the number of search trials and works more closely with the search method of the dynamic linker. We give the implementation, written in R, in Listing 9. For each short library name, passed to `dynfind`, there are up to two function calls to `.dynload`; the latter is a direct R interface to the dynamic linker for the lookup. Note that the ⟨*location*⟩ naming component is omitted in this scheme since users and system administrators can configure the dynamic linker appropriately via system configuration settings and environment variables.

The derivation of file names from short library names is platform-specific using a maximum of two calls for searching per short library name:

- On Windows the pattern "`lib`⟨*name*⟩" is used to search for "lib" prefixed DLLs and then the pattern "⟨*name*⟩" to search for DLLs without prefix.
- On Mac OS X the pattern "⟨*name*⟩.framework/⟨*name*⟩" is used to search for a framework and the pattern "`lib`⟨*name*⟩`.dylib`" is used to search for a dynamic library.
- On ELF platforms the pattern "`lib`⟨*name*⟩`.so`" is used to search for an unversioned shared library and the pattern "`lib`⟨*name*⟩" is used to search for a soname-based shared library where the short name ends with "`.so.`⟨*major*⟩".

```
  dynfind1 <-
if (.Platform$OS.type == "windows") {
  function(libnames, ...) {
    handle <- .dynload(paste("lib",name,sep=""),...)
    if (!is.null(handle)) return(handle)
    .dynload(name,...)
  }
} else {
  if ( Sys.info()[["sysname"]] == "Darwin" ) {
    function(name, ...) {
      handle <- .dynload(paste(name,".framework/",name,sep=""),...)
      if (!is.null(handle)) return(handle)
      .dynload(paste("lib",name,".dylib",sep=""),...)
    }
  } else {
    function(name, ...) {
      handle <- .dynload(paste("lib",name,".so",sep=""),...)
      if (!is.null(handle)) return(handle)
      .dynload(paste("lib",name,sep=""),...)
    }
  }
}

dynfind <- function(libnames, auto.unload = TRUE) {
  for (libname in libnames) {
    handle <- dynfind1(libname)
    if (!is.null(handle)) return(handle)
  }
}
```

Listing 9: Implementation of `dynfind`.

4.2.6 Low-level interface

Since the existing interface `dyn.load` is inappropriate to make use of the search capabilities of the dynamic linker, an alternative R interface to the dynamic linker was incorporated into the `rdyncall` package. The following table gives a side-by-side comparision between the `base` and the `rdyncall` interface to the dynamic linker:

Task	`base` package	`rdyncall` package
Load	`dyn.load`	`.dynload`
Resolve	`getNativeSymbolInfo`	`.dynsym`
Unload	`dyn.unload`	`.dynunload`

Load and *unload* tasks are for opening and closing a shared library. Access to the symbol table for retrieving pointers to functions and global variables is given via *resolving* symbolic names to addresses.

4.2.6.1 Improved Life-Cycle Management

Since shared libraries are resources that can be loaded at run time, they should also be unloaded dynamically when they are no longer needed. The `base` R interface to the dynamic linker is mainly used to manage loading and unloading of shared libraries as part of the R package loading and unloading mechanism. Since the `rdyncall` interface is designed for working with *external* C libraries further considerations went into the implementation of the `.dynload` and `.dynsym` functions to offer a fine-grained resource management.

Resource objects in R are often represented as external pointers, which are copied *by-reference* so that there exists a one-to-one mapping between an R external pointer object and corresponding resource. The garbage collector of R offers the ability to register a *finalizer* function for external pointers that is called when the external pointer is about to be removed. This enables the implementation of methods for cleanup of resources such as closing files or unloading shared libraries. Aside from storing an address, external pointers may also contain an *owning* reference to another object in order to *protect* the latter from garbage collection.

The `rdyncall`-based interface to the dynamic linker incorporates *finalizers* and *protection* in order to offer a fine-grained life-cycle model for loading/unloading shared libraries. Shared libraries that are opened via `.dynload` are returned as *handle* objects represented as external pointer R object. `.dynload` has an optional parameter, named `auto.unload`; if set to `TRUE` (the default) a finalizer is registered for the external object; then unloads the shared library when the object is about to be removed by the garbage collector. The handle to a shared library is passed to `.dynsym` for resolving symbols; the addresses of resolved symbolic functions are returned as external pointers that *protect* the library's handle object from garbage collection. So shared libraries, opened via `.dynload`, are not removed as long as there exist any active external pointer R objects that reference their code or data objects.

The implementation of the low-level interface to the dynamic linker of the operating system is based on the `dynload` library that provides a thin portable abstraction layer to the platform-specific C API of the dynamic linker; see Section 5.6 for details.

4.3 Foreign Function Calls

The facility to make calls to foreign functions of external precompiled libraries is probably the most obvious component of an FFI. We gave an overview of different types of FFIs in Section 3.4; we noted that the `.C` FFI of the `base` package is a *static* version of an FFI; it supports a small subset of C function types that can be called directly from R without additional C code wrapper. Dynamic versions of FFIs offer interoperability with a wide range of foreign function types by using a generic dynamic implementation and type information about the target function type.

4.3.1 Interface

The function `.dyncall` constitutes the main interface for making calls to foreign C functions. The interface is as follows:

```
.dyncall(address, signature, ..., callmode = "default")
```

`.dyncall` requires two items of information about the C function:

- the memory location of the function's code, which is specified by `address`,
- and the function type, which is specified by `signature`.

The actual argument values for marshalling to C arguments are passed via `...` according to their positional order in the foreign C function type.

The `callmode` is an optional parameter. It specifies the calling convention of the target C function and is of particular interest on the Microsoft Windows 32-bit `x86-32` platform to make function calls to System DLLs. We give details on the selection of calling conventions in Section 4.3.4.

The `address` is passed as an external pointer; a corresponding object for refering to a "loaded C function" can be obtained by using `dynfind` and `.dynsym`, as discussed in Section 4.2, or by using the standard R `base` functions for loading shared libraries and resolving of symbols to addresses, such as `dyn.load` and `getNativeSymbolInfo`, respectively.

The parameter `signature` specifies the *C function type* of the target C function encoded as a *call signature* `character` string. The inclusion of this parameter constitutes a main difference to the FFI of the `base` package.

4.3.2 Call Signature

The call signature specifies the C function type. The encoding was specified in the syntax diagram illustrated in Section 3.7.2.1; we give the text pattern of the format again:

$$[\ \langle \textit{parameter-type}\rangle\ldots\ ]\ \text{‘}\texttt{)}\text{’}\ \langle \textit{return-type}\rangle$$

The parameter types are specified according to the C function prototyp declaration, as defined in C header files or sources, *from left to right*. ⟨*parameter-type*⟩ and ⟨*return-type*⟩ are specified as *object signatures* that refer to a particular C parameter- and return data type. Table 4.6 gives the mapping between supported basic C types and corresponding object type signature encoding.

Example In Section 3.4.2 we discussed the limitation of R's static FFI using a simple example of binding the C `double sqrt(double x)` function of the `math` library to R. We showed that development of C wrapper code and compilation was needed for connecting R with the C function `sqrt`.

`.dyncall` can be used to make a *direct* call to `sqrt` from within R without the need for adapters in C. In the following listing we first give the steps to load the `sqrt` C function, which works across R platforms using `dynfind` and `.dynsym` as described in Section 4.2, followed by an FFI call via `.dyncall`.

C data type	Object signature
`char`	‘c’
`short`	‘s’
`int`	‘i’
`long`	‘j’
`long long`	‘l’
`float`	‘f’
`double`	‘d’
`unsigned char`	‘C’
`unsigned short`	‘S’
`unsigned int`	‘I’
`unsigned long`	‘J’
`unsigned long long`	‘L’
`void`	‘v’
`void*`	‘p’
`const char*`	‘Z’
`Bool` , `_Bool`	‘B’
`typedef` ... *name*	‘<’ name ‘>’
TYPE `*`	‘*’ ...

Table 4.6: Mapping between C data types and object signature characters.

```
> library(rdyncall)
> lib <- dynfind(c("msvcrt","m","m.so.6"))
> addr <- .dynsym(lib,"sqrt")
> .dyncall(addr,"d)d", 144)
[1] 12
```

As indicated by the return value `12`, we can assume the function call was successful. Note the call signature “`d)d`” passed as second argument, which specifies the C function type of `sqrt`. The signature is derived from the C function declaration `double sqrt(double)`.

4.3.3 Supported Function Types

`.dyncall` offers support for making calls to a large subset of function types. Functions where all parameter types and the return type are in the following set can be called directly:

- All standard signed integer types from `signed char` to `long long int`.
- All standard unsigned integer types from `unsigned char` to `unsigned long long int`.
- Floating-point types `float` and `double`.
- Code and data pointers.
- The boolean type `_Bool`.

No support is currently provided for functions where at least one parameter type, or the return type, is in the following set:

- Floating-point type `long double`.
- Composite `struct` and `union` data types passed as *call-by-value* objects.
- Any other standard and platform-specific C data types (see Table 3.4 for an overview).

The maximum number of arguments that is supported by `.dyncall` varies; but the number is very large. The total size of memory that C argument objects of a function call require must be less than 4096 bytes; this leads to a limitation of approximately 512-1024 arguments per function call depending on the actual argument types. This is effectively no limitation since, in practice, C API functions comprise a much smaller number of parameters.

The implementation of `.dyncall` is based on the Generic Dynamic FFI `dyncall` which provides a portable C interface for dynamic invocation of function calls. It is partly implemented in assembly language with ports to a large range of platforms; all major R platforms are supported. (See Table 5.11) Note that some of the platforms listed in the table may not be supported by R. The implementation of `.dyncall` by means of the `dyncall` library is discussed in Section 5.4.6.2.

4.3.4 Supported Calling Conventions

`.dyncall` supports several calling conventions as selected by the `callmode` parameter. This parameter may be needed currently on the Microsoft Windows 32-bit platform for the `x86-32` processor architecture where C functions of system libraries use a different calling sequence to the default C calling convention for C libraries.

`.dyncall` is a front-end function that uses the `callmode` argument to delegate to specialized back-end functions that implement a particular calling convention:

High-Level Interface:
```
.dyncall ( address, signature, callmode = "default", ... )
```
Low-Level Interface:
```
.dyncall.<callmode> ( address, signature, ... )
```

The calling convention is selected by a named character string passed via the `callmode` argument. Table 4.7 gives a list of possible values.

`.dyncall` also supports virtual method calls to C++ member functions, named *thiscalls*. Usually the `this` pointer of an instance to a C++ class has to be passed as first argument. For the `x86-32` architecture, the calling convention also needs to be further specified, depending on the C++ compiler that was used for the target code. The implementation of `.dyncall` currently supports the Microsoft Visual C++ and GNU C++ compiler for Microsoft Windows (MinGW32).

`callmode`	Description
`default`	Default calling convention for C functions
`cdecl`	Standard C functions (alias to `default`)
`stdcall`	C `stdcall` functions of Windows 32-bit System libraries
`thiscall`	Calling convention for C++ member functions
`thiscall.msvc`	C++ member functions compiled by `Visual C++`
`thiscall.gcc`	C++ member functions compiled by `MinGW32`
`fastcall`	Calling convention for C `fastcall` functions
`fastcall.msvc`	C `fastcall` functions compiled by `Visual C++`
`fastcall.gcc`	C `fastcall` functions compiled by `MinGW32`

Table 4.7: Supported non-standard calling conventions required on Windows 32-bit platforms.

4.3.5 Type Checking

If argument values of a call do not match the corresponding parameter types of the target function, the execution of the call can lead to a fatal run-time error. For this reason C compilers perform type analysis of caller C code before code generation, and abort compilation if an unresolvable argument type mismatch is detected. FFIs represent run-time services for making function calls and, likewise, they need type information to guarantee system stability.

During an FFI call, the `.C` function uses the actual types of R arguments to derive the C function type of the call and which is determined dynamically at call time. When C functions are registered as part of the initialization of an R package that contains compiled code, type information about the function's parameter types can be specified. (See R Development Core Team, 2012c, Section 5.4 for details.) This information is later used during a call via `.C` for type checking. However, when C functions of an *external* library are called via `.C`, there is no source of information available that provides type information for type checking.

The call signature parameter provides type information about the target C function that `.dyncall` uses for three tasks:

- *Type Checking*: While the arguments are processed, several type checks are performed: the number of actual arguments and formal parameters need to be equal and the argument values need to be convertable to C parameter types.
- *Value Conversion*: If arguments are not of the same type `.dyncall` checks if the value can be converted automatically.
- *Directing*: The argument objects in C data representation are passed to the Generic Dynamic FFI, which copies the sequence of arguments to CPU registers and on the C call stack, followed by the execution of an ABI-conformant function call. This procedure is mainly driven by the call signature.

In the following, we consider *type checking* and *value conversion*; *directing* the function call at low-level is discussed in detail in Section 5.4, and the implementation of `.dyncall`, written in C, for all three tasks is discussed in Section 5.4.6.2.

Before making the foreign function call, the `.dyncall` FFI performs a number of tests on the passed arguments, based on the `signature` in order to check whether the number of arguments and type of the arguments are compatible or can be meaningfully and easily converted to compatible arguments.

A function call is rejected if the passed number of arguments does not match with the expected number of parameters as specified by the function type signature:

```
> .dyncall(addr, "d)d")
Error in .dyncall(addr, "d)d") :
  Not enough arguments for function-call signature 'd)d'.
> .dyncall(addr, "d)d", 144, 13)
Error in .dyncall(addr, "d)d", 144, 13) :
  Too many arguments for signature 'd)d'.
```

The call is also rejected if the type of an argument is incompatible with parameter type, the call is also rejected. For example, when a `character` string is passed to `sqrt` via `.dyncall`, this is rejected as follows:

```
> .dyncall(addr, "d)d", "some-text")
Error in .dyncall(addr, "d)d", "some-text") :
  Argument type mismatch at position 1: expected C double convertable value
```

Numeric objects in R are represented as vectors that can have length zero. Calls are reject if an argument to C is an empty vector.

```
> .dyncall(addr, "d)d", integer(0))
Error in .dyncall(addr, "d)d", ...) :
  Argument type mismatch at position 1: expected length greater
  zero.
```

Furthermore, automatic value conversion is performed if there is a conversion that is feasible without creating new R memory objects. For example, while the natural counter-part to a `double` C argument is given by an R `double` argument, the user may pass an `integer`, `logical` or even `raw`, such as the literal integer value `144L` suffixed by the `L` that indicates an `integer` value:

```
> .dyncall(addr, "d)d", 144L)
[1] 12
```

In summary, the call signature "`d)d`" leads to the following type checks and conversion rules:

- The number of arguments, represented by `...`, must be 1.
- The type of the argument must be a `logical`, `double`, `integer` or `raw` atomic vector.
- The length of the vector needs to be at least 1.
- If the argument is an R `double` vector the first element is passed as argument to the C function,
- otherwise the value needs to be converted: The first scalar element is extracted from the atomic

vector and converted to a C `double` object.

4.3.6 Value Conversion

Several components of the **rdyncall** package, namely `.dyncall`, `.pack`, `.unpack` and `new.callback`, incorporate data conversion and mapping strategies between data objects of R and C. The following table gives an overview of the different facilities and the context in which marshalling/unmarshalling of R language objects to/from C data objects takes place:

Marshalling (R $\rightarrow$ C)	Unmarshalling (C $\rightarrow$ R)
`.dyncall` arguments	`.dyncall` return values
`.pack` argument	`.unpack` return value
Callback return values	Callback arguments from C

Since C functions and objects have a static type, whereas the type of R language objects is determined at run-time, we consider differentiated mappings depending on the direction of data conversion. The mapping between R and C types was decided on the internal data representation of R object types. A large number of mappings was built-in for C scalar arithmetic and untyped pointer types as illustrated in Table 4.8, which gives mappings for both directions. Mappings of "R $\rightarrow$ C" are given in columns one and two, and "C $\rightarrow$ R" in columns two and three.

R input type							C type		R output type
`logical`	`integer`	`double`	`raw`	`character`	`externalptr`	`NULL`	Signature		
✓	✓	✓	✓	-	-	-	'B'	`_Bool`	`logical`
✓	✓	✓	✓	-	-	-	'c'	`char`	`integer`
✓	✓	✓	✓	-	-	-	'C'	`unsigned char`	`integer`
✓	✓	✓	✓	-	-	-	's'	`short`	`integer`
✓	✓	✓	✓	-	-	-	'S'	`unsigned short`	`integer`
✓	✓	✓	✓	-	-	-	'i'	`int`	`integer`
✓	✓	✓	✓	-	-	-	'I'	`unsigned int`	`double`
✓	✓	✓	✓	-	-	-	'j'	`long`	`double`
✓	✓	✓	✓	-	-	-	'J'	`unsigned long`	`double`
✓	✓	✓	✓	-	-	-	'l'	`long long`	`double`
✓	✓	✓	✓	-	-	-	'L'	`unsigned long long`	`double`
✓	✓	✓	✓	-	-	-	'f'	`float`	`double`
✓	✓	✓	✓	-	-	-	'd'	`double`	`double`
✓	✓	✓	✓	✓	✓	✓	'p'	*any-type* `*`	`externalptr`
-	-	-	-	✓	✓	✓	'Z'	`char*`	`character`

Table 4.8: Value Conversion between R and C arithmetic and pointer types.

In the direction "R → C" a *dynamic* mapping was incorporated where different kinds of R object types can be passed as argument for a particular C data type that is automatically converted. This provides for a convenient interface which makes coercion of input R argument types to `.dyncall` and `.pack` unnecessary. For example, the literal R value "`123`" can be passed as value for C `int` data types without the need to convert the argument via `as.integer`. Furthermore, dynamic mappings are required for scalar C data types where there exists no counterpart as data representation in R. For example, R objects can be passed for the target C scalar data type `float` even though there is no R atomic vector type that uses this representation for its element type.

Seven R object types are considered as valid R source objects for value conversion to C data objects:

- `logical`, `integer`, `double` and `raw` vectors can be passed for any supported *C scalar arithmetic* type where the first element of the vector is considered as source for value conversion. Vectors can also be passed for C untyped pointers in that the address of the first element of the vector is passed as pointer value.
- `character` vectors can be passed for C untyped or string pointers in that the address of the first character of the first string element of the vector is passed as pointer value.
- `externalptr` R objects can be passed for any C pointer object.
- `NULL` can be passed for any C pointer type to denote a C `NULL` pointer value.

R `complex` vectors are omitted since C `_Complex` data types are currently not supported by the underlying Generic Dynamic FFI. All atomic vectors need to have a length greater than 0.

In the direction "C → R" a *static* mapping is used due to the static source type. For each supported C source type a specific R target type is used that should preserve the C value although it may use a different representation. This works fine for most C data types but except for 64-bit integer types because R does not offer a native 64-bit integer data type. Currently C (unsigned) `long long` data objects are cast to C `double` and then passed as R `double` vectors.

4.3.6.1 Support for typed C Pointers

C pointers need to be handled with caution as outlined in Section 3.6.4. In contrast to C arithmetic scalar objects which are passed *by-value* and which can be easily converted from/to R without allocation of array objects, C pointers reference memory objects, which can be manipulated by a foreign function as a side effect. Untyped pointers, denoted by '`p`', allow the passing of any atomic vector object which maybe modified by calls to `.dyncall` and `.pack`, *typed pointers*, such as `int*` denoted by '`*i`', further constrain the set of possible input atomic vector objects and slightly improvement of type safety. Table 4.9 gives an overview of possible mappings between R objects and typed C pointers.

The signature '`*v`' is an alias for untyped pointers using the prefix pointer type signature notation. For typed C pointer objects only those R atomic vectors are accepted which comprise a C array data

R input type								C type	R output type
`logical`	`integer`	`double`	`raw`	`character`	`externalptr`	`NULL`	Signature		
✓	✓	✓	✓	✓	✓	✓	‘*v’	`void*`	`externalptr`
?	?	-	✓	-	✓	✓	‘*B’	`_Bool*`	`externalptr`
-	-	-	✓	✓	✓	✓	‘*c’	`char*`	`character`
-	-	-	✓	✓	✓	✓	‘*C’	`unsigned char*`	`externalptr`
-	-	-	✓	-	✓	✓	‘*s’	`short*`	`externalptr`
-	-	-	✓	-	✓	✓	‘*S’	`unsigned short*`	`externalptr`
✓	✓	-	✓	-	✓	✓	‘*i’	`int*`	`externalptr`
✓	✓	-	✓	-	✓	✓	‘*I’	`unsigned int*`	`externalptr`
?	?	-	✓	-	✓	✓	‘*j’	`long*`	`externalptr`
?	?	-	✓	-	✓	✓	‘*J’	`unsigned long*`	`externalptr`
-	-	-	✓	-	✓	✓	‘*l’	`long long*`	`externalptr`
-	-	-	✓	-	✓	✓	‘*L’	`unsigned long long*`	`externalptr`
-	-	-	✓	-	✓	✓	‘*f’	`float*`	`externalptr`
-	-	✓	✓	-	✓	✓	‘*d’	`double*`	`externalptr`

Table 4.9: Passing of typed pointer objects between R and C.

representation that is compatible with the target C pointer type. As an exception, `raw` atomic vectors are always accepted; they are used as generic containers of data. The mapping of `_Bool*` and `long*` is displayed as a “?” as the data representation of the C data type depends on the target platform:

- `Bool_` is usually based on `int` (4 bytes). However, on Mac OS X for the `ppc32` architecture, a smaller sized element type of 1 byte is used for its representation.
- On 64-bit architectures, the `long` is based on `int` (4 bytes) on Windows, but `long long` (8 bytes) on all System V-based ABI platforms (Linux, Mac OS X, Solaris and BSDs). For further discussion of 64-bit platforms see Section 5.3.8 and Table 5.8.

For all the C pointer types where there exists no compatible R atomic vector with a compatible C array, such as `short*`, `long long*` and `float`, only `raw`, `externalptr` and `NULL` can be passed. Pointers to C character data, such as `char*`, `unsigned char*` or `signed char*`, are accepted by R data objects of type `raw` and `character`.

4.4 Wrapper Functions

The *call signature* makes for a *direct* and *flexible* call interface to native compiled functions; it provides automatic value conversion and type-safety. However, `.dyncall` calls require at least two parameters that need to match to ensure type-safety, i.e. the `signature` character string needs to encode a C function type that corresponds with the type of the foreign function that is referenced by the `address` parameter. When making calls to several functions of a C API from within a scripting environment, the proper arrangement of the two parameters for each call can be error prone and burdensome process.

In general wrapper functions are used for providing a call interface to C API functions. As the arguments of function calls are dynamically typed, wrappers need to check and convert the type of each argument according to the target function type. But in contrast to wrappers of the static FFIs of R, which need to implement specific type checks and value conversion in C, `.dyncall` wrappers are lightweight R objects comprising of a single call object with appropriate parameters `address` and `signature`.

In this section we compare writing R wrappers to C API functions using the FFI of `base` and `rdyncall`. We also discuss a convenient automation function, named `dynbind`, for creating bindings to C API functions of a shared library, that works across platforms. We report the results of a performance benchmark between the FFIs of `base` and `rdyncall`.

4.4.1 Comparison of Wrapper Function

In this section we compare the creation of a wrapper functions by using the example of the `sqrt` function of the standard C `math` library. Listing 10 gives the hybrid implementation in R and C for type-safe wrappers using `.C`, `.Call` and `.External`.

Since all wrappers of `sqrt` need to provide a type-safe interface, they need to ensure that

- the number of arguments is 1,
- the vector length of the first argument is ≥ 1,
- the first value of the vector is converted to a C `double` object,
- the call uses the C function type signature `double sqrt (double)`, and
- the return value of `sqrt` is passed as a `double` object.

The R wrapper code of `.C` is used for type-checking and conversion, while the C wrapper is used to implement the actual C function call. `.C` can not be used here directly since it does not have support for passing C return values and has no support for scalar C argument types; arguments are always passed as data pointers.

While the `.C` interface was designed for making *direct* calls to *external* C functions, the `.Call` and `.External` interfaces represent interfaces for *writing R functions in C*, which give access to the internal structure of R arguments and enable the creation of new R objects. In both cases a lightweight R wrapper code is used to delegate to a corresponding C wrapper code. At first the length of the input argument is checked via `LENGTH(arg)`. Then type checking and value conversion is performed via explicit

FFI | R Wrapper | C Wrapper

`.C`

```
sqrt.C <- function(x)
{
  x <- as.double(x)
  length(x) <- min(length(x), 1)
  .C(sqrt_C, x)[[1]]
}
```

```
#include <math.h>
void sqrt_C(double* ptr)
{
  double x, result;
  x = ptr[0];
  result = sqrt(x);
  ptr[0] = result;
}
```

`.Call`

```
sqrt.Call <- function(x)
  .Call(sqrt_Call, x)
```

```
#include <Rinternals.h>
#include <math.h>
SEXP sqrt_Call(SEXP arg)
{
  SEXP nv;
  double x;
  if (LENGTH(arg) == 0)
    Rf_error("length(x) is zero.");
  nv = PROTECT(coerceVector(arg,REALSXP));
  x = REAL(nv)[0];
  x = sqrt(x);
  UNPROTECT(1);
  return ScalarReal(x);
}
```

`.External`

```
sqrt.External <- function(x)
  .External(sqrt_Ext, x)
```

```
#include <Rinternals.h>
#include <math.h>
SEXP sqrt_Ext(SEXP args)
{
  SEXP nv, arg;
  double x;
  args = CDR(args);
  arg  = CAR(args);
  if (LENGTH(arg) == 0)
    Rf_error("length(x) is zero.");
  nv = PROTECT(coerceVector(arg,REALSXP));
  x = REAL(nv)[0];
  x = sqrt(x);
  UNPROTECT(1);
  return ScalarReal(x);
}
```

`.dyncall`

```
sqrt.dyncall  <- function(...)
  .dyncall(sqrt, "d)d", ...)
```

Listing 10: Comparison of FFI Wrapper functions to C `sqrt` library function.

coersion of the R object `arg` to a `double` R object by the C API function `coerceVector(arg,REALSXP)` of R. The operation returns a new temporary object, which needs to be protected from garbage collection for the duration of its use within the C function; this is done by a call to `PROTECT`. The first `double` element is extracted so that it can be passed as argument to a function call to `sqrt`. A corresponding call to `UNPROTECT` is needed to balance the number of `PROTECT`/`UNPROTECT` calls. The return value is placed in a newly created `double` vector via `ScalarReal`.

Wrappers for the `.External` FFI look quite similar to those for `.Call`. The difference of their *R to C* calling convention is due to the method of passing arguments. While `.Call` passes the S-Expression of each argument separately, the `.External` call passes a single S-Expression which gives access to the whole call object, a LISP-like pairlist that contains all arguments in a list and has its target function name as first element. In the case of `.External` the arguments are accessible via traversing of the pairlist using a combination of calls to `CDR` and `CAR` macros.

In contrast to the required C code needed for `.C`, `.Call` or `.External`, wrappers for `.dyncall` are very compact and simply wrap the `.dyncall` call using the correct target address and a matching signature without any C code.

4.4.2 Automation of Wrapper Function Creation

`.dyncall` significantly simplifies the writing of wrapper functions in comparison to the existing FFI facility in R such as `.C`. However, the development of bindings for a complete set of C API functions can be still a complex task; for example, the C API of the `SDL` library comprises more than 200 functions.

The `rdyncall` package includes an automation function, named `dynbind`, that uses the *DynPort* encoding format for *library signatures* and the toolchain that were discussed in Section 3.7 and 3.8, respectively.

4.4.2.1 Interface

The `dynbind` interface is given below (a few advanced parameters were omitted):

```
dynbind(libnames, signature, envir=parent.frame(), callmode="default")
```

`libnames` gives a list of short names which is passed to `dynfind` to load a native library across platforms, as described in Section 4.2.2.

The `signature` represents a sequence of C function signatures, as described in Section 3.7.3.6. The syntax format is as follows:

$$\langle \textit{name} \rangle \text{ `('} \text{ [ } \langle \textit{argument-type} \rangle \ldots \text{ ] } \text{`)'} \langle \textit{return-type} \rangle \text{ `;'} \ldots$$

After the library is located and loaded via `dynfind`, `dynbind` processes the signature. For each C function's name `<NAME>` and call signature `<SIGNATURE>`, the C function's address `<ADDR>` is resolved via `.dynsym` and a corresponding R wrapper function is created using the following template:

```
<NAME> <-function (...) .dyncall.<CALLMODE> ( <ADDR> , <SIGNATURE>, ... )
```

Note the `<CALLMODE>` is specified as an argument to `dynbind`. The wrapper objects are assigned in the R environment given by `envir` with the name of the C function.

Example Consider an R application that uses two C functions of the `SDL` library for the initialization and setup of a graphics output window:

```
int           SDL_Init        (Uint32 flags);
SDL_Surface * SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);
```

The following function call to `dynbind` can be used for the complete task for loading the SDL library and creation of two R wrappers to corresponding C API functions:

```
> dynbind(c("SDL","SDL-1.2","SDL-1.2.so.0"), "
SDL_Init(I)i;
SDL_SetVideoMode(iiiI)*<SDL_Surface>;
")
```

It is assumed that the SDL library is installed on the system; see Section 4.7.3 for details. If the search was successful, two new R functions, namely `SDL_Init` and `SDL_SetVideoMode`, are installed in the global environment:

```
> SDL_Init
function (...)
.dyncall.default(<pointer: 0x55562c2>, "I)i", ...)
> SDL_SetVideoMode
function (...)
.dyncall.default(<pointer: 0x555129e>, "iiiI)*<SDL_Surface>", ...)
```

Note that external pointer is directly placed as an argument to the `.dyncall` call so that no symbolic name lookup is performed. Since the C API is available to R using a similar name, the syntax of the R user code is very similar to that of C for using the SDL C API as Listing 11 illustrates. Note that the symbolic constants need to be specified in R according to the C API headers of SDL. In Section 4.7 we discuss the `dynport` function and *DynPort* files which use `dynbind` for C functions but also provides for mapping of symbolic constants and data types.

```
#include "SDL.h"
int main(int argc, char* argv[]) {
  SDL_Surface * surface;
  SDL_Init(SDL_INIT_VIDEO)
  surface = SDL_SetVideoMode(640,480,32,SDL_OPENGL)
}
```

```
SDL_INIT_VIDEO <- 0x00000020
SDL_OPENGL     <- 0x00000002

SDL_Init(SDL_INIT_VIDEO)
surface <- SDL_SetVideoMode(640,480,32,SDL_OPENGL)
```

Listing 11: C and R: Using SDL for opening an OpenGL window.

4.4.3 Implementation

The function `dynbind` was implemented in R by using text processing and meta programming functions for manipulation of language objects. We give a detailed description of the implementation using a slightly simplified version but that uses the core steps for automation of R wrapper functions from *DynPort* signatures. We give the implementation in successive steps. For each step a brief description is given, followed by the implementation in R and a detailed explanation.

1. The function definition begins as follows:

   ```
   dynbind <- function(libnames, signature, envir=parent.frame()) {
   ```

   `libnames` specifies the C library, `signature` specifies a sequence of function signatures given as a character string and `envir` specifies an environment for storing created wrapper functions.

2. The `libnames` argument is passed to `dynfind` to load the shared library:

```
 lib<-dynfind(libnames)
```

3. The text string `signature` is converted to a list of character vectors with two elements (name and signature):

```
 sigtab <- gsub("[ \n\t]*","",signature)   # remove white spaces and new line
 sigtab <- strsplit(sigtab, ";")[[1]]      # split functions at ';'
 sigtab <- strsplit(sigtab, "\\(")         # split name/call signature at '('
```

White space and new line characters are removed from the signature string, so that all function signatures are separated by a ';'. `gsub` is used for pattern-based text substitution. The first argument specifies the pattern in regular expression notation; `"[ \n\t]*"` denotes character sequences of white space, tabulator codes and new lines. The text is then separated by ';' characters using `strsplit`. The result is returned as a list with a string vector of function signatures. `strsplit` is applied a second time to split each string in the vector by '(' to separate name from call signature. Notice, the character '(' has to be escaped via `"\\("`. As a result, a list of character string vectors with two elements is returned.

4. The lists of names and signatures are processed. In each iteration step the name and signature is stored in the local variables `name` and `signature` for the next wrapper to create.

```
for (i in seq(along=sigtab)) {
  name      <- sigtab[[i]][[1]]
  signature <- sigtab[[i]][[2]]
```

   (a) The C function `name` is resolved to its address using `.dynsym`:

   ```
   address<- .dynsym( lib, name )
   ```

   (b) All parameters for `.dyncall` are now available for the creation of a wrapper function:

   ```
   call <- substitute(
     .dyncall.default(X, Y,...),
     list(
         X = address,
         Y = signature
     )
   )
   ```

   The `call` represents a call object, such as the following call expression:

   ```
   .dyncall.default(<pointer: 0x15711bd0>, "I)i", ...)
   ```

   `substitute` is a powerful function for manipulation of R language objects. It receives two arguments, namely an expression and a list of named elements. Symbols within the expression are substituted by matching list elements. Note that the first argument to `substitute` is not evaluated; it is a template for the returned expression.

   (c) An R function is created and its body is assigned the call object.

```
    wrapper <- function(...) { }
    body(wrapper) <- call
```

First, we create a 'dummy' function in order to apply `body()<-`, which assigns a new body of a function. Note that the formal argument `...` is needed for the wrapper to pass all arguments to the inner call object.

(d) Finally, the wrapper function `wrapper` is assigned to an environment `envir`.

```
    assign( name, wrapper, envir=envir )
```

The `envir` parameter defaults to the environment of the caller of `dynbind` i.e. when `dynbind` is called from the interpreter, the environment returned by `parent.frame()` is the global environment.

This implementation was done purely in R and it emphasises the advantage of a simple text-based encoding format for C type information, which can be utilized in different contexts. In our case, we used the function signature for the creation of wrapper functions; one part of the signature is also used for resolving symbols via `.dynsym` and the other part is incorporated as a *call signature* for `.dyncall`.

4.4.4 Performance Benchmark

In this section we give a summary of a small performance benchmark to compare the timings needed for calling dummy C functions with one, two, four and eight arguments using wrappers implemented by means of `.C`, `.Call`, `External` and `.dyncall`.

The dummy C functions `t1`, `t2`, `t3` and `t4` are defined as follows:

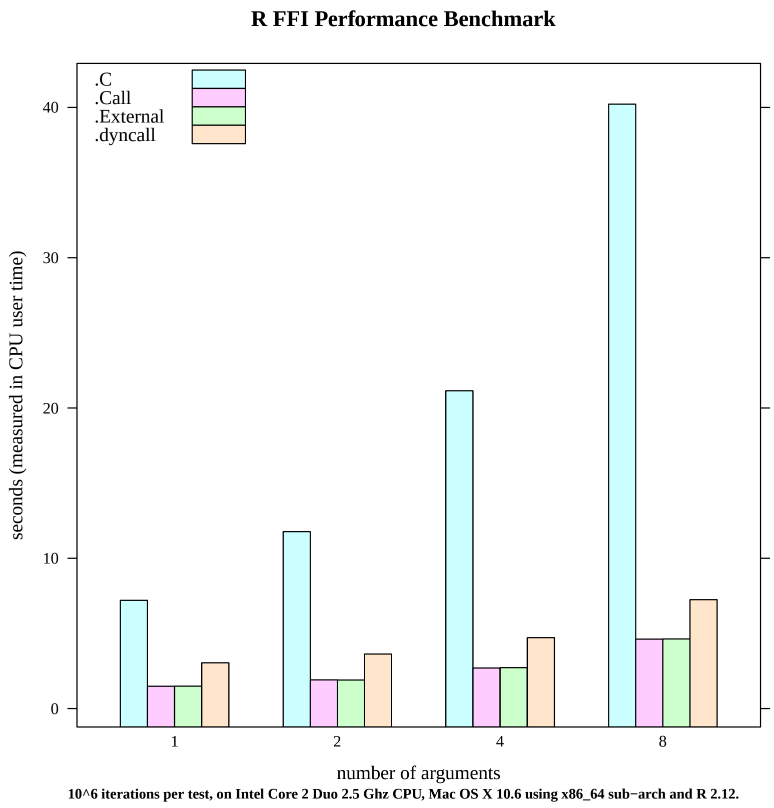
```
void t1(int x) { }
void t2(int x, int y) { }
void t4(int x, int y, int z, int w) { }
void t8(int x, int y, int z, int w, int r, int s, int t, int u) { }
```

For each test the call is repeated 10^6 times in a loop to obtain results on a per second scale.

```
Test2.C        <- function(N) for (i in 1:10^6) { Wrapper2.C(1,2) }
Test2.Call     <- function(N) for (i in 1:10^6) { Wrapper2.Call(1,2) } }
Test2.External <- function(N) for (i in 1:10^6) { Wrapper2.External(1,2) } }
Test2.dyncall  <- function(N) for (i in 1:10^6) { Wrapper2.dyncall(1,2) }}
```

In each test R `numeric` argument values are passed that need to be coerced to an `integer` vector in R or to cast to `int` scalars in C somewhere along the call path to the final target C function.

Our results are illustrated in Figure 4.7 as a bar chart and a table of CPU user time in seconds. These show the low performance of `.C`; we can assume this is due to the use of interpreted R code for type-checking and value conversion. While the performance of `.Call` and `.External` are nearly equal, `.dyncall` takes about twice of CPU user time. However, this overhead shows be assessed in terms of the time-savings for development due to automation of bindings.



Arguments	`.C`	`.Call`	`.External`	`.dyncall`
1	7.20	1.48	1.49	3.05
2	11.77	1.91	1.90	3.63
4	21.15	2.70	2.72	4.72
8	40.22	4.62	4.63	7.25

Figure 4.7: Result of FFI benchmark performance test: Bar chart (*left-hand panel*) and table of CPU user time in seconds for 10^6 function calls.

4.5 Data-level Interoperability

In this section we discuss a component of the `rdyncall` package for handling foreign C `struct` and `union` data objects from within R. The development was motivated by the following example.

Example SDL-based applications run a sequence of top-level tasks in a main loop, such as generating graphics output and processing input events. Listing 12 gives a typical main loop skeleton, written in C and also given in R using the facility of `rdyncall` for handling C `union` and `struct` data types.

During event processing, the `SDL` function

```
int SDL_PollEvent(SDL_Event* event)
```

is called successively in a loop to read events from the event input queue of `SDL`. In each round a data record of event information is written to user memory specified by the pointer argument `event`. The return value of the function indicates whether an event was successfully written (value of 1) or the queue is empty (value of 0) so that the event-processing loop can break. Typically a single C object of type `SDL_Event` is allocated and then reused for each subsequent calls to `SDL_PollEvent`.

Records are further processed via read operations of member fields of the record. The C data type

```c
#include "SDL.h"
#include "SDL_opengl.h"
int main() {
  int quit = 0;   // application state variables
  SDL_Event e;    // allocate event data storage
  // Init SDL:
  SDL_Init(SDL_INIT_VIDEO|SDL_INIT_JOYSTICK)
  SDL_SetVideoMode(640,480,32,SDL_OPENGL)
  SDL_EnableUNICODE(1)
  // Main loop:
  while(!quit) {
    // Update display:
    glClear(GL_COLOR_BUFFER_BIT)
    SDL_GL_SwapBuffers();
    // Process events:
    while(SDL_PollEvent(&e)) {
      switch(e.type) {
        case SDL_QUIT:            // quit
          quit = 1; break;
        case SDL_MOUSEBUTTONDOWN: // button pressed
          printf("button %d at %d,%d\n",
            e.button.button, e.button.x, e.button.y);
          break;
        case SDL_MOUSEMOTION:     // mouse motion
          printf("pos: %d,%d\n",e.motion.x,e.motion.y);
          break;
        case SDL_KEYDOWN:         // key pressed
          printf("unicode: %d\n",e.key.keysym.unicode);
          break;
        case SDL_JOYBUTTONDOWN:   // joystick events
        case SDL_JOYAXISMOTION:
        case SDL_JOYBALLMOTION:
        case SDL_JOYHATMOTION:
      }
    }
  }
}
```

```r
dynport(SDL)
dynport(GL)
main <- function() {
  quit <- FALSE
  e <- new.struct(SDL_Event)
  # Init SDL:
  SDL_Init(SDL_INIT_VIDEO+SDL_INIT_JOYSTICK)
  SDL_SetVideoMode(640,480,32,SDL_OPENGL)
  SDL_EnableUNICODE(1)
  # Main loop:
  while(!quit) {
    # Update display:
    glClear(GL_COLOR_BUFFER_BIT);
    SDL_GL_SwapBuffers();
    # Process events:
    while(SDL_PollEvent(e)) {

      if (e$type == SDL_QUIT) {
        quit = 1
      } else if (e$type == SDL_MOUSEBUTTONDOWN) {
        cat("button", e$button$button,
            " at ",e$button$x,",",e$button$y,"\n");
      }
      else if (e$type == SDL_MOUSEMOTION) {
        cat("pos: ",e$motion$x,",",e$motion$y,"\n");
      }
      else if (e$type == SDL_KEYDOWN) {
        cat("unicode: ",e$key$keysym$unicode,"\n");
      }
      else if (e$type == SDL_JOYBUTTONDOWN) { }
      else if (e$type == SDL_JOYAXISMOTION) { }
      else if (e$type == SDL_JOYBALLMOTION) { }
      else if (e$type == SDL_JOYHATMOTION) { }

    }
  }
}
```

Listing 12: C and R: SDL Event Handling.

`SDL_Event` is a user-defined C `union` comprising overlays of event-specific C `struct` members. Figure 4.8 gives a graphical illustration of this data structure; note that the objects left to the top-level `SDL_Event` are all `struct` members, some of which are nested. All `struct` members consist of a common header, similar to S-Expressions. The header comprises a single integer-based type identifier, which is also accessible as a top-level member field of the union, named `type`, which indicates the type of the event. Corresponding symbolic constants are defined in the C API of SDL such as `SDL_QUIT` for close events, `SDL_MOUSEBUTTONDOWN` for mouse button events, and so on. After the event type is known, an event-type specific member of the union is accessed to get detailed event information, e.g. when `e.type` equals to `SDL_MOUSEBUTTONDOWN`; subsequent code selects the member field `button`, which is of C struct type `SDL_MouseButtonEvent`, in order to accessed the button number (`button.button`) and coordinates (`button.x` and `button.y`) of the pointing device.

As shown in Listing 12 the processing is structured using a block of conditional cases where each one handles a specific event type. The processing of event data is done consistently via access to member fields by *symbolic name*. So the FFI of `rdyncall` was extended by a framework for handling foreign C composite data types via *syntactic sugar* in R, such as accessing foreign member fields by symbolic field names.

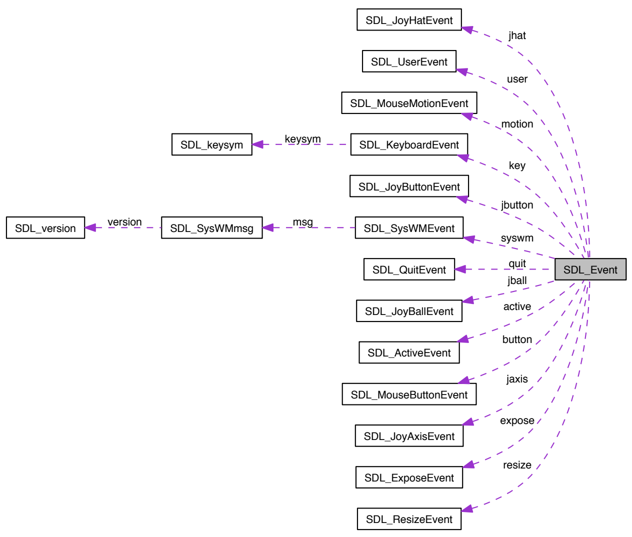


Figure 4.8: Overview of `union SDL_Event` C data type for processing user-interface events.

4.5.1 Interface

Our suggested R interface for symbolic access to members of C `struct` and `union` data objects leans on the corresponding C syntax by using the following syntax mapping between C and R:

	C Syntax	R Syntax
Read	`value = pointer->member`	`value <-pointer$member`
	`value = reference.member`	`value <-reference$member`
Write	`pointer->member = value`	`pointer$member <-value`
	`reference.member = value`	`reference$member <-value`

The *syntactic sugar* for the required R syntax can be achieved via S3 classes and method overloads for the operators `$` (read) and `$<-` (write). For this purpose a new S3 class, named "`struct`", was defined that is used as a *proxy* for pointers or references to C composite data types. Two S3 methods are defined for the implementation of read and write operations on member fields:

```
$.struct <-function(x, index) { ... }
$<-.struct <-function(x, index, value) { ... }
```

The R proxy object `x` is represented either as an external pointer or as an atomic vector; the latter is used for user-allocated objects that is managed by the R garbage collector; see Section 4.5.4.1 for details. By convention of the R syntax of the `$` and `$<-` operators, the `index` is passed as a `character` string. So in this context it gives the symbolic name of the member field of a foreign C data type.

In general read/write operations at member-field level are carried out by value conversion of a value from/to the member field's *type* and *address*. The *address* of a member object can be decomposed as

$$address = base + offset$$

where *base* refers to the address of the composite object, given by `x`, and *offset* is a constant value specific to the particular *member* of a *type*. Thus the offset can be *shared* across all objects, and together with the *type* of `x`, it is indirectly given by `index`.

The mapping between a proxy `x` to the underlying *type* is achieved via an attribute "`struct`" that is attached to the proxy; it gives the *type name*.

Since the *offset* needs to be determined once per data type, the use of a helper R object was utilized. For each `struct` and `union` data type that needs to be supported by this framework, the data type is registered and a corresponding R helper object is created, which comprises total *sizes* and *alignment* properties as well as *names*, *types* and computed *offsets* of member fields.

4.5.2 Registration of C data types

So far we utilized type signatures for the specification of function type information. Now we use signatures for the registration of C data types; we introduced a *DynPort* encoding for `struct` and `union` data types in Section 3.7.3.3. The format was defined as follows (in text pattern notation):

⟨*struct-name*⟩ '{' ⟨*field-types*⟩ '}' ⟨*field-names*⟩ ';' ...

⟨*union-name*⟩ '|' ⟨*field-types*⟩ '}' ⟨*field-names*⟩ ';' ...

The `rdyncall` package contains two parser functions that processes data type signatures and installs type information objects of S3 class `TypeInfo` as a side effect.

```
parseStructInfos(sigs, envir=parent.frame())
parseUnionInfos (sigs, envir=parent.frame())
```

`sigs` is a character string that comprises a batch of data type signatures. For each signature, a corresponding `TypeInfo` object is created and assigned in the environment given by `envir` with the same name as given by the data type signature (e.g. the ⟨*struct-name*⟩ or ⟨*union-name*⟩, respectively).

The mapping between member names and offsets is done via an R data frame object, which comprises two columns, namely type and offset, indexed by member names as row names. The offset is a ABI specific value that is computed and cached when we create this object. The algorithm for the computation is described in Section 5.3.8.1.

4.5.3 Implementation

The implementation of `$.struct` and `$<-.struct` involves a sequence of steps for the computation of the *base* and *offset* for a given proxy `x` and symbolic member field name `index`:

1. The C type *name* of `x` is given by the attribute "`struct`" of `x`.

2. Type information about the composite C data type "*name*" is available from a corresponding `TypeInfo` S3 object, which needs to be registered via `parseStructInfos` and `parseUnionInfos`.

3. The `TypeInfo` object comprises a table of member field *offset*s and *type*s which can be obtained by the member field's *name*, given by `index`.

4. The *base* address is provided by `x`, i.e. the value of the external pointer or the start address of the first element of an atomic vector object.

After the parameters *base*, *offset* and *type* have been determined the read/write operation can be performed using `pack` and `unpack`, which are discussed in the following section. Note that the difference between precomputation and handling of unions and structs is marginal; union member fields have a fixed offset at 0.

4.5.4 Low-level Interface

The low-level interface comprises two functions: `.pack` for packing R values to memory locations using a C data representation (marshalling), and `.unpack` for unpacking C values from memory locations to R values (unmarshalling). The interface is as follows:

```
value <-.unpack(x, offset, sigchar)
 .pack (x, offset, sigchar, value)
```

`.unpack` reads a C value from memory and returns the value converted to a scalar R value. `.pack` converts an R `value` to a C data type representation and writes the data to memory. The C data type is indicated by the `sigchar` character string that consists of a single character using the type signature scheme giving in Table 4.6. The effective memory location for reading and writing data is computed by `x` and `offset`, where `x` specifies a base memory address and `offset` represents a byte offset to be added to or substracted from that base. `x` can be either an external pointer, to refer to external memory, or an atomic R vector. In the later case one operates on the internal R vector storage in-place using the start address of the data area skipping header information including the S-Expression header and vector length field of the object `x`.

The two functions `.pack` and `.unpack` represent very powerful operations that perform pointer operations at C level from within R. But they are also very dangerous since they can give access to internal R memory and, if wrongly used, corrupt the management structures of R and other components of the run-time environment. The usual call-by-value copying semantics are not given. For example when passing a numeric vector as `x` to `.pack`, the object can have been modified when the function call returns. But this also gives great flexibility when working with C APIs that require user code to work with C data objects. The `value` passed in from R can be any atomic vector type; it is converted to the C data type according to the given signature and described in detail in Section 4.3.6.

Example We give an example in which `.pack` and `.unpack` can be very helpful to manipulate `raw` R memory in order to make use of certain API functions that would otherwise be inaccessible.
The `.dyncall` FFI offers flexible scalar conversion of values for arguments but the support for C functions that receive pointers to C arrays is limited. Given that a C function expects that the array of a scalar C type initialized by user code, it is not clear if this can be accomplished directly in R with the given means provided by `.dyncall`. As long as the C array type is *compatible* with an R atomic vector type (where the R implementation matches with that of the underlying C array type), we can initialize an object of that type in R and then pass it as an argument i.e. we have built-in support for C arrays of type `bool`, `char`, `int`, `double` arrays but no support for `short` and `float`.
Since `.pack` and `.unpack` also offer flexible conversion of R scalar values as given by `.dyncall`, but for reading and writing within memory chunks, we can create foreign data types that do not exist in R. That said, we can add virtual data types to this facility such as an array for 16-bit short values and write values passed in from R as numeric or integer values. 16-bit data buffers of `short` signed integer are often used for data exchange with audio output components.

We give a converter function `as.short` that transforms a `numeric` vector (with values ranging from -1 and 1) to a buffer of 16-bit signed integer values stored as in a `raw` vector using the `.pack` function:

```
as.short <- function(x)
{
  n <- length(x)
  r <- raw(2*n)
  for(i in 1:length(x)) {
    .pack(r, (i-1)*2, 's', x[[i]]*2^15)
  }
  return(r)
}
```

4.5.4.1 Allocation of C data types

The framework includes a utility function for allocation of C data type objects in R memory:

```
dataobj <-new.struct(name)
```

The function `new.struct` allocates a `raw` vector whose length equals that of the C data type given by `name`. `name` can be a `character` string, a type information object, or if used within the same environment as given by `envir`, in literal form (where it refers symbolically to a type information object).

The returned `raw` vector `dataobj` is tagged with S3 class of `'struct'` and a named attribute `'struct'` which specifies the C data type.

Example SDL defines a data type structure for the specfication of colors as given in the top-left panel of Listing 13. In the bottom-left panel we give the steps in R to define the C data type, allocate an object in R memory, read/write fields and dump the object. The right-hand panel gives the contents of run-time type information about the data type given by a `TypeInfo` object.

```
// Definition of C composite data type
typedef struct SDL_Color {
  Uint8 r;
  Uint8 g;
  Uint8 b;
  Uint8 unused;
} SDL_Color;
```

```
> # Definition of C composite data type in R:
> parseStructInfos("SDL_Color{CCCC}r g b unused ;")
> # Allocation
> x <- new.struct(SDL_Color)
> # Write to fields
> x$r <- 60; x$g <- 80; x$b <- 100
> # Reading from fields
> cat("red ",x$r,"\n")
red 60
> # Print C data object
> x
struct  SDL_Color  {
   r :60
   g :80
   b :100
   unused :0
 }
```

```
# Dump of R Type Information object
> SDL_Color
$name
[1] "SDL_Color"

$type
[1] "struct"

$size
[1] 4

$align
[1] 1

$basetype
[1] NA

$fields
       type offset
r         C      0
g         C      1
b         C      2
unused    C      3

$signature
[1] NA

attr(,"class")
[1] "typeinfo"
```

Listing 13: R Run-time Type Information on C Composite Data Types.

4.6 Callbacks

Callbacks represent a second form of function call interaction between user code and a library. In contrast to library function calls, callbacks are performed in the opposite direction; from library code to user-defined functions. Callbacks offer library frameworks that enable the user to decide and implement a certain behaviour that should be activated upon a library-specific *event*.

A callback function is a user-defined C function that is registered to a library for later execution from within the execution context and by decision of the library. So it is called *passively* due to an event decided by library code.

If callbacks are to be utilized from within a scripting language via a FFI, it is required that scripting functions can be used where usually C function pointers are passed to library functions and the scripting function can be activated as of a function call from C. Therefore we added a facility for wrapping user-defined R functions as C function pointers.

4.6.1 Interface

The interface of `new.callback` is as follows:

```
new.callback( signature, fun , envir = new.env() )
```

`new.callback` wraps the R function `fun` as a C function pointer that can be called from C code. The type of the C callback function needs to be specified by a *call signature*, similarly to `.dyncall`, given by `signature`. `new.callback` returns an external pointer that represents a C function pointer. The underlying C function object is generated synthetically and works as an adapter that can be passed as argument to other foreign C functions for use as a C function pointer. During a callback from C, the C arguments are converted to R objects as specified by the call signature. The R function is called and evaluated within the environment given by `envir`. On return of the R function its return value is converted to a C value and control flow is passed back to foreign C code where the callback was originated.

Example The standard `C` library contains a number of sorting and searching algorithms. Although the library is compiled to native code, the algorithms offer a generic implementation design that allows to sort and search arrays of user-defined data types due to the use of *callbacks*. Any array of elements with a user-defined data type can be sorted or searched by these functions given that the user-defined data type

- is of byte-size granularity,
- has a fixed-size,
- and can be compared by a function that the user has to implement.

Among the sorting algorithms that are supported by the standard C library there is the quick sort algorithm provided as the C function `qsort` and declared as follows:

```
void qsort(void *array, size_t len, size_t el_size, int (*compar)(const void*, const void*));
```

The function has four parameters: `array` is a pointer to an array of values, `len` gives the length of the array, `el_size` specifies the byte size of the element type and `compar` is a C *callback* function pointer for comparing two elements.

`qsort` uses two operations, namely *compare* and *swap*, to offer a *generic* implementation that works with user-defined data types as illustrated in Figure 4.9. The *swap* operation works on memory chunks, specified by the first three parameters `array`, `len` and `el_size`. In order to determine equality of two elements as a criteria to swap the order, the algorithm calls the user-defined *callback* function `compar`

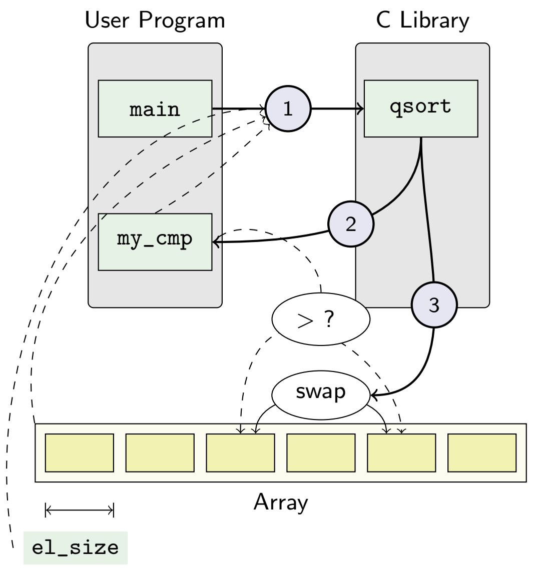


Figure 4.9: Generic quick-sort algorithm using C callbacks.

where the return value indicates the order, such as less-than (< 0), equal-to ($= 0$) or greater-than (> 0).

Listing 14 gives an example for sorting rational numbers using `qsort` from with in C and using `new.callback` in R.

```c
#include <stdlib.h>
typedef
  struct { int nom; int denom; }
  rational;

int my_cmp(const void *p1, const void *p2)
{
  rational* ra = (rational*) p1;
  rational* rb = (rational*) p2;
  double va = ((double)ra->nom)/((double)ra->denom);
  double vb = ((double)ra->nom)/((double)ba->denom);
  return (va<vb) ? -1 : (va>vb) ? 1 : 0;
}

int main() {
  rational values[4] = {
    {6,8}, {4,5}, {1,2}, {3,4}
  };
  size_t cnt =
    sizeof(array)/sizeof(array[0]);
  size_t elsize = sizeof(rational);

  qsort( values, cnt, elsize, &my_cmp );
  return 0;
}
```

```r
library(rdyncall)
dynbind(c("msvcrt","c","c.so.6"),"qsort(piip)v;")

mycmp <- function(p1,p2) {
  a_nom <- .unpack(p1,0,"i")
  a_den <- .unpack(p1,4,"i")
  va    <- a_nom / a_den
  b_nom <- .unpack(p2,0,"i")
  b_den <- .unpack(p2,4,"i")
  vb    <- b_nom / b_den
  if (va<vb) return (-1) else if (va>vb) (1) else 0
}

mycmp <- new.callback("pp)i", mycmp)

main <- function() {
  values <- as.integer(c(6,8,4,5,1,2,3,4))
  qsort(values, 4, 16, mycmp)
}
```

Listing 14: C and R: Using callbacks with C qsort.

We discuss the R implementation in more detail. As a prequisite we need to bind the C `qsort` function:

```
> dynbind(c("msvcrt","c","c.so.6"), "qsort(piip)v")
```

First, we define an R function that compares two rational numbers; the arguments are always passed as pointers to an array of two `int` numbers. The callback argument of `sqrt` has the following type:

```
int (*compare_fn) (void * pa, void * pb);
```

We define a compare function in R with two parameters; both parameters will also be of type `const void*` which is passed as an external pointer.

```
> f <- function(pa,pb) {
  an <- .unpack(pa,0,"i")
  ad <- .unpack(pa,4,"i")
  a <- an / ad
  bn <- .unpack(pb,0,"i")
  bd <- .unpack(pb,4,"i")
  b <- bn / bd

  if (a==b) return(0)
  else if (a<b) return (-1)
  else return(1)
}
```

Note that we need to extract the rational values from C pointer arguments. Both pointers, `pa` and `pb`, are represented as external pointer objects to an object comprising two C `int` objects, the nominal and the denominal component. Thus we use `.unpack` to extract the nominal and denominal part of each rational number, followed by a division to create a value. Note that we do not have to coerce the type although it is `integer`; R's division operator automatically converts operands to `double`. As a last step we compare both values and return the result as an indicator of 0 ($a = b$), -1 ($a < b$) and 1 ($a > b$). The return value indicates to `qsort` whether the two elements need to be swapped. The R function is now wrapped as a callable C function object via:

```
> callback <- new.callback("pp)i", f)
```

Note that we use '`p`' as an untyped pointer here. Finally we call `qsort` using the callback as a forth argument. As input vector we use the following sequence of rational numbers: $\frac{6}{8}, \frac{4}{5}, \frac{1}{2}, \frac{3}{4}$. We encode them as a `numeric` vector.

```
> values <- as.integer(6,8,4,5,1,2,3,4)
> qsort(values, length(values)/2, 16, callback)
```

After the function call returns, the values have been sorted in-place. An output as a matrix is given next:

```
> print(as.matrix(values,nrow=2))
     [,1] [,2] [,3] [,4]
[1,]    1    6    3    4
[2,]    2    8    4    5
```

Example In the second example callbacks are used for parsing XML documents.

There are two basic frameworks for processing of XML documents:

1. XML documents are parsed as a node tree representation in memory.
2. XML documents are parsed as a stream of data; for each identified type an event is notfied to user-supplied callback functions.

While the first variant is ideal for traversing in a tree data structure, the required amount of memory depends on the size of the document. The second variant has no large memory requirements. In order to define a processing application of an XML document the user has to define callback functions.

The second variant is supported by the `expat` C library (Clark, 2007). Users specify callback handlers for a set of XML processing events, such as the start tag or the end tag of an XML node. During event document processing event-data, such as the XML tag name and attribute maps, are passed by the parser to the callbacks.

For this example we use three functions of the expat C API, which we first need to bind via `dynbind`:

```
dynbind(c("expat","expat.so.1"),"
XML_ParserCreate(Z)p;
XML_SetElementHandler(ppp)v;
XML_Parse(pZii)i;
")
```

Then a parser object needs to be created. The constructor function `XML_ParserCreate` is declared as follows:

```
XML_Parser XML_ParserCreate(const XML_Char* encoding)
```

If encoding is non-`NULL`, the given C string specifies the encoding to use for the document. Otherwise the document's encoding declaration is used. For this example, a `NULL` is passed using the nil R object `NULL`.

```
p <- XML_ParserCreate(NULL)
```

The returned object is a pointer that represents a *handle* to the newly created parser object. Now the parser is configured by registration of callback functions. For handling of processing events, such as XML tag open and close elements, the following two callback functions are defined as follows:

```
typedef void (*XML_StartElementHandler) (void *userData, const XML_Char *name, const XML_Char **atts);
typedef void (*XML_EndElementHandler)   (void *userData, const XML_Char *name);
```

The first argument, `userData`, is used for passing application-specific data. The value can be bound with a parser object; for this example it can be ignored. Event data, such as the XML tag name, is transferred via `name` as a C string; note `XML_Char` is a type alias to `char`. For the callback signatures we can use the object type signature '`Z`', which indicates, that the argument should be automatically converted as a `character` object in R. Two R functions are defined that should be used as callbacks.

```
startFun<-function(user,tag,attr) {
  cat("Start tag:", tag, "\n")
}
S<-new.callback("pZp)v",startFun)

endFun<-function(user,tag) {
  cat("End tag:", tag, "\n")
}
E<-new.callback("pZ)v",endfun)
XML_SetElementHandler(p,S,E)
```

Now the parser can be tested on an XML document.

```
d <- "<hello><world></world></hello>"
XML_Parse(p, d, nchar(d), 1)
```

The following output is generated:

```
Start tag: hello
Start tag: world
End tag: world
End tag: hello
```

```
/* C: */
#include <expat.h>

void S(void* userData,
  const XML_Char *name,
  const XML_Char **attrs) {
  printf("onStart: %s\n", name);
}
void E(void* userData,
  const XML_Char *name) {
  print("onStop: %s\n", name);
}
int main() {
  XML_Parser p;
  char* d;
  p = XML_ParserCreate(NULL);
  XML_SetElementHandler(p,S,E);
  d ="<hello><world></world></hello>";
  XML_Parse(p,d,sizeof(d),1);
}
```

```
# R:
dynport(expat)

startFun<-function(user,tag,attr) {
  cat("Start tag:", tag, "\n")
}
S<-new.callback("pZp)v",startFun)

endFun<-function(user,tag) {
  cat("End tag:", tag, "\n")
}
E<-new.callback("pZ)v",endfun)


main <- function() {
  p <- XML_ParserCreate(NULL)
  XML_SetElementHandler(p,S,E)
  d <- "<hello><world></world></hello>"
  XML_Parse(p, d, nchar(d), 1)
}
```

Listing 15: C and R: Using callbacks with Expat XML Parser C library.

In contrast to this facility, other FFIs for R, such as the package `base` or `Rffi` (Lang, 2011), do not support callbacks. The callback support is based on the `dyncallback` C library, which is discussed in Section 5.5. In Section 5.5.6 we discuss the application for the callback facility of `rdyncall`. Note that the supported set of argument and return types is equal to that of `.dyncall`. However, there are several missing platform ports of `dyncallback`.

4.7 Dynamic R Bindings to C Libraries

In this section we discuss a high-level interface for loading dynamic R bindings to C APIs. So far we have considered functions of the `rdyncall` package for the creation of R wrapper functions to foreign C functions and helper objects for working with foreign C run-time data objects. Now these functions

DynPort	Library/Description	Functions	Constants	Data Types Struct/Union
`GL`	OpenGL	337	3253	-
`GLU`	OpenGL Utility	59	154	-
`glew`	OpenGL Extension Wrangler	1465	-	-
`gl3`	OpenGL Version 3	324	838	1
`SDL`	Simple Directmedia library	203	465	51
`SDL_image`	Pixel image loaders	29	-	-
`SDL_mixer`	Music player & loaders	63	12	-
`SDL_ttf`	Font format loaders	35	9	-
`SDL_net`	Network programming	34	5	3
`expat`	XML parsing(Clark, 2007)	65	70	-
`CUDA`	GPU programming	387	665	84
`OpenCL`	GPU programming	78	260	10
`stdio`	Standard I/O	76	3	-
`R`	R library	238	700	27

Table 4.10: Table of DynPorts: R Bindings to C Libraries as of `rdyncall` 0.7.4.

are incorporated into a data-driven component for management of a number of R bindings to C APIs.

For each supported C API a *DynPort* file is defined which comprises C API information encoded using the *DynPort* type signatures. In the current version *DynPort* files are written as R scripts. They comprise function calls to services of the `rdyncall` package, such as a call to `dynbind` for the creation of wrapper functions and calls to `parseStructInfos` and `parseUnionInfos` for the registration of C data type information, and also assignment statements for defining symbolic constants. As a binding is specified by text-based *DynPort* signatures, the parser framework that was discussed in Section 3.8 can be used for automation of *DynPort* files from C API header files. Table 4.10 gives an overview of cross-platform R bindings to C APIs currently available within the package.

4.7.1 Interface

The interface for loading a *DynPort* resemblance that for loading R packages via `library(pkgname)` or `require(pkgname)`:

```
dynport(portname)
```

`portname` is a character string or literal value that specifies the name of a *DynPort* file. The file is searched for within a repository directory, which is located relative to the package installation of `rdyncall` using the file path pattern "`dynports/`⟨*name*⟩`.R`". The *DynPort* file is evaluated using the `base` R function `sys.source`.

In R environment objects are used for managing named objects. When the R interpreter is evaluating an expression and it encounters a symbolic name, the value for that name is searched for within a number of environments. It first searches for a named object in the local and enclosing environments of the function of the expression in lexical scope. Then the interpreter searches within a sequence of environments, which are stored in the 'search path'.

As each *DynPort* file is evaluated in a new environment, the bindings are organized in isolated environment objects. By attaching the environment object to the search path, the wrapper components to a C API become available at global lexical scope.

Example The following example illustrates the resemblance of this mechanism by comparing the interface of the R package loader and the *DynPort* loader.

After loading an R package via `library(pkg)` the effect becomes visible via `search()`, which gives the current search path:

```
> library(lattice)
> search()
 [1] ".GlobalEnv"        "package:lattice"   "tools:RGUI"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

As a result of loading an R package, a public environment is created and inserted into the search path at the second index. The first index is reserved for the global workspace environment, named `.GlobalEnv`, so that user-defined objects that name clash with object of packages are given precedence.

The same mechanism is used for managing dynamic R bindings in `rdyncall`. As a result of loading a *DynPort*, an environment is created with R wrapper objects to C API components. The environment object is attached to the search path at index two.

```
> library(rdyncall)
> dynport(SDL)
> search()
 [1] ".GlobalEnv"        "package:SDL"       "package:rdyncall"
 [4] "tools:RGUI"        "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:utils"     "package:datasets"
[10] "package:methods"   "Autoloads"         "package:base"
```

4.7.2 Implementation

We take closer a look at the content of a *DynPort* environment, such as the one of `SDL` library after loading via `dynport(SDL)`:

```
> str(ls(2))
 chr [1:651] "AUDIO_S16" "AUDIO_S16LSB" ...
> table(sapply(ls(2), function(x) typeof(get(x,2)) ))
closure  double    list
    201     416      34
```

The output of `str(ls(2))` reveals that the `SDL` *DynPort* consists of 651 R objects that are of three different R object types. Table 4.11 gives an overview of the three object types, their corresponding C

API element and their implementation, based on a specific *DynPort* signature encoding and `rdyncall` package function or statement form.

R Wrapper	C API Element	Signature	R Implementation
`closure`	Function declarations	*library*	`dynbind`
`list`	`struct` definitions	*struct*	`parseStructInfos`
	`union` definitions	*union*	`parseUnionInfos`
`double`	`#define NAME REPLACEMENT`	*value*	Assignments:
	`enum { NAME = VALUE, .. }`	*value*	$\langle name \rangle = \langle value \rangle$

Table 4.11: Implementation of DynPort in R for each different category of C API Element; the signature format is given and the R implementation function or expression.

DynPort files, distributed with the package, are generated by means of the automation framework described in Section 3.8. An excerpt of the `SDL` *DynPort* R script is given in Listing 3.

4.7.3 Installation of run-time libraries

As a prequisite to working with *DynPort* bindings, the shared library needs to be installed on a system. If a shared library is not found, `dynport` gives the following message:

```
dynbind error: Unable to find shared library 'SDL'.
For details how to install dynport shared libs, type: ?'rdyncall-demos' might help.
If there is no information about your OS, consult the projects page how to build and install the shared library
for your operating-system.
Make sure the shared library can be found at the default system places
or adjust environment variables (e.g. %PATH% or $LD_LIBRARY_PATH).
```

Currently there is work in progress to incorporate automatic installation of shared libraries by using an abstraction layer to package management systems. For now several installation procedures are collected in a manual page accessible via `?‘rdyncall-demos‘`. For `OpenGL` and `SDL` the platform-specific installation procedure is briefly summarized in the following paragraph.

Table 4.12 gives an overview of the various installation procedures and commands to install pre-compiled packages for different operating systems.

`OpenGL` is usually part of the operating system; it is pre-installed on Microsoft Windows and Apple Mac OS X operating systems. Depending on the Linux Distribution, OpenGL might already been installed; otherwise it is available as part of desktop environments or explicitly named as `GL` or also as `Mesa`[2]. Usually BSD systems require explicit installation of OpenGL. `SDL` is usually not pre-installed but often available through package management systems or via explicit download and manual installation.

[2]Mesa is an open-source implementation of OpenGL.

OS/Distribution	Installation
Mac OS X	Download `SDL-1.2.15.dmg`, mount and copy `SDL.framework` to `/Library/Frameworks`.
Windows 32-bit	Download `SDL-1.2.15-win32.zip`, unpack and add directory to `%PATH%`.
Windows 64-bit	Download `SDL-1.2.15-win32-x64.zip`, unpack and add directory to `%PATH%`.
Debian Linux 6	`$ aptitude install libsdl1.2debian`
Fedora Linux 13/14	`$ pkcon install mesa-libGL SDL`
Ubuntu 12.04 LTS	`$ apt-get install libsdl1.2deban`
ArchLinux 2013.5	`$ pacman -S mesa sdl`
FreeBSD 9.0	`$ pkg_add -r sdl`
OpenBSD 4.8	`$ pkg_add SDL`
NetBSD 5.1	`$ pkg_add Mesa SDL`

DragonFly 3.4	`$ pkg_radd SDL`
	`$ pkg install sdl`

Table 4.12: Installation of SDL and OpenGL on various platforms.

4.8 Example

In this section we illustrate the usage of `rdyncall` and `dynport` for *scripting R applications by gluing C library components*. We present a 'demo' application written in R and available from the package. We discuss its implementation and we focus on R programming techniques for exchanging data with C APIs using the example of the R and the OpenGL API.

4.8.1 Simulation and Visualization of Random Fields using OpenGL

The study of interdependent data, such as time series, spatial field, spatio-temporal processes of observations of even higher dimensions, is of importance in a number of branches of science and technology. Of interest is the modelling and estimation of the dependency structure of the observations. A popular way to assess the plausibility of the stochastic processes used to model such observations is to generate realizations from the model and check whether these display the same features and properties as the observed values.

The R package `RandomFields` (Schlather et al., 2013) provides a wide range of simulation methods for generating random fields. The simulation task involves a large number of computational operations depending on the dimension and size of the field and the number of point processes, etc.. Although the package is mainly implemented in C, the simulation of random fields can be considered an off-line process; it is not fast enough for real-time graphics visualization or simulation.

The field of computer graphics offers efficient real-time graphics methods for imitating the behaviour of natural processes such as gases, cloud, fluids and natural materials. More recently, GPU hardware is also being utilized in scientific applications such as high-performance simulation and visualization in real-time. We give a simple example in which we use R and OpenGL via `rdyncall` for the computation and display of Gaussian Markov random fields in high resolution (512x512); for each round of simulation we simulate a large number of random point processes (5000) and we achieve a rate, required for real-time graphics output and visualization, between 30-100 FPS (Frames Per Second).

The demonstration is available as a `demo` in `rdyncall`:

```
> library(rdyncall)
> demo(randomfield)
```

Note that the demo requires `SDL` and `OpenGL` shared libraries for execution (see Section 4.7.3 or open the help file with `?'rdyncall-demos'` for installation details).

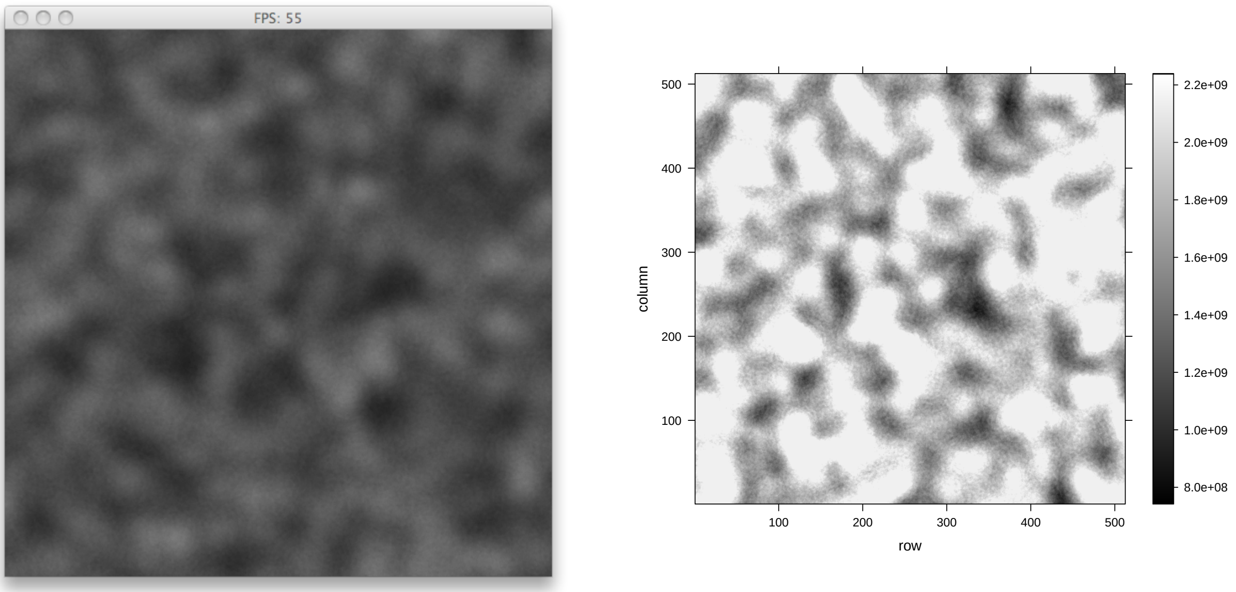


Figure 4.10: Screenshot of real-time visualization (*left*) and plot (*right*) of `demo(randomfield)`.

The real-time visualization of the simulation is displayed in a separate output window , depicted in the left-hand panel of Figure 4.10. The simulation runs at 55 FPS (indicated in the window's title bar); the output resembles quick motion of particles in a swirl. By pressing the left button on the pointing device, the current frame is downloaded from the GPU to R as a matrix object and then plotted in R (right-hand panel in Figure 4.10). Note that although the 'simulation and visualization' task generates approximately 55 matrices per second[3] but it can take several seconds to download data from graphics memory to R and to plot the matrix subsequently using R graphics.

4.8.1.1 Implementation

We now discuss the implementation of `demo(randomfield)`. It is written entirely in R and uses `OpenGL` via `dynport(GL)`/`rdyncall` so that it runs across major R platforms. The 'graphics hardware' require-

[3]Platform: Mac OS X 10.6 MacBook Pro, 2.5 GHz using Nvidia 8600M GT.

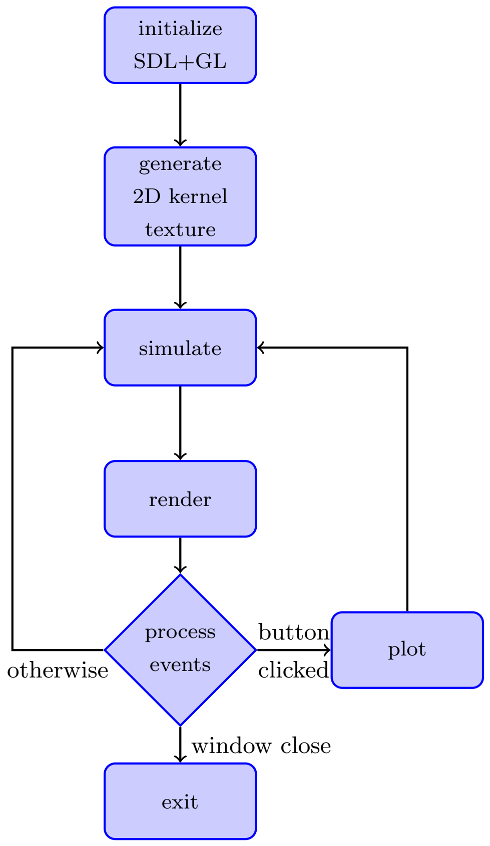


Figure 4.11: Control flow diagram of `demo(randomfield)`.

ments for running this demonstration are low as we use the OpenGL Version 1.1 API. In the following discussion we focus on various OpenGL function calls with an emphasis on efficient data exchange between R/`rdyncall` and C.

We generate a finite field of random values of size 512×512 elements. The values assigned to these elements are generated in a number of steps. First 5000 points are sampled from a spatial point process over the field. A 2D Gaussian kernel is placed over each point. Associated with each point is also a randomly sampled radius attribute. The value of each element of the 512×512 matrix is given by the sum of the values of the generated kernels at the coordinates of the elements. The computation of the final matrix is implemented by means of OpenGL via *rendering* squares (one for each point process) using *texture mapping* of an image (the kernel) and *blending* (for summing up the kernel values).

Overview The control flow diagram in Figure 4.11 outlines the steps implemented by the software. In a first stage, we initialize the application. Then we generate a 2D Gaussian kernel matrix by using R functions; we upload the matrix to OpenGL as a texture image i.e. the kernel is transferred to fast memory of the GPU. After that we run the 'simulation and visualization' main loop. In each round we simulate 5000 point processes (including radii) using standard R functions. We then draw the points as squares to produce the final image using OpenGL. Then the display is refreshed. As a last step of the round, we process input events; if the window close button was pressed, we stop the main loop; if the mouse button was pressed, we transfer data from OpenGL to R for plotting.

Main Function The application is written as a main function that calls sub-functions for particular tasks. After initialization the main loop starts: 5000 observations are sampled using R, and then drawed as squares via OpenGL. Finally input events are processed and the FPS counter is updated.

```
main <- function() {
  init()
  while(!quit) {
    sim <- sim(5000)
    render(sim)
    pollEvents()
    fps <- fpsUpdate(fps)
  }
}
```

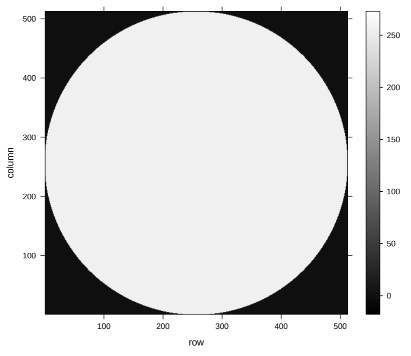
Initialization At first, we load *DynPorts* of `SDL` and `OpenGL`, and initialize the video sub-system. We define the output matrix by opening a OpenGL video output surface with 512 $\times$ 512 pixels.

We initialize global parameters of the OpenGL graphics rendering pipeline: "clear color" is set to zero, the blending operation is enabled and configured, and the parameters for data transfer are configured for download of data from GPU memory to R. We then generate a kernel matrix that is uploaded to OpenGL. As a final step we initialize a FPS counter object and initialize global variable `quit` to `FALSE` (the invariant of the mainloop).

```
init <- function() {
  # -- Load Bindings
  dynport(SDL)
  dynport(GL)
  # -- Init OpenGL Output
  stopifnot(!SDL_Init(SDL_INIT_VIDEO))
  surface <<- SDL_SetVideoMode(512,512,32,SDL_OPENGL+SDL_DOUBLEBUF)
  # -- Set OpenGL Globals
  glClearColor(0,0,0,0)
  glColor3d(colorunit,colorunit,colorunit)
  glBlendFunc(GL_SRC_ALPHA, GL_ONE)
  glEnable(GL_BLEND)
  glPixelStorei(GL_PACK_ALIGNMENT,1)
  # -- Compute kernel
  img <- genTex.bnorm(512)
  # -- Prepare OpenGL Texture Unit
  initTex()
  loadTex(img)
  # -- Initialize globals (for main loop)
  fps <<- fpsInit()
  quit <<- FALSE
}
```

Generate Kernel Matrices We specify the kernel by computing a matrix of values. Later we upload this matrix to the texture mapping unit of the GPU. We give two R functions and corresponding R plots in Listing 16

```
genTex.circle <- function(n) {
  m<-matrix(nr=n,nc=n);r<-n/2
  for(i in 1:n) {
    for(j in 1:n) {
      m[[i,j]] <- ifelse(
        (i-r)^2+(j-r)^2 > r^2,
        0L,255L
      ) } }; return(m)
}
```

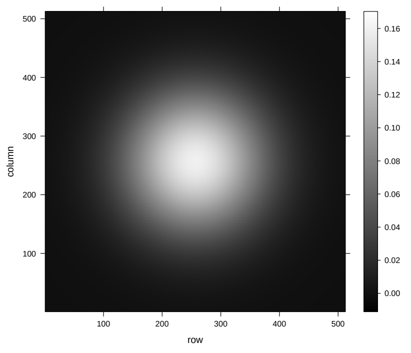


```
genTex.bnorm <- function(n) {

  x <- seq(-3,3,len=n)
  d <- dnorm(x)
  m <- outer(d,d)

  return(m)

}
```



Listing 16: R: Texture generators for 2D Kernels: R function (left) and corresponding plots (right); Kernels: Boolean circular (top) and Bivariate Gaussian (bottom).

Preparing the OpenGL Texture Unit During initialization, we also need to allocate an OpenGL texture object resource and upload image data to the GPU from R. We discuss this task in more detail now. The allocation function in C is given as follows:

```
void glGenTextures(GLsizei n, GLuint* textures);
```

As a result of this call, OpenGL assigns `n` distinct identifier numbers and outputs them in a user-allocated array of C `GLuint` elements (`GLuint` is a type alias to `unsigned int`). Since the internal format of an R `integer` vector is compatible with `GLuint*`, we can allocate an R object of length 1 and pass it as the second argument.

```
tex.ids <- integer(1)
glGenTextures(length(tex.ids),tex.ids)
```

We give now the C API function for uploading image data to OpenGL:

```
void glTexImage2D(GLenum texture, GLint    level, GLint internalFormat,
                  GLsizei  width, GLsizei height, GLint border, GLenum format,
                  GLenum    type, const GLvoid* pixels);
```

We use this function in the R function `initTex` for uploading an R atomic `numeric` matrix as a texture image to the texture object; in this example the texture object is specified by the global variable `tex.ids[[1]]`. We convert the input matrix `img` to a `raw` vector. Note that we use a primitive format for the texture image data type by using unsigned 8-bit integer values to provide for compatibility with older graphics cards.

After we have bound the texture object to the texture 2D unit, we transfer image data to the texture unit via `glTexImage2D`. The last two parameters of `glTexImage2D` specify the format and pointer; we specify the internal format `GL_UNSIGNED_BYTE` and pass the `raw` vector as a pointer value. After the function call, R data has been copied to GPU memory and it is associated with the texture object.

```
initTex <- function(img) {
  m <- max(img)
  texdata <- as.raw( ( img/m ) * 255 )
  glBindTexture(GL_TEXTURE_2D, tex.ids[[1]])
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1)
  glTexImage2D(GL_TEXTURE_2D, 0, GL_ALPHA, nrow(img), ncol(img), 0,
    GL_ALPHA, GL_UNSIGNED_BYTE, texdata)
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,GL_CLAMP_TO_BORDER)
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,GL_CLAMP_TO_BORDER)
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
  glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE)
}
```

Simulation We sample 2D coordinates (x and y) and a radius value for each mixture component.

```
sim <- function(N=5000) {
  obj <- list(
    x=runif(N, -1.1, 1.1),
    y=runif(N, -1.1, 1.1),
    r=runif(N,  0.1, 0.2)
  )
  return(obj)
}
```

Visualization Initially we clear the frame buffer to black color (by means of a matrix with all elements set to zero) by OpenGL `glClear(bufferFlags)`. Then we call a function that renders squares, specified by simulated data of x, y and radius vectors. Finally the graphics rendering output is presented on the display via swapping front and back video buffers (as we initialized the video output display using the `SDL_DOUBLEBUF` flag).

```
draw <- function(o) {
  glClear(GL_COLOR_BUFFER_BIT)
  drawSquareTexVertexArray(o$x,o$y,o$r)
  glFinish()
  SDL_GL_SwapBuffers()
}
```

Draw texture-mapped 2D squares At first we transform input data to vertex and texture coordinates in order to pass data to OpenGL. Then we enable texture mapping and client states; the latter is used for transfer of vertex information via pointers. We now pass R objects to OpenGL vertex and texture coordinates via function calls to `glVertexPointer` and `glTexCoordPointer`. Then we draw all square shapes via a single function call; OpenGL reads vertex information from R objects. Since we are about to leave the function afterwards, we need to disable OpenGL pointer access to R object.

```
drawSquareTexVertexArray <- function(x,y,r) {
  n <- max(length(x),length(y),length(r))
  x1 <- x-r ; y1 <- y-r
  x2 <- x+r ; y2 <- y+r
  vertexArray   <- as.vector(rbind(x1,y1,x2,y1,x2,y2,x1,y2))
  texCoordArray <- rep( as.double(c(0,0,1,0,1,1,0,1)), n )
  glEnable(GL_TEXTURE_2D)
  glEnableClientState(GL_VERTEX_ARRAY)
  glEnableClientState(GL_TEXTURE_COORD_ARRAY)

  glVertexPointer(2,GL_DOUBLE,0,vertexArray)
  glTexCoordPointer(2,GL_DOUBLE,0,texCoordArray)

  glDrawArrays(GL_QUADS, 0, n*4)
  glDisableClientState(GL_VERTEX_ARRAY)
  glDisableClientState(GL_TEXTURE_COORD_ARRAY)
  glDisable(GL_TEXTURE_2D)
}
```

Event Handling We process the input queue to determine when to stop the main loop. We also check if the button pointer device was recently pressed to initiate transfer of data to R for plotting.

```
pollEvents <- function() {
  doReadPixels <- FALSE
  while( SDL_PollEvent(event) != 0 ) {
    type <- event$type
    if (type == SDL_MOUSEBUTTONDOWN) {
      doReadPixels <- TRUE
    } else if (type == SDL_QUIT) {
      quit <<- TRUE
    }
  }
  if (doReadPixels) {
    pixels <<- readpixels()
    image(pixels)
  }
}
```

Download GPU data to R Data from the frame buffer are read via `glReadPixels`. We allocate an R `integer` matrix in advance, which is passed as a pointer; OpenGL writes data into that object.

```
readpixels <- function()
{
  array <- matrix(NA_integer_,512,512)
  glReadPixels(0,0,fb.size,fb.size, GL_LUMINANCE, GL_INT, array)
  return(array)
}
```

Frames Per Second Counter We define functions for the initialization and update of an FPS counter represented as an R `list` object. For the implementation we make use of C functions of `SDL`, such as `SDL_GetTicks` to query time, and `SDL_WM_SetWindowCaption` to update the window title bar.

```
fpsInit <- function() {
  list(
    tbase  = SDL_GetTicks(),
    frames = 0
  )
}
fpsUpdate <- function(fps) {
  tnow <- SDL_GetTicks()
  if ((tnow - fps$tbase) > 1000) {
    fps$tbase <- tnow
    SDL_WM_SetCaption(paste("FPS:", fps$frames),NULL)
    fps$frames <- 0
  } else {
    fps$frames <- fps$frames + 1
  }
  return(fps)
}
```

4.9 Summary

This chapter covers a middleware package for R that offers scripting access to C API components. The package contributes a dynamic FFI for code- and data-level interoperability between R and C. Based on this foundation it also offers a simple cross-platform interface for dynamic loading of R bindings of C APIs. This concept provides a model for rapid application development. We motivated the approach by comparing the merits of developing interactive 3D real-time graphics applications, based on OpenGL and SDL, in R/rdyncall instead of C.

We illustrated the inconvenience of compiling and linking user code with libraries in C. In contrast to this, we emphasized the potential for scripting ambitious applications, based on portable C libraries, in a dynamic language by using dynamic bindings to C APIs. Since the core application code is interpreted it can be executed immediately across a range of platforms even though it uses low-level components such as OpenGL and SDL.

As an example of this package we discussed an implementation of the *Dynamic Bindings Model* based on core services of a dynamic FFI. We first discussed the components of this foundation layer comprising facilities for loading external C libraries, resolving functions, making calls, handling C run-time data objects and wrapping R functions as C callback functions.

As a prequisite for *cross-platform* scripting of shared library C components we considered the general problem of loading code. While a large number of valuable C libraries exist as portable components with a stable C API across platforms, providing a *common* interface for requesting access is challenging due to the platform-specific details for naming and loading a C library. We illustrated the diversity of naming schemes by comparing the file paths of four C libraries across ten operating systems. We discussed the details of dynamic linkers and presented a method for loading libraries across platforms with a simple interface.

A common text-based interface was used across core components based on the *DynPort* type signature encoding format; we showed several advantages of this design choice. Firstly, the compact and intuitive encoding scheme enables users to experiment with foreign C components in interactive development sessions. Secondly, type signatures are used for driving function calls but also as input parameters for type checking and value conversion so that type-safe and convenient interfaces to C APIs can be created by wrapping FFI call objects. Thirdly, components for code- and data-level interoperability can be combined via type signatures.

We showed the advantage of character strings for encoding and passing type information across different language contexts. Scripting languages, such as R, offer efficient text processing tools to split the text chunk of C API specifications into components for type information. The latter is put into R wrappers as character objects and then, at call-time, is processed by the workhorse of the FFI in C for driving foreign function calls, type checking and value conversion.

The discussion was illustrated by several examples that emphasized the syntactic similarity of R to C user code. We closed this chapter with a comprehensive example of a visualization and simulation software written in portable R code using `SDL` and `OpenGL`, where advanced techniques for data transfer between R and C APIs were addressed. Included here was uploading of R vector data for texture mapping and processing of vertex information.

The package was implemented in R and C as a portable middleware solution by using a *Generic Dynamic FFI* library which provides an abstraction layer to ABI details of the calling sequence for code-level interoperability; a portable C interface encapsulated non-portable code in assembly language implemented for a range of ABI platforms. The design of the *Generic Dynamic FFI* library `dyncall` and details of data- and code-level interoperability at ABI level across five processor architecture families is covered in Chapter 5.

Chapter 5

Dynamic Interfaces to Compiled Code

This chapter covers a software abstraction layer with a portable and dynamic C interface for making function calls, handling of callbacks and loading code. This layer represents a fundamental building block for the implementation of *portable Dynamic FFIs*, such as `rdyncall` for R presented in Chapter 4.

As an introduction to the topic we consider the anatomy of function calls at machine level. Subsequently we give a survey of binary interfaces, including calling conventions and data type mapping schemes, on five current processor architecture families and common operating systems.

Finally we discuss the open-source *Generic Dynamic FFI* software package *DynCall* comprising three small C libraries. `dyncall` provides a portable interface for making dynamic function calls with support for arbitrary function types including support for several calling conventions. `dyncallback` offers an implementation framework for dynamic handling of C callbacks. Lastly, `dynload` offers a portable C API to dynamic linkers of various operating systems. We discuss the design of the C APIs and the implementation in C and partly in assembly language for a range of processor architectures and operating systems. The chapter closes with a discussion on development issues including portable build systems and software testing suites of the package.

5.1 Anatomy of Function Calls

At first we consider the basic execution environment of machine-level function calls, i.e. the core components of computer platforms and the run-time organization of processes and threads within an operating system. We then give a brief overview of some characteristics of calling conventions and discuss general steps of the machine for making a function call.

5.1.1 Execution Environment

The CPU (Central Processing Unit) of a processor is the *heart* of a computer system. It executes the operating-system kernel and user programs by processing primitive instructions encoded in machine code which are read from the main memory via an address bus. CPUs contain a small number of general-purpose *registers* to store fixed length bit sequences that are interpreted by instructions as operand values for bit/integer arithmetic or as memory addresses for load/store memory operations.

In addition a set of registers for special purposes and hardwired tasks is built-in, for example, to control the execution of programs. The PC (Program Counter) references the next instruction of execution. The CPU fetches a word of machine code and decodes the instruction which is then executed and the PC is advanced to the address of the next machine instruction.

For the implementation of control flow, such as *goto* jumps and *for/while* loops in high-level languages, `branch` or `jump` instructions (explicitly or conditionally) load the PC. Furthermore, machines offer a `call` instruction (or semantics) with which sub-routine procedure and function calls can be executed, which we discuss in more detail in the next section.

The machine-code encoding defines the mapping between binary codes and corresponding instructions and operands specified in the ISA (Instruction Set Architecture). "One of the important abstractions that a programmer uses is the instruction set architecture (ISA). The ISA defines the personality of a processor and specifies how a processor functions: what instructions it executes, what interpretation is given to these instructions, and so on. The ISA, in a sense, defines a logical processor. If these specifications are precise, it gives freedom to various chip manufacturers to implement physical designs that look functionally the same at the ISA level. Thus, if we run the same program on these implementations, we get the same results. Different implementations, however, may differ in performance and price."(Dandamudi, 2005b, p.347) Instructions are encoded by an operation code, followed by operand identifiers (usually two or three). As operand the programmer can choose between a register, integer constants, absolute memory reference and also relative memory addresses. The latter is given by a base register and an offset specified by a register or constant value. However, the permitted number and kinds of operands per instruction can be more or less restrictive, depending on the ISA.

The CPU interprets data in a register and memory as a signed/unsigned integer or bit field data type and different fixed size sub-types are supported (usually 8, 16, 32 up to 64 bit). Floating-point data types and arithmetic are supported by means of software implementation but with low performance. However, an FPU (Floating-point unit) contributes hardware-based floating-point support via hardwired instructions that process floating-point data in a separate register file. Modern CPUs also offer a SIMD (Single Instruction Multiple Data) extension for vector-based operations on packed data types; typically 2, 4, 8 or 16 component vectors of integer and/or floating-point data types are packed in a single (64, 128, 256 up to 512 bit) register on a separate register file. Other implementations of this extension use "register pairing" of existing register files such as SIMD extensions of `SPARC` and `MIPS` processor architectures. Instructions of the core and its extensions are combined in a single

ISA, as depicted in Figure 5.1, so that a stream of machine code comprises the sequential execution of CPU, FPU and SIMD instructions (although they might be executed asynchronously internally). Furthermore, instructions exist for data exchange between the different register files.

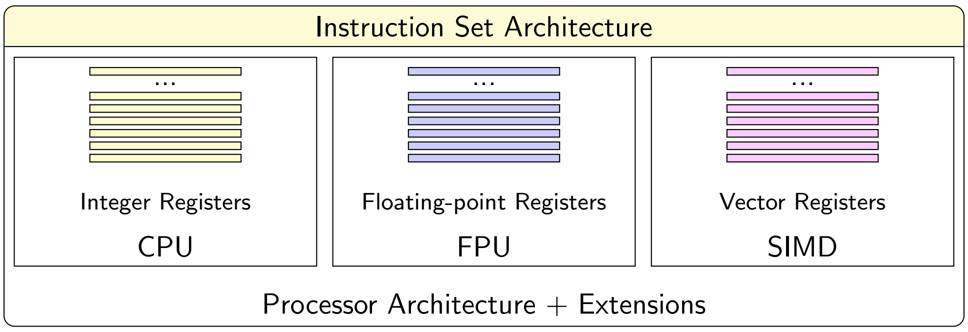


Figure 5.1: Overview of the Instruction Set Architecture.

Modern operating systems virtualize computing resources. The physical memory is shared among multiple applications and the operating system via an MMU (Memory Management Unit) hardware extension. Each process is given a private address space. This allows one to run multiple processes isolated from the operating-system kernel and other processes.

User-space programs are started in a process environment with a main thread of execution that runs the main function of the program. A thread is a virtualization of the core resources of an ISA for executing user-space program code. Thus it has a private copy of CPU, FPU and SIMD user registers, including the PC which gives the current location of code that is being executed. A user-space thread runs for a small amount of time and is interrupted by the kernel for a context switch. The kernel suspends the thread by saving the register content as part of the thread state and exchanges their content with the saved state of another thread that is up next for running a number of CPU cycles. Several threads can be spawned in a single process and each one has a private view on the processor and also a private region of memory named the *Stack*.

5.1.2 Call Stack

In general, programs are not encoded as a single linear block of code that runs sequentially from start to finish. Rather, programs are written as functions and sub-routines that get called. A program has a main entry point function, executed at the start of the process that calls sub-routines and functions, which may call other functions.

Repetitive computations and algorithms are typically encapsulated as functions. The compiler translates a C function, whose activation can occur multiple times, as a block of machine code only once. Its activation can occur multiple times. Function calls are often nested, for example in recursive computations. For the duration of a function call a data record is allocated to provide temporary storage for local variables, arguments and results. Over the course of a program run the required amount of memory grows and shrinks according to the pattern of nested function calls/returns in *last in, first* *out* (LIFO) order. A corresponding memory reservation system can be implemented using the *stack* abstract data type. Memory for arguments/results and local variables can be placed as interleaved memory chunks on a linear vector of memory as depicted in Figure 5.1.

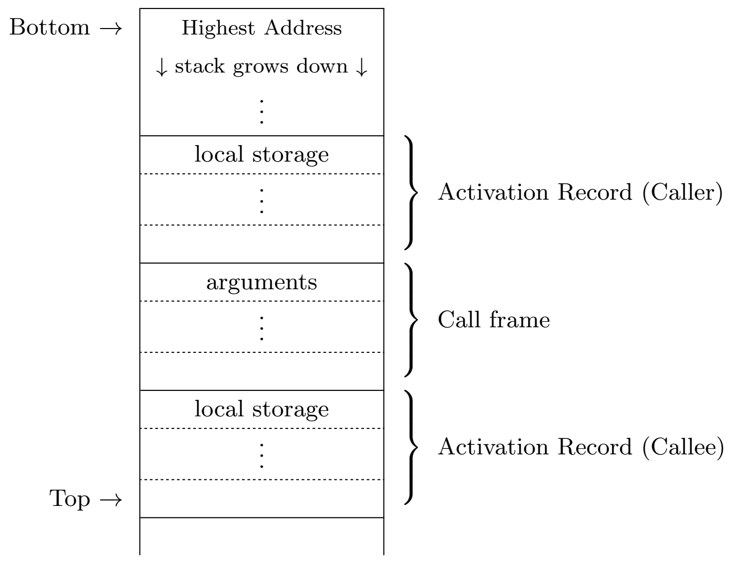


Table 5.1: Illustration of a call stack.

Stacks are elementary structures, simple to implement in machine code by reserving one hardware register as the *stack pointer* (SP) for the top of the stack and by applying read/write and arithmetic instructions for the implementation of `push` and `pop` operations. For each newly created thread of execution a fixed size linear region of stack memory is allocated that is large enough for deeply nested function calls. The bottom of stack is at a fixed address while the *top of stack* grows and shrinks. Although scalar values can be pushed and popped on stacks, call stacks are often organized in *stack frame* structures that are placed or removed as a whole; this can also be implemented by a single addition and subtraction instruction to the SP. The size of each frame is variable depending on the number of arguments (for a caller frame) and local variables (for a callee frame). As an important side effect the stack memory remains defragmented throughout the life-cycle of a program.

Compilers of high-level languages (e.g. C and Pascal) translate functions by using this run-time data model; the generated code for accessing local variables, arguments and results uses *relative addressing* to the SP register so that caller and callee code can be executed dynamically at arbitrary and multiple times. Effectively the call stack works as a generic interface for calling public compiled functions. Given that the order of arguments is known compiled functions can also be called from low-level assembly routines, for example the start-up code for user-space processes, or the caller code of a dynamic generic FFI library.

In general the stack shares the main memory with the *Heap* - a pool of memory for dynamic allocation of memory objects per run-time process. Since applications may have different needs and characteristics regarding dynamic memory and nesting of function calls, operating systems, utilize a memory organization where the heap grows from low to high addresses and the stack grows from high to low addresses. This scheme originates historically from single-threaded systems with a small (virtual) address space. Current multi-threaded 32- and 64-bit operating systems manage multiple threads of execution, each one with its own stack but shared with all other threads in one virtual address space per process. However, the direction of call stacks has still remained on current platforms; they start at the highest address of their memory region and grow to lower addresses.

5.1.3 Calling Sequence

Function calls can be divided in a caller and callee side. Both parts are distinct blocks of machine-code instruction, possibly compiled in separate compilation runs in distinct units and linked to different files (e.g. executable file and dynamically linked shared library).

At the border between caller and callee, a standard compiler generates code that conforms with the language- and platform-specific calling conventions. The latter give exact specifications for the calling sequence i.e. how parameters and return values are passed, and control flow is transferred between caller and callee. “The general architecture dictates how parameters are passed on to the procedures. There are two basic techniques: register-based or stack-based. In the first method, parameters are placed in processor registers and the called procedure reads the parameter values from these registers. In the stack-based method, parameters are pushed onto the stack and the called procedure would have to read them off the stack. The advantage of the register method is that it is faster than the stack method. However, because of the limited number of registers, it imposes a limit on the number of parameters. Furthermore, recursive procedures cannot use the simple register-based mechanism. Because RISC processors tend to have more registers, register-based parameter passing is used in RISC processors. The IA-32 tends to use the stack for parameter passing due to the limited number of processor registers. Some architectures use a register window mechanism that allows a more flexible parameter passing. The SPARC and Intel Itanium processors use this parameter passing mechanism.”(Dandamudi, 2005a, p.28) On architectures with a large number of registers a subset is used as carrier for parameters. Whether registers of hardware extensions, such FPU and SIMD, are incorporated in a calling convention depends on requirements for binary compatibility; designers of calling conventions often make a compromise here between available hardware features and backward compatibility with legacy hardware.

Registers take on one of three possible roles for the entire period of the program run, which is defined by the calling conventions that serve for the register selection of the compiler:

- *Volatile* registers can be used for local variables until a function is called. This includes registers for passing of parameters and return values.
- *Preserved* registers keep their value throughout a function call. If they are used by a function as local storage, their content needs to be saved in advance and restored before returning control flow.

- *System-reserved* registers can not be used by user code and are exclusively used by the operating system for management tasks.

In addition, architectures define a number of registers with special roles:

- The Program Counter (PC) points to the current instruction in machine code.
- The Link Register (LR) is loaded with the return address during a `call` instruction.
- The Stack Pointer (SP) points to the current top of stack that contains the latest frame.
- The Frame Pointer (FP) is a second pointer on the stack that points to the local stack frame.

The stack is organized in cells of same size, similar to the bit size property of the processor architecture. Smaller values (in bits) still occupy a full register and/or stack cell. Placement of values and the layout of data structures is constrained by alignment restrictions of the platform; these are either hardware constrains or result from performance considerations (unaligned memory can decrease performance of the memory bus and cache logic).

A general outline of the calling sequence is given below:

1. A function call is prepared at the caller side by loading parameter arguments to registers and on the stack.
   (a) The first arguments are loaded in volatile registers that are reserved for parameter transfer. Which register class is selected (i.e. general-purpose, FPU or SIMD) depends on the parameter type.
   (b) Remaining arguments are passed on the stack. In general for C, the push order is right-to-left. However, historically Pascal uses the opposite order.
2. The function call is executed by loading the PC with the target address of callee function. In addition, the return address is also transferred via a register or on the stack: On CISC architectures the `call` instruction pushes the return address on the stack. On RISC architectures the `bl` (branch and link) instruction loads the LR with the return address.
3. Functions consist of prolog and epilog code blocks that are executed on entry and exit, respectively. The prolog section of the callee initializes a new activation record for local storage on the stack (by decrementing the SP). Preserved registers that are used by the callee's function block are saved in the activation record. On RISC architectures the content of LR is also saved on the stack; as an optimization, leaf functions (functions that do not call other functions) can omit this step.
4. The main code of the function is executed.

5. The function's return value is passed via volatile registers that are reserved for this purpose. In accordance with parameter transfer, the return value type decides which register class is used. Typically C function return values do fit in one or two registers and are thus passed via registers. However, large objects that do not fit into registers are passed via memory. In that case the caller would allocate the memory storage in advance and the calling sequence would include a hidden first argument for the memory address (in Step 1) in which the callee would write the return value.

6. The function returns by executing the epilog code section:

   (a) Preserved registers are restored.

   (b) The activation record is removed from the stack (via incrementation of the SP).

   (c) The return address is assigned to the PC. RISC architectures `move` the contents of the LR to the PC; CISC architectures use a `ret` instruction that pops the return address from the stack and loads the PC in one step.

   Furthermore, the call's stack frame (containing arguments on the stack) need to be removed. Whether the caller or the callee is responsible for this task depends on the calling convention.

5.1.4 Diversity of Calling Sequences

Details of the calling sequence depend foremost on the processor architecture. However, calling conventions can further vary depending on the operating system as the following example illustrates.

Example We compare the `gcc` compiler's assembly output of a C function call for two different operating systems running on the `x86-64` processor architecture. We consider a function call with a long sequence of heterogeneous types[1]:

```
void samin(int n, double *x, double *Fmin, optimfn fn, int maxit, int tmax, double temp, int trace, void *ex);
```

The calling sequences is *visualized* in the assembly output by tagging each parameter using an ordered sequence of ascending numerical constants:

```
#include "R_ext/Applic.h"
void call_samin() { samin(1, (double*) 2, (double*) 3, (optimfn*) 4, 5, 6, 7.0, 8, (void*) 9); }
```

Listing 17 gives the assembly output generated by the `GCC` compiler for two major binary platforms of the `x86-64` processor architecture, namely the Unix-based System V standard (`sysv`) for Linux, Mac OS X, BSDs and Solaris (*left panel*), and the Microsoft Windows (`x64`) platform (*right panel*). Based on that output we can analyse the storage class and location for each positional argument i.e. whether data is passed in a register or on stack and which register type and number or stack memory offset was chosen by the code generator of the compiler.

[1] `samin` is an optimizing function (simulated annealing) available in the shared C library of R.

```
_call_samin:
  movq  %rsp, %rbp
  subq  $16, %rsp
  movq  $9, 8(%rsp)
  movl  $8, (%rsp)
  movl  $1, %edi
  movl  $2, %esi
  movl  $3, %edx
  movl  $4, %ecx
  movl  $5, %r8d
  movl  $6, %r9d
  movsd LCPI1_0(%rip), %xmm0

  callq _samin
  addq  $16, %rsp
  popq  %rbp
  ret
```

```
call_samin:

  subq    $88, %rsp
  movabsq $4619567317775286272, %rax # bits := 7.0
  movl    $4, %r9d
  movl    $3, %r8d
  movl    $2, %edx
  movl    $1, %ecx
  movq    $9,   64(%rsp)
  movl    $8,   56(%rsp)
  movq    %rax, 48(%rsp)
  movl    $6,   40(%rsp)
  movl    $5,   32(%rsp)
  call    samin
  addq    $88, %rsp

  ret
```

Listing 17: Comparison of assembly: C caller code to `samin` on the `x86-64` processor architecture for the `sysv` (*left* side) and `x64` (*right* side) ABI platform.

The sequence of assembly language instructions is different in each case. Notice the order of instructions for loading arguments on stack and in registers is not strict and can even change depending on compilation flags. Rather, as illustrated in Figure 5.2, it is the contents of the register files and the call stack right before the `call` instruction that is of interest for a comparison of calling sequences.

Linux, Mac OS X, BSD derivates, Solaris

General-purpose

Register	Value
R9	6
R8	5
RCX	4
RDX	3
RSI	2
RDI	1

SIMD

Register	Value
XMM7	-
XMM6	-
XMM5	-
XMM4	-
XMM3	-
XMM2	-
XMM1	-
XMM0	7.0

Stack

Offset	Value
8	9
0	8

Microsoft Windows 64-bit

General-purpose

Register	Value
R9	4
R8	3
RDX	2
RCX	1

SIMD Regs

Register	Value
XMM3	(skip)
XMM2	(skip)
XMM1	(skip)
XMM0	(skip)

Stack

Offset	Value
80	-
72	-
64	9
58	8
48	7.0
40	6
32	5
24	(home)
16	(home)
8	(home)
0	(home)

Figure 5.2: Comparison of general-purpose and SIMD register files and call stack during a C function call to `samin` between `sysv` (*left* side) and `x64` (*right* side) platforms on the `x86-64` processor architecture.

On both platforms a set of general-purpose and SIMD registers are used for the first arguments; further arguments are passed via the stack. However, the `sysv` platform reserves six general-purpose registers and eight SIMD registers for passing arguments while the Microsoft `x64` platform reserves only four registers of both register files. Furthermore there exists an interdependency between general-purpose and SIMD registers on the `x64` ABI; when an argument value is assigned to a general-purpose register a corresponding SIMD register is marked as occupied (labeled “skip”). The `x64` ABI also reserves a fixed size area on the stack, named the *homing area* that has the capacity to save the first four arguments in general-purpose or SIMD registers. In summary, the number of registers for passing arguments can differ significantly across software platforms. Details, such as the homing area and the interdependency effects between register files, need to be taken into account in order to address a generic call facility that works across processor architectures and operating systems.

5.2 Application Binary Interface

Function calls between applications and linked libraries cross compilation units where data need to be transferred in a consistent manner. Whether a simple number is passed or a pointer to complex data records, whose fields need to be precisely read and written by both sides, the underlying C code, separately compiled, needs to be generated in a consistent manner to ensure stable transfer of control- and data-flow i.e. code- and data-level interoperability. The so-called ABI provides the foundation for this consistency. It specifies the rules for a seamless functioning of all components on a given binary software platform, like the gears in an engine block.

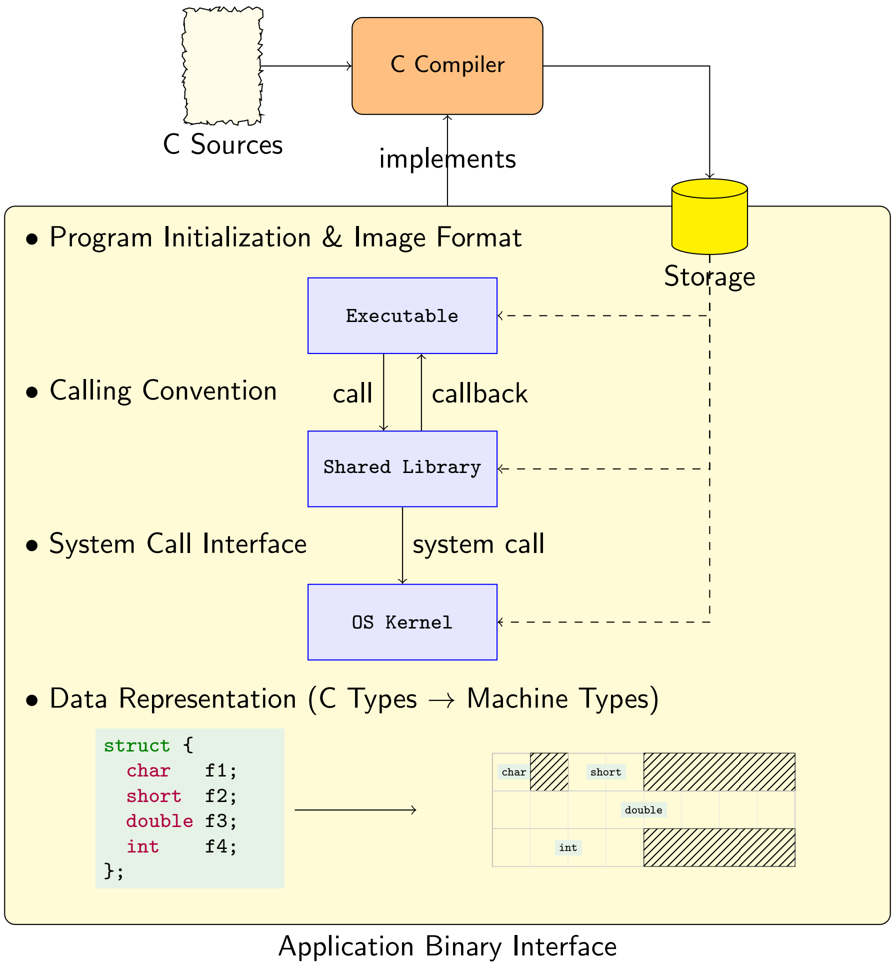
“An Application Binary Interface (ABI) includes a set of conventions that allows a linker to combine separately compiled and assembled elements of a program so that they can be treated as a unit. The ABI defines the binary interfaces between compiled units and the overall layout of application components comprising a single task within an operating system. Therefore, most compilers target an ABI. The requirements and constraints of the ABI relevant to the compiler extend only to the interfaces between shared system elements. For those interfaces totally under the control of the compiler, the compiler writer is free to choose any convention desired, and the proper choice can significantly improve performance.”(Hoxe et al., 1996, p.157)

In general ABI specifications include recommendations for the *implementation-defined* parts of the C standard, including details for the implementation of standard conforming C functions and function calls at machine level, and the data representation of C data objects in memory.

5.2.1 Overview

The interplay between hardware and software, regulated by the ABI, is illustrated in Figure 5.3. A “C Compiler” is responsible for building the system, including the“OS Kernel”, the “Binary Executables” and “Libraries”, from “C sources”. Their run-time object representation is depicted as blue boxes; arrows show run-time interactions such as function calls, callbacks and system calls. Since the code generator of the compiler “implements” the ABI specifications, depicted as the big yellow box, the generated data and code is suitable for code- and data-level interoperability among all components of the system. The rules of the ABI can be roughly divided into four categories:

- “Program Initialization” specifies the required steps for an operating system to create a run-time process from a “Binary Executable” file stored on persistent “Storage”. The file format is specified by an “Image Format” specification and includes the binary layout of “Shared Library”



```
struct {
  char   f1;
  short  f2;
  double f3;
  int    f4;
};
```

Figure 5.3: Anatomy of the Application Binary Interface.

object files and the “OS kernel” image file.

- “Calling Conventions” give details on the calling sequence for code-level interoperability between code modules that make calls to public functions depicted as arrows between “Binary Executable” and “Library”.

- The “System Call” interface is used for calling “OS Kernel” functions from a user-space environment. Typically this interface is used by a “Shared Library” that provides a standard C API such as the I/O functions of the standard C library to call functions of the operating-system kernel for its implementation. Since there is a strict separation between user and kernel space, the mechanism to call system functions is quite different from user-space function calls i.e. typically user-space code has to request an interrupt and passes a service number and parameters via registers. Thus, system calls are usually implemented in assembly language.

- “Data Representation” gives details on the mapping between scalar C data types to machine data types in registers and memory. This includes *size* and *alignment* properties. This also influences the “Calling Convention” since arguments may be passed via the call stack and thus need to be aligned properly.

Designers of native software platforms can freely define an ABI and typically include calling conventions and data type mappings for C. ABIs are specified by standard consortiums (e.g. System V), or designers of processor architectures (e.g. ARM and SPARC Consortium), operating systems (e.g. Apple and Microsoft) or compilers (e.g. `fastcall` calling convention of Watcom, Borland and Microsoft).

The *System V* Unix standard is often used as the base for a large number of Unix-related operating systems. The standard document is divided in two parts:

- The *generic* ABI specification “gABI” (AT&T, 1990) defines the binary standard that is common across all processor hardware platforms. This includes the ELF image and object file format for code and data on persistent storage for executable files and shared object libraries.

- The *processor-specific supplement* part “psABI” specifies the details related to a particular processor architecture hardware and gives details for the implementation of C function calls and mapping of C data types to machine types. System V processor supplements exist for a large number of architectures including `i386`, `AMD64`, `SPARC`, `MIPS`, `PowerPC`, `Motorola 88000` and `Itanium`.

Operating systems, such as Linux, Solaris, BSD derivates and Haiku, usually base their ABI on System V. However, Mac OS X followed System V only recently: “The heart of Mac OS X is the XNU kernel. XNU is basically composed of a Mach core [..] with supplementary features provided by Berkeley Software Distribution (BSD).”(Miller and Zovi, 2009, p.4) “Mach, developed at Carnegie

Mellon University by Rick Rashid and Avie Tevanian, originated as a UNIX-compatible operating system back in 1984."(Miller and Zovi, 2009, p.4) Mac OS X uses the Mach-O file format instead of ELF (as discussed in Section 4.2.3). At first, Mac OS X ran on `PowerPC` processor architectures using an Apple-specific ABI (Apple Inc., 2010b) which is different to the System V ABI for `PowerPC`. As the porting process to the `x86` processor architecture family Mac OS X adopted the System V ABI processor supplement for `x86` processor architectures, including the 32-bit and 64-bit sub-systems. However, for the 32-bit variant small refinements are included, specified in Apple Inc. (2010b, p.43):"

- Different rules for returning structures.
- The stack is 16-byte aligned at the point of function calls.
- Large data types (larger than 4 bytes) are kept at their natural alignment.
- Most floating-point operations are carried out using the SSE unit instead of the x87 FPU, expect when operating on `long double` values."

Although Microsoft Windows on `x86` 32-bit platforms is also very similar to the System V calling conventions, a different image format is used due to the fundamental differences between Windows and Unix. On 64-bit platforms even the calling convention completely differs to all the other Unix-related systems.

While ABIs are designed for longevity, refinements need to be made for optimization purposes to support an extension of a processor architecture such as an FPU with the consequence of breaking backward compatibility, such as in the Mac OS X ABI for `PowerPC` 32-bit: "In Mac OS X v10.4 and later and GCC 4.0 and later, the size of long double extended precision data types is 16 bytes (it's made up of two 8-byte doubles). In earlier versions of Mac OS X and GCC, long double is equivalent to double. You should not use the long double type when you use GCC 4.0 or later to develop or or in programs targeted at Mac OS X versions earlier than 10.4."(Apple Inc., 2010b, p.10)

5.3 Processor Architecture Families

The development of an abstraction interface to function calls and its implementation on a target platform requires a deeper study of the different machine interfaces (i.e. the ISA) and the functioning of their call mechanisms (i.e. the ABI and calling conventions). As a preparation for a discusson on the implementation of `dyncall` and `dyncallback`, we briefly introduce the following five processor architecture families.

- X86 architecture family (`x86`).
- ARM architecture family (`arm`).

- PowerPC architecture family (`ppc`).
- MIPS architecture family (`mips`).
- SPARC architecture family (`sparc`).

We use a common abbreviation and naming scheme for refering to a particular processor architecture family as given in parenthesis the above. For a particular architecture the bit size (32 or 64) is appended. Specific to the `x86`, a dash ('`-`') separates the family name and bit number. Furthermore, when we refer to a specific ABI, such as System V abbreviated `sysv`, it is appended separated by a dash i.e. `x86-32-sysv` refers to the System V ABI on the `x86` 32-bit platform. Table 5.2 gives an overview of the five architecture familes and available bit sizes, their class of architecture and the number of available general-purpose registers (GPRs) and floating-point registers (FPRs).

Family	Architecture		Number of Registers		
	Category	Bits	GPRs	FPRs	Floating-point formats
`x86`	CISC	16/32	8	8	80-bit extended precision (`x87`)
		64	16	16	32/64-bit single/double precision (`SSE`)
`arm`	RISC	32/64	16	32	32-bit single precision or 16 × 64-bit double precision (overlapped) (`VFP`)
`ppc`		32/64	32	32	64-bit double precision
`mips`		32/64	32	32	32/64-bit single/double precision or 128-bit quad precision (overlapped)
`sparc`		32/64	32	32	32/64-bit single/double precision or 128-bit quad precision (overlapped)

Table 5.2: Overview of Processor Architecture Families, bit architectures, number of general-purpose registers (GPRs) and floating-point registers (FPRs) and supported floating-point formats.

An overview of processor architectures and ABIs to be discussed in the following sections is given in Table 5.3.

5.3.1 From CISC to RISC

In the early years of computing the design of a processor chip was mainly characterized by the complexity of the instruction set. "Popular processor designs can be broadly divided into two categories: Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC). The dominant processor in the PC market, Pentium, belongs to the CISC category. However, the recent trend is to use the RISC designs. Even Intel has moved from CISC to RISC design for their 64-bit processor."(Dandamudi, 2005a, p.vii)

"In the 1970s and early 1980s, processors predominantly followed the CISC designs. [..] Several factors contributed to the popularity of CISC in the 1970s. In those days, memory was very expensive and small in capacity."(Dandamudi, 2005a, p.5) "The motivation for designing such complex instruction sets is to provide an instruction set that closely supports the operations and data structures used by Higher-Level Languages (HLLs). [..] The evolution of CISC designs can be attributed to the desire of early designers to efficiently use two of the most expensive resources, memory and processor, in a

Family	Bits	ABI	Variants	Notes (Languages, OSs and Compilers)
x86	32	`cdecl`		Standard C calling convention (compatible with System V, GNU, Microsoft)
		`stdcall`		Standard call for System DLLs on Microsoft Windows
		`fastcall`	`ms`	Fastcall Microsoft compiler specific
			`gnu`	Fastcall GNU compiler specific
		`thiscall`	`ms`	C++ member function calls using Microsoft compiler
			`gnu`	C++ member function calls using GNU compiler
		`plan9`		Plan9 calling convention on i386
		`syscall`	`linux`	System call on Linux (and `dos`)
			`bsd`	System call on BSDs
	64	`sysv`		System V ABI
		`x64`		Microsoft Windows 64-bit calling convention
`ppc`	32	`sysv`		System V ABI
		`osx`		Mac OS X Calling Convention
`arm`	32	`aapcs`	`oabi`	Old Linux ABI (hybrid FPU utilization / very slow)
			`eabi`	Standard ABI (soft-float i.e. no FP registers)
			`armhf`	Standard ABI (hard-float)
`mips`	32	`o32`		System V ABI for 32-bit (Old ABI)
		`eabi`		Embedded ABI for Playstation Portable (Homebrew)
	64	`n64`		System V ABI for 64-bit (New ABI)
		`n32`		32-bit mode on 64-bit (New ABI)
`sparc`	32	`v7`		System V ABI
	64	`v9`		System V ABI

Table 5.3: Families, Architectures, ABIs and Calling Conventions.

computer system. In the early days of computing, memory was very expensive and small in capacity. This forced the designers to devise high-density code: that is, each instruction should do more work so that the total program size could be reduced. Because instructions are implemented in hardware, this goal could not be achieved until the late 1950s due to implementation complexity."(Dandamudi, 2005a, p.39-40)

"The decision of CISC designers to provide a variety of addressing modes leads to variable- length instructions. For example, instruction length increases if an operand is in memory as opposed to in a register. For these and other reasons, in the early 1980s, designers started looking at simple ISAs. Since these ISAs tend to produce instruction sets with far fewer instructions, they coined the term Reduced Instruction Set Computers (RISC). Even though the main goal was not to reduce the number of instructions, but rather the complexity, the term has stuck. SPARC, PowerPC, MIPS, and Itanium are all examples of RISC designs."(Dandamudi, 2005b, p.348)

"The design of Reduced Instruction Set Processors (RISC) began in earnest in the early 1980s. Early RISC processors typically were characterized by a load-store architecture, single instruction-per-cycle execution, and 32-bit addressing. The instruction set architecture of these early RISC chips was well matched to the level of computer optimization available in the early 1980s, and provided a minimal interface for the UNIX(TM) operating system."(Weaver and Germond, 1994, xiii)

5.3.2 X86 Processor Architecture Family

x86 is one of the oldest processor architecture families that is still widely use in current notebook, desktop and server systems. In the late-1970s Intel released the 16-bit *8086/8088* processors. The ISA was extended to 32-bit with Intel's *80386* CPU core in 1985. Other companies, such as AMD, VIA, IBM, Texas Instruments, Cyrix and National Semiconductor, manufactured processor chips that were binary compatibility with the x86-16/x86-32 ABI of Intel's *80386*. Due to its CISC design, the ISA comprises a large number of instructions. However, only eight registers are available for general-purpose, and of those, some were hardwired for special purposes such as the stack pointer register ESP and corresponding stack manipulation instructions: push and pop.

Hardware-based floating-point support was introduced with the x87 FPU co-processor, which uses a register file of eight registers of 80-bit precision. Later, the co-processor was integrated as part of the *80486* CPU core. With the era of graphics multimedia and video games on standard PC hardware in the 1990s, Intel designed a SIMD extension which introduced native vector-based data types and operations. At first the ISA was extended by integer-only vector operations and eight 64-bit registers, primary designed for graphics pixel processing effects with support for 8×8-bit, 4×16-bit or 2×32-bit packed (unsigned) integer data types. The extension was branded as MMX. Soon afterwards AMD released a similar extension named '3Dnow!'. Effectively MMX registers are shared with those of the x87 FPU so that x87 and MMX operations need to be synchronized for mode switching. Later a modern SIMD extension, named SSE (SIMD Streaming Extension),was introduced that supports floating-point vector operations on a separate set of eight 128-bit vector registers (which is not shared with registers of the x87 FPU). Several revisions followed, such as SSE2, SSE3, SSE4 and so on. AMD's implementation of SSE2 doubled the number of the SSE registers to sixteen (XMM0-15).

With the specification of a 64-bit ISA extension for x86, published by AMD in 2000, the number of general-purpose registers was doubled to sixteen registers and the SSE2 extension was integrated. Major vendors, such as VIA and Intel, adopted the 64-bit ISA standard; the extension became known as x86-64 and other acronyms such as amd64, x86_64, IA-32e, Intel64 and x64. Ever since, backward compatibility has remained one of the major features of the x86 ISA; even modern 64-bit x86-64 CPU cores are still capable of executing legacy 16-bit and 32-bit code.

5.3.2.1 ABIs and Calling Conventions of x86

The list of binary interfaces available for x86-16 and x86-32 gives a prime example of the diversity of binary software standards for a single processor architecture; diversity decreased with the x86-64 architecture as depicted in Table 5.4.

Presumably the limited number of registers and the lower clock-speed of x86 cores available between

Architecture	Calling Convention	Argument Registers	Stack Push order	Stack Cleanup
16	`cdecl`	-	rtl	caller
	`pascal`	-	ltr	callee
	`fastcall-ms`	AX,DX,BX	ltr	callee
	`fastcall-borland`	AX,DX,BX	ltr	callee
	`fastcall-watcom`	AX,DX,BX,CX	rtl	callee
32	`cdecl`	-	rtl	caller
	`stdcall`	-	rtl	callee
	`fastcall-ms`	ECX,EDX	rtl	callee
	`fastcall-gnu`	ECX,EDX	rtl	callee
	`fastcall-borland`	EAX,EDX,ECX	ltr	callee
	`fastcall-watcom`	EAX,EDX,EBX,ECX	ltr	callee
	`thiscall-gcc`	-	rtl	caller
	`thiscall-ms`	ECX	rtl	callee
	`plan9`	-	rtl	caller
	`syscall-dos/linux`	EAX,EBX,ECX,EDX,ESI,EDI	-	-
	`syscall-bsd`	EAX	rtl	caller
64	`sysv`	RDI,RSI,RDX,RCX,R8-9,XMM0-7	rtl	caller
	`x64`	RCX,RDX,R8-9, XMM0-3	rtl	caller

Table 5.4: Some calling Conventions of the `x86` architecture. 'rtl' and 'ltr' refers to the stack push order of arguments from 'right-to-left' or 'left-to-right', respectively.

1980s and 90s encouraged compiler and platform vendors to develope different calling conventions to optimize overall system performance.

In order to compare calling conventions at register level for each architecture, we use register charts, such as depicted in Figure 5.4; a description of the notation is given in the box below:

Register Charts A set of registers is displayed as cells in rows. The base color indicates the register class, such as general-purpose (yellow), FPU (blue) and SIMD (magenta), respectively. The intensity of the cell color indicates whether a register is volatile (*light color*) or preserved (*dark color*) during a function call. The order of registers for passing parameters and results is given by indexed labels a_{index} and r_{index}, respectively. Special-purpose registers are labels SP (stack pointer), LR (link register), FP (frame pointer), IP (intra-procedure pointer), TR (thread register), SB (static base) and PC (program counter). Furthermore, system-reserved and constant zero (labeled "0") registers are colored red. When comparing multiple calling conventions, the labels are given on the left of the register bar. Additional information, such as register numbers and names, grouping structure, and overlapping register layout schemes, are given above and below the bar.

The register utilization for some `x86` 32- and 64-bit ABIs are depicted in Figure 5.4. The first eight general-purpose registers, available since the first `x86` 16-bit architecture, are given symbolic names such as EAX, ESI and ESP. The *accumulator*, *base index*, *counter* and *data* registers are abbreviated by AX, BX, CX and DX, respectively. *Source index*, *destination index*, *stack pointer* and *base pointer* registers are abreviated SI, DI, SP and BP, respectively. The upper eight registers, introduced with the `x86-64` architecture, are specified by numeric names (R8-R15). As the lower eight registers are available in 16-bit mode, they are prefixed by E and R in 32-bit and 64-bit mode, respectively. Registers of the `x87` FPU, named ST(0) to ST(7), are organized and programmed in a stack-based manner; they are only part of calling conventions for passing scalar floating-point return values.

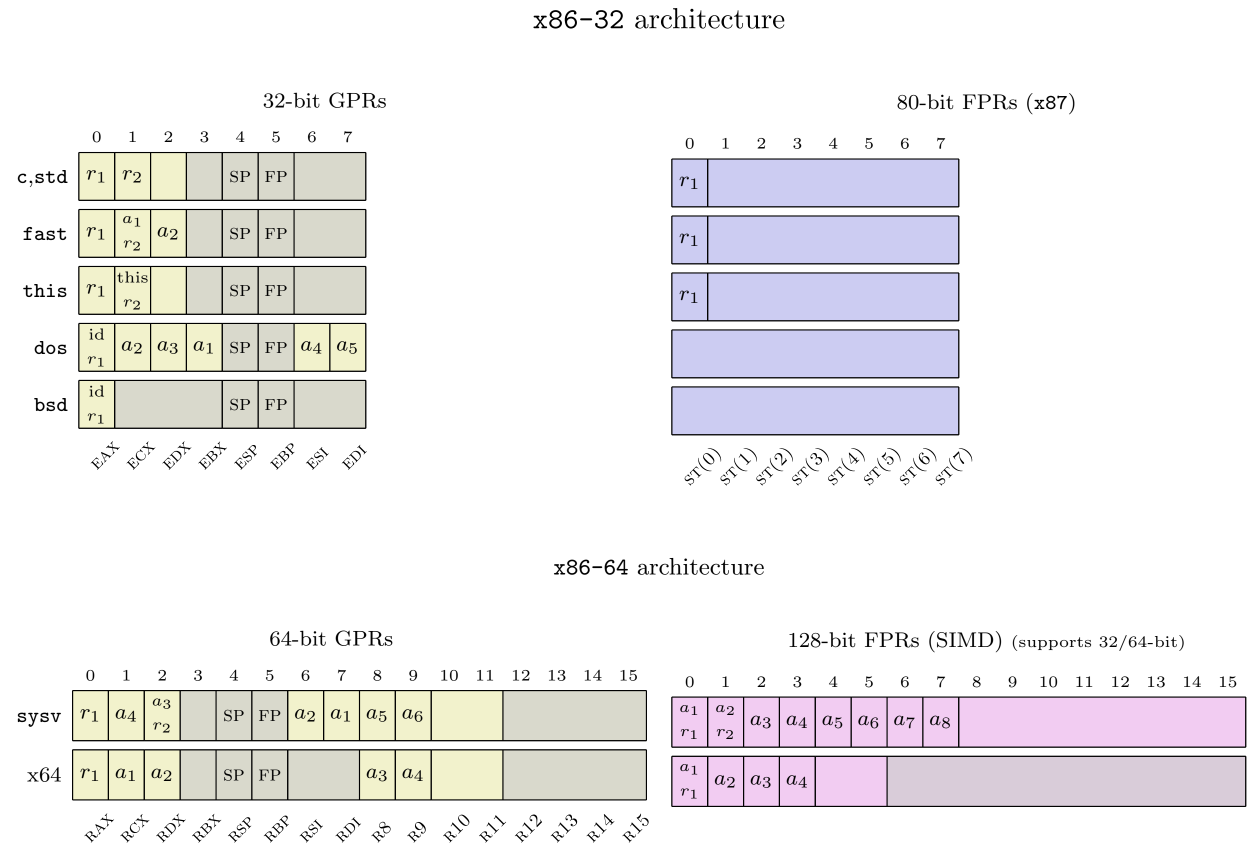

x86-32 architecture
32-bit GPRs
80-bit FPRs (x87)
c,std
fast
this
dos
bsd
SP
FP
this
id
EAX ECX EDX EBX ESP EBP ESI EDI
ST(0) ST(1) ST(2) ST(3) ST(4) ST(5) ST(6) ST(7)
x86-64 architecture
64-bit GPRs
128-bit FPRs (SIMD) (supports 32/64-bit)
sysv
x64
RAX RCX RDX RBX RSP RBP RSI RDI R8 R9 R10 R11 R12 R13 R14 R15

Figure 5.4: Register usage in calling conventions of `x86`.

The *C calling convention* (depicted as "`c`" in Figure 5.4), implemented on `x86`, uses a simple scheme: all arguments are passed via the stack and return values are passed via registers. Small integer values (up to 32 bits) are placed in EAX, 64-bit integer values are placed in the register pair ECX:EAX, and floating-point values, such as `float`, `double` and `long double`, are returned via ST(0). All other `x87` floating-point registers are volatile and need to be empty when a function returns.

This calling convention is commonly found on major `x86-32` platforms for C programs (Windows) or for the entire user-space system (Unix) with slight modifications.

- On the Microsoft's Windows 32-bit platforms, it is commonly named the `cdecl` calling convention.
- `x86-32` ports of Unix-based commercial, free and open-source operating systems are mostly

based on the *System V ABI Intel386 Architecture Processor Supplement* (AT&T, 1991) which defines the C calling convention as the base in their ABI.

- Apple's Mac OS X ABI for `x86-32` (Apple Inc., 2010b, pp.43) is mainly based on System V with small modifications as outlined in Section 5.2.

A peculiarity of the Microsoft Windows is the `stdcall` calling convention, used for the public C interface of system component DLLs. In principle it functions similiarly to the C calling convention. However, the stack cleanup strategy is different. While the caller is usually responsible for removing all arguments on the stack, it is the callee in this case.

Major C compilers of `x86-32` offer a compiler-specific `fastcall` calling convention; the name refers to an optimized calling convention which utilized a subset of the eight GPRs; an overview of 16-bit and 32-bit variants of `fastcall` is listed in Table 5.4. Figure 5.4 gives the register utilization for `fastcall` on `x86-32`, supported by Microsoft and GNU compilers (displayed as "`fast`" in Figure 5.4). Both calling conventions make use of registers ECX and EDX to pass the first two integer arguments.

C and *Pascal* calling conventions on `x86` differ by the order in which arguments are pushed on the stack. In C arguments are pushed in *right-to-left* order. Consequently, since the stack grows downwards, the sequence of argument data is placed upwards on stack memory, similar to fields of a C data structure, in ascending *left-to-right* order. The majority of calling conventions for `x86` and other processor architecture familes use this scheme. Exceptions to this include the Pascal calling convention (`pascal`) and a few `fastcall` variants for `x86-16/32` that use the opposite order.

Support for calling C++ member functions necessitates passing a reference of a C++ object. This can be transparently implemented by means of the C calling convention; the instance `this` pointer is passed as a first argument of the calling sequence. However, the `thiscall` calling convention is used for C++ member function calls on Microsoft Windows `x86-32` platforms; the instance pointer is passed via the register ECX (depicted as "`this`" in Figure 5.4) and the callee is responsible for cleaning up the stack.

Although most Unix-based operating systems on `x86-32`, such as Linux and BSD derivates, are based on System V and utilize the ELF format for binary executable and shared library formats (as discussed in Section 4.2), there exist different calling conventions for system calls. In general, system functions are called by execution of an interrupt; the function is specified by passing a service number in a register. On Linux all arguments are also passed via registers while FreeBSD arguments are passed via the stack.

The doubling of general-purpose registers as of the `x86-64` ISA led to a RISC-based design of calling conventions for `x86`. Two ABI standards cover a majority of 64-bit `x86` platforms. Both standards make use of the larger number of available registers and also utilize `SSE` registers for transfer of floating-point parameters and results. The *System V Application Binary Interface: AMD64 Architecture Processor Supplement* (Matz et al., 2012) specifies a calling convention for C (denoted "`sysv`" in Figure

5.4) that was adopted by Linux, Solaris, Mac OS X and BSD operating systems. Microsoft defines its own calling convention for Windows 64-bit platforms (Microsoft, 2013), namely "x64 Software Conventions" (denoted "`x64`" in the illustration in Figure 5.4). Microsoft's ABI uses fewer registers for passing arguments as illustrated in the example of Section 5.1.4. Note also that `x64` uses one register for return values while `sysv` uses up to two registers. For further details on calling conventions of the `x86` architecture family see Fog (2012).

5.3.3 ARM Processor Architecture Family

The ARM (Advanced Risc Machines) processor family was originally designed as a 32-bit RISC architecture and its origin dates back to the mid-1980s when it was first used in the Acorn (Archimedes) personal computers. "Acorn had developed a strong position in the UK personal computer market due to the success of the BBC (British Broadcasting Corporation) microcomputer" (Turber, 2000, p. 36). "The first ARM processor was developed at Acorn Computers Limited, of Cambridge, England, between October 1983 and April 1985. At that time, and until the formation of Advanced RISC Machines Limited (which later was renamed simply ARM Limited) [..], ARM stood for Acorn RISC Machine" (Turber, 2000, p. 36). "At the time the first ARM chip was designed, the only examples of RISC architectures were the Berkeley RISC I and II and the Stanford MIPS (which stands for Microprocessor without Interlocking Pipeline Stages), although some earlier machines such as the Digital PDP-8, the Cray-1 and the IBM 801, which predated the RISC concept, shared many of the characteristics which later came to be associated with RISCs" (Turber, 2000, p. 37).

The company "ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee. Subsequently, additional companies, including Texas Instruments, NEC, Sharp, and ST Microelectronics, licensed the ARM processor designs, extending the applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems, and many other consumer products."(Yiu, 2010, p. 2)

We can assume that ARM will take on an important role in the CPU market in the coming years. In fact, it is already dominating the embedded, mobile and handheld market. "ARMs designers have come a long way from the first ARM1 prototype in 1985. Over one billion ARM processors had been shipped worldwide by the end of 2001."(Sloss et al., 2004, p. 3) "There were 20 billion ARM cores used by 2002. Nowadays, ARM processors are almost in everybodys pocket because almost all of the mobile phones, PDAs are developed based on ARM cores."(Muresan, 2005, p. 6) "Nowadays [2010], ARM partners ship in excess of 2 billion ARM processors each year. Unlike many semiconductor companies, ARM does not manufacture processors or sell the chips directly. Instead, ARM licenses the processor designs to business partners, including a majority of the worlds leading semiconductor companies. Based on the ARM low-cost and power-efficient processor designs, these partners create their processors, microcontrollers, and system-on-chip solutions. This business model is commonly called intellectual property (IP) licensing."(Yiu, 2010, p. 2) "The ARM company bases their success on a simple and powerful original design, which continues to improve today through constant technical innovation. In fact, the ARM core is not a single core, but a whole family of designs sharing similar design principles and a common instruction set."(Sloss et al., 2004, p. 3) See Table 5.5 for a sample selection of hardware systems of the past two decades, ranging from home computers, mobile cell phones, handheld game consoles and tablet PCs, that incorporated one or more ARM cores and/or licensed ARM IP in their System-On-Chip design.

					CPU		SoC
Year	ISA	Features and Extensions	Sample Products		Family	Core	Package
1985	ARMv1	26-bit address space	Prototyp	(1985)	ARM1	*ARM1*	
1986	ARMv2	Multiply instruction	BBC Archimedes 305	(1987)	ARM2	*ARM2*	
1990	ARMv3	32-bit address space	Acorn A5000	(1991)	ARM3	*ARM3*	
1993	ARMv4	32-bit address space only System mode	Apple Newton 100	(1993)	ARM6	*ARM610*	
			3DO	(1993)		*ARM60*	
			StrongARM RiscPC	(1996)	StrongARM	*SA-110*	
			Sharp Zaurus SL-5500	(2002)		*SA-1110*	
1994	ARMv4T	16-bit ISA Thumb	Nokia 6110	(1997)	ARM7	*ARM7TDMI*	
			Apple iPod	(2001)			
			Nintendo GBA	(2001)			
2002	ARMv5TE	ARM/Thumb interworking	Nintendo DS	(2002)	ARM9	*ARM946E-S*	
		VFPv2 FPU, DSP	Marvell SheevaPlug	(2009)		*Marvell Kirkwood 88F6281*	
	ARMv5TEJ	Java Bytecode Execution	RIM BlackBerry Quark	(2003)	ARM7	*ARM7EJ-S*	
2007	ARMv6	16/32-bit ISA Thumb-2 Enhanced SIMD DSP	Apple iPhone	(2007)	ARM11	*ARM1176JZF-S*	*SSL8900*
			Raspberry Pi	(2012)			*BCM 2825*
			HTC Dream	(2008)		*ARM1136EJ-S*	*MSM7201A*
			Samsung Galaxy i7500	(2009)			*MSM7200A*
2008	ARMv7-A	Advanced SIMD NEON VFPv3 FPU VFPv4 FPU	TI BeagleBoard	(2008)	Cortex-A	*Cortex-A8*	*OMAP3530*
			Apple iPhone 3GS	(2009)			*S5PC100*
			Genesi Efika MX	(2009)			*i.MX515*
			Apple iPad 1	(2010)			*Apple A4*
			Samsung Galaxy Note	(2011)		*Cortex-A9*	*Exynos 4*
			Microsoft Surface RT	(2012)			*Tegra 3*
			Apple iPhone 4S	(2011)			*Apple A5*
			Apple iPhone 5	(2012)		*Cortex-A15*	*Apple A6*
2012	ARMv8-A	64-bit Architecture	Apple iPhone 5S	(2013)	Cortex-A50	*Cortex-A53* *Cortex-A57*	*Apple A7*

Table 5.5: Overview of the ARM Architecture family. (See also Sloss et al. (2004, p.39, Table 2.7))

Recently, ARM released a 64-bit extension, which leads to the assumption that ARM cores may be a future alternative to the dominating `x86` architecture for notebook, desktop and server systems: AMD announced "to release its first ARM-based chip in 2014 [to follow] in the steps of Dell and HP which have previously announced plans to build ARM-based servers."(BBC News, 2012)

5.3.3.1 ARM Architecture and Revisions

"The ARM core uses a RISC architecture. RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed. The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware. As a result, a RISC design places greater demands on the compiler. In contrast, the traditional complex instruction set computer (CISC) relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated."(Sloss et al., 2004, p. 4)

"An ARM processor is an implementation of a specific instruction set architecture (ISA). The ISA has been continuously improved from the first ARM processor design. Processors are grouped into implementation families (ARM7, ARM9, ARM10, and ARM11) with similar characteristics."(Sloss et al., 2004, p. 44) "The processor family is a group of processor implementations that share the same hardware characteristics. For example, the *ARM7TDMI*, *ARM740T*, and *ARM720T* all share the same family characteristics and belong to the *ARM7* family."(Sloss et al., 2004, p.38)

At the core of the CPU 16 general-purpose 32-bit registers are available for integer/bit arithmetic, load/store operations and control flow. Over time a number of extensions have been designed for ARM. See Table 5.5 for a chronological list of ISA revisions and extensions.

The address space of the first ARM architecture was limited to 26 bits. Support for a 32-bit address space was later added with ARMv3. ARMv4 introduced a privileged system mode for handling exceptions.

The instructions of ARM are encoded as 32-bit words, which is very typical for 32-bit RISC architectures. However, a significant ISA extension for the embedded and mobile market was added with ARMv4T: "Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space. Since Thumb has higher performance than ARM on a processor with a 16-bit data bus, but lower performance than ARM on a 32-bit data bus, [it is suggested to] use Thumb for memory-constrained systems. Thumb has higher code density - the space taken up in memory by an executable program - than ARM. For memory-constrained embedded systems, for example, mobile phones and PDAs, code density is very important. Cost pressures also limit memory size, width, and speed. On average, a Thumb implementation of the same code takes up around 30% less memory than the equivalent ARM implementation.[...] Each Thumb instruction is related to a 32-bit ARM instruction."(Sloss et al., 2004, p. 87) "In Thumb state, you do not have direct access to all registers. Only the low registers R0 to R7 are fully accessible."(Sloss et al., 2004, p.89)

From then on developers needed to handle two separate machine-code instructions and corresponding decoding 'modes' of the CPU. Mode switches between Thumb and ARM can be intermixed in a single code execution flow. The ARMv5 architecture improved interworking between ARM and Thumb mode switching.

Note the naming schemes for ARM architectures and cores, as depicted in Table 5.5, looks confusingly similar and can lead to wrong assumptions, e.g. the ***ARM6*** processor and the `ARMv6` architecture are not related; ***ARM6*** cores implement the `ARMv4` ISA. The following two tables give the notation syntax for the ARM architecture (*left*) and processor (*right*).

"`ARMv`" ⟨*number*⟩ ⟨*letter*⟩...

letter	Description
'`T`'	Thumb: 16-bit compressed ISA
'`E`'	Enhanced DSP
'`J`'	Jazelle: Java byte-code execution

"`ARM`" ⟨*number*⟩ ⟨*letter*⟩...

letter	Profile
'`T`'	Application
'`D`'	JTag debugging
'`M`'	Fast multiplier
'`I`'	EmbeddedICE macrocell
'`E`'	Enhanced instructions (assumes TDMI)

The suffix letter codes, such as '`T`', '`E`' and '`J`', indicate a certain feature of the architecture or CPU core. Later revisions of the ARM architecture adopted this feature set completely. So that `Thumb`, encoded as '`T`' till `ARMv5`, is a core feature for `ARMv6` and `ARMv7-A` architectures. Similarly, "all ARM cores after the ***ARM7TDMI*** include the `TDMI` features even though they may not include those letters after the 'ARM' label."(Sloss et al., 2004, p.38)

ARM has support for FPU co-processor extensions, such as `VFPv1`, `VFPv2`, `VFPv2-D32`, `VFPv3` and `VFPv4`. New SIMD operations were also added to ARM with the `ARMv6` architecture, which was later enhanced with the `NEON` extension for the `ARMv7-A` architecture.

Owing to the success of the `Thumb` ISA, a new mixed-mode ISA, named `Thumb-2`, was added that integrates `ARM` 32-bit and `Thumb` 16-bit machine-code encoding. "The `Thumb-2` technology extended the Thumb Instruction Set Architecture (ISA) into a highly efficient and powerful instruction set that delivers significant benefits in terms of ease of use, code size, and performance. The extended instruction set in `Thumb-2` is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions. It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between `ARM` state and `Thumb` state."(Yiu, 2010, p. 8)

Since `ARMv7` the architecture has been split in three profiles, each with a different focus of use case:

"`ARMv`" ⟨*number*⟩ '`-`' ⟨*letter*⟩

letter	Profile
'`A`'	Application
'`R`'	Realtime
'`M`'	Microcontroller

The `ARMv8-A` *Application* profile is currently the latest version of the architecture. It introduces a new 64-bit execution environment named `AArch64` and a new 64-bit ISA named `A64`; the 32-bit execution environment is still supported and referd to as `AArch32`. The number of registers was almost doubled offering 31 general-purpose registers and a 48-bit address space is used for a larger virtual address space where the `LP64` and `LLP64` are the primary data models A detailed discussion on 64-bit data models is given in Section 5.3.8. The `AArch32` environment supports `ARM` and `Thumb` that are referred to as `A32` and `T32`, respectively. However, the architecture is relatively new so that we focus in this

ABI	Note
`apcs`	ARM Procedure Call Standard (`OABI`)
`iwmmxt`	Support for Intel XScale MMX extensions
`tpcs`	Thumb Procedure Call Standard
`atpcs`	ARM-Thumb ABI (precursor to `aapcs`)
`apcs-gnu`	Legacy ABI for `arm32` on Linux
`aapcs`	ARM Architecture Procedure Call Standard (`EABI`)
`aapcs-linux`	`EABI` using 32-bit integer enums
`armhf`	`EABI` using floating-point registers
`ios-v6`	Apple iOS ABI for ARMv6 architecture based on `aapcs`
`ios-v7`	Apple iOS ABI for ARMv7 architecture based on `aapcs`

Table 5.6: ABIs of ARM processor architectures.

thesis on 32-bit architectures `ARMv4` to `ARMv7-A`.

5.3.3.2 ABIs and Calling Conventions of `arm`

"On the ARM architecture, there are two major ABI types to choose from: `EABI` and `OABI`. There is also a `Thumb` ABI and an Intel `IWMMX` specific ABI but these are generally not recommended for most uses. The `EABI` (Embedded ABI) is newer and supports additional features, faster software floating point operations, and Thumb interworking, but is only compatible with `ARMv4t` and newer cores. The EABI has sub-ABIs of: `aapcs-linux` and `aapcs`. `aapcs-linux` has standard Linux 4 byte enums while `aapcs` has variable length enums. `aapcs-linux` is recommended over `aapcs`. The `OABI` (old ABI) is called `apcs-gnu` and supports `ARMv4` and older cores. Generally the `OABI` is not used by modern ARM processors."(Ciccone et al., 2012, sec 6.3) "The `atpcs` is a precursor to the `aapcs` while the `apcs` and `tpcs` ABIs are considered obsolete."(ARM Limited, 2009, p.6) Table 5.6 gives an overview of some ARM ABIs including legacy standards. Despite the large number of ARM-based hardware platforms the total number of available ABIs is relatively small; several ABIs contain mostly small modifications. Today most platforms have converged to the *Procedure Call Standard for the ARM Architecture*(ARM Limited, 2009), abbreviated `aapcs`, which specifies the data types and alignments, the base procedure call standard, standard variants and C/C++ language mappings.

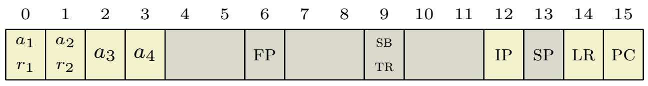


Figure 5.5: Register files on the `arm` processor architecture.

Figure 5.5 gives an overview of the utilization of general-purpose registers for procedure calls in the `aapcs` base standard. The first four registers, R0-R3, are used for passing argument values. R0-R1 are used for passing return values. Further values are passed via the stack managed by the stack pointer register R13 (SP). Register R14 and R15 are used as link register (LR) and program counter (PC), respectively. R12 is a scratch register that can also be used transparently by the linker as an

Intra-Procedure-call register (IP).

aapcs provides a flexible framework that addresses different hardware profiles. For example, R9 is defined to be a platform-specific register; it can be used as a register for *thread-local* storage (TR) or as a *static base* register (SB) in a position-independent data model, depending on the needs of the target platform. aapcs provides a base calling convention which does not require the existance of a FPU; floating-point argument/return values are passed via integer registers and the stack, even if the architecture supports a FPU co-processor. However, aapcs includes recommendations to support FPU co-processor registers and, more recently, a new ABI, named armhf (ARM hard-float), was specified for Debian GNU/Linux on ARM systems (Debian, 2012) as an optimization for architectures with a VFPv2 (or higher) FPU co-processor such as provided by ARMv6 and ARMv7-A CPU cores.

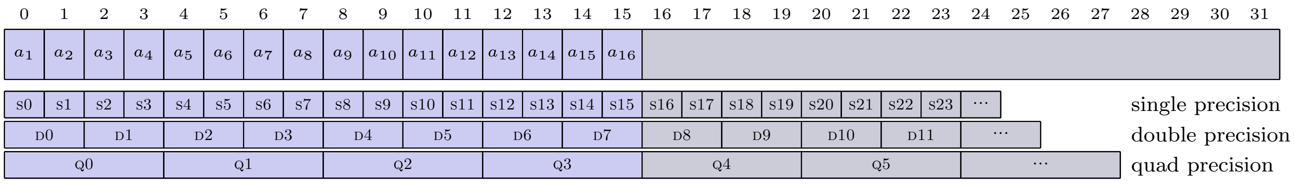

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15} a_{16}
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17 S18 S19 S20 S21 S22 S23 ···
single precision
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 ···
double precision
Q0 Q1 Q2 Q3 Q4 Q5 ···
quad precision

Figure 5.6: FPU registers of the VFPv2 co-processor of arm and utilization in armhf ABI.

The VFPv2 FPU comprises 32 registers for single-precision floating-point values, named S0 to S31, as depicted in Figure 5.6. The armhf ABI utilizes the first 16 registers for passing floating-point values. Note that double-precision floating-point values are also supported by VFPv2, via register pairing, and these can be addressed in assembly via D0-D15. Thus up to eight double floating-point values are passed via FPU registers before being passed via the stack. Similarly, quad-precision floating-point in hardware is also supported via register pairing as of the NEON SIMD extension.

Since ARMv4T the ARM architecture offers the 16-bit Thumb (and later Thumb-2) machine-code mode besides the standard 32-bit ARM mode where both modes can co-exist within a single program code and running process. For a portable implementation of generic function calls at machine level that cover different ARM architecture versions, both ISA modes need to be considered. Note that the roles of registers and the layout of the stack remains identical in both modes. However, special care is needed for handling the so-called *interworking* when calling functions that require a mode switch between ARM and Thumb. The latter can take place during a *branch* code, including function returns. The target mode is specified by bit 0 of the code address of a target branch; ARM compiled code addresses are as-is while Thumb compiled code have odd addreses where bit 0 is set to 1 although the actual code still begins at even addresses. Depending on the ARM architecture to be supported different *interworking* techniques for branch instructions (via a code pointer) need to be considered. For details see ARM Limited (2009, sec 5.6).

The aapcs base standard has been widely adopted or refined. For example, the ABI for Apple's ARM-based iOS platform (used in iPod, iPhone and iPad mobile devices) is based on the core aapcs (without FPU registers) and includes small refinements: "The function call calling convention used in the ARMv6 environment are the same as those used in the Procedure Call Standard for the ARM Architecture (release 1.07), with the following exceptions:

- The stack is 4-byte aligned at the point of function calls.
- Large data types (larger than 4 bytes) are 4-byte aligned.
- Register R7 is used as a frame pointer.
- Register R9 has special usage." (Apple Inc., 2010a, p.5)

Note that `aapcs` specifies 8-byte alignment requirements for the stack for public function calls and for large data types (see also Table 5.9 for a comparison of C data type alignments of '`arm32`' prefixed ABIs).

For the `ARMv7` architecture Apple further states: "In general, applications built for the `ARMv7` environment are capable of running in the `ARMv6` environment and vice versa. This is because the calling conventions for the `ARMv7` environment are nearly identical to those found in the `ARMv6` environment."(Apple Inc., 2010a, p.14)

5.3.4 PowerPC Processor Architecture Family

In the 1990s the PowerPC architecture offered an attractive alternative to Intels `x86` as a personal computer platform for several operating systems, including Motorola 68000-based (`m68k`) platforms and newer operating systems. "Although announced in 1991, the PowerPC architecture represents the end product of nearly 20 years of evolution starting with work on the 801 system at IBM. From the beginning, advanced hardware and software techniques were intermingled to develop first RISC and then superscalar computer systems. [...] Historically, CISC architectures evolved in response to the limited availability of memory because complex instructions result in smaller programs. As technology improved, memory cost dropped and access times decreased, so the decode and execution of the instructions became the limiting steps in instruction processing. Work at IBM, Berkeley, and Stanford demonstrated that performance improved if the instruction set was simple and instructions required a small number of cycles to execute, preferably one cycle. The reduction in cycle time and number of cycles needed to process an instruction were a good trade-off against the increased path length. Development along these RISC lines continued at IBM and elsewhere. [...] The work at IBM led to the development of the POWER architecture, which implemented parallel instruction (superscalar) processing, introduced some compound instructions to reduce instruction path lengths in critical areas, incorporated floating-point as a first-class data type, and simplified the architecture as a compiler target."(Hoxe et al., 1996, pp 1–2)

"IBM developed many of the concepts in 1975 for a prototype system. An unsuccessful commercial version of this prototype was introduced around 1986. Four years later, IBM introduced the RS/6000 family of processors based on their POWER architecture. In early 1991, a group from Motorola, IBM, and Apple began work on the PowerPC architecture using the IBM POWER architecture as the base."(Dandamudi, 2005a, p.79) As a competitor to the `x86` and the IBM PC compatible, a standard system architecture for PowerPC was specified, named PReP (PowerPC Reference Platform), in order to allow hardware developers to design hardware-compatible machines that can run Windows NT, OS/2, Solaris and AIX. In 1995, IBM and Apple Computers published the successor CHRP (Common Hardware Reference Platform) which included support for Mac OS and NetWare.

In 1994, Apple Computers exchanged the processor architecture of the Macintosh platform from Motorola 68000 processor family to PowerPC. However, in 2006 the Mac OS X platform was shifted to the `x86` architecture and support for PowerPC-based Macintosh hardware was discontinued by release of Mac OS X 10.6 in 2009.

PowerPC is also considered as the successor architecture for the Amiga "classic" platforms which were also based on Motorola 68000. When Commodore released the Amiga 1000 in 1985, the platform quickly attracted a growing community of users and developers due to its multimedia hardware capabilities. As stated by Andy Warhol: "The thing that I like most about doing this kind of art on the Amiga is that it looks like my work."(Guy Wright and Glenn Suokko, 1986) Beyond its multimedia capabilities, some users consider the AmigaOS operating system ahead of its time, compared to the commercially more successful systems of Microsoft and Apple. As stated by John C. Dvorak: "The AmigaOS remains one of the great operating systems of the past 20 years, incorporating a small kernel and tremendous multitasking capabilities the likes of which have only recently been developed in OS/2 and Windows NT. The biggest difference is that the AmigaOS could operate fully and multitask in as little as 250 K of address space. Even today, the OS is only about 1MB in size. And to this day, there is very little a memory-hogging CD-ROM-loading OS can do the Amiga can't. Tight code there's nothing like it. I've had an Amiga for maybe a decade. It's the single most reliable piece of equipment I've ever owned. It's amazing! You can easily understand why so many fanatics are out there wondering why they are alone in their love of the thing. The Amiga continues to inspire a vibrant - albeit cultlike - community, not unlike that which you have with Linux, the Unix clone."(Dvorak, 1996) However, Commodore International declared bankruptcy in 1994. Since then several companies relicensed the trademark Commodore and Amiga; an official successor platform based on PowerPC was out of sight. In 1997, Phase 5 released the first PowerPC accelerator card for Amiga that should *rescue* the system to the PowerPC architecture. "At the time Phase 5 were one of the only companies that appeared serious to invest money into the Amiga market and move it away from the ageing 68k platform."(Knight, 2006) This provided the base for porting AmigaOS to PowerPC and started the development of new derivates of AmigaOS, such as MorphOS (MorphOS Development Team, 2013). In 2009, Hyperion Entertainment released AmigaOS 4.1 which runs on Amiga "classic" systems with PowerPC-accelerators and PowerPC-based computer systems, including a series of new Amiga hardware systems (A-EON Technology Ltd., 2011) under the label "AmigaONE". In 1995, the AROS project was initiated to develop an open-source version of AmigaOS; currently more than 80% of the original AmigaOS 3.1 are covered (AROS Development Team, 2013). It can run natively on `x86` and PowerPC architectures and has also been *back-ported* to run on `m68k`-based Amiga "classic" machines.

The BeOS operating system was originally developed for PowerPC and later ported to `x86`; but it was not successful. Nowadays it continues to exist as "Haiku", an open-source reimplementation of BeOS for `x86` (Haiku, Inc., 2013).

Currently, the PowerPC is used on IBM server systems running the commercial AIX Unix operating system. A majority of open-source Unix-based operating systems also offer PowerPC ports.

PowerPC cores are also commonly used in embedded systems and video game consoles, such as the Sony Playstation 3, Microsoft XBox 360 and Nintendo Wii. "The PowerPC architecture has made its presence felt in the embedded market where AMCC PowerPC and Motorola PowerPC deliver 32-bit system-on-chip (SOC) integrated products. These SOCs encompass the processor along with built-in clocks, memory, busses, controllers, and peripherals. The companies who license PowerPC include AMCC, IBM, and Motorola."(Rodriguez and Fischer, 2006, Section 1.6)

"In 2006, Freescale and IBM collaborated on the creation of the Power ISA Version 2.03, which represented the reunification of the architecture by combining Book E [Embedded environment] content with the more general purpose PowerPC Version 2.02. A significant benefit of the reunification is the establishment of a single, compatible, 64-bit programming model. [..] Because of the substantial differences in the supervisor (privileged) architecture that developed as Book E was optimized for embedded systems, the supervisor architectures for embedded and general purpose implementations are represented as mutually exclusive categories. Future versions of the architecture will seek to converge on a common solution where possible."(IBM, 2010, p.iii)

5.3.4.1 ABIs and Calling Conventions of `PowerPC`

"The PowerPC architecture is a 64-bit architecture with a 32-bit subset."(IBM and Motorola, 1997, p.1-4) In general, the architecture defines three levels, or programming environments, of the PowerPC architecture: PowerPC user instruction set architecture (UISA), PowerPC virtual environment architecture (VEA) and PowerPC operating environment architecture (OEA). "The UISA defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers."(IBM, 2000, p.xxv) "IBM has defined three ABIs for the PowerPC architecture: the AIX ABI for big-endian 32-bit PowerPC processors and the Windows NT and Workplace ABIs for little-endian 32-bit PowerPC processors."(Hoxe et al., 1996, p.157) In addition to the AIX ABI, based on specifications found in Hoxe et al. (1996, Appendix A), we review three other ABIs for the 32-bit subset of the `PowerPC` architecture:

- The System V ABI (`sysv`) supplement for `PowerPC` 32-bit (Zucker and Karhi, 1995).
- The Embedded ABI (`eabi`) for `PowerPC` 32-bit (Steven Sobek and Kevin Burke, 2004), which is based on a `sysv` ABI.

- The Mac OS X ABI (`osx`) for `PowerPC` 32-bit (Apple Inc., 2010b).

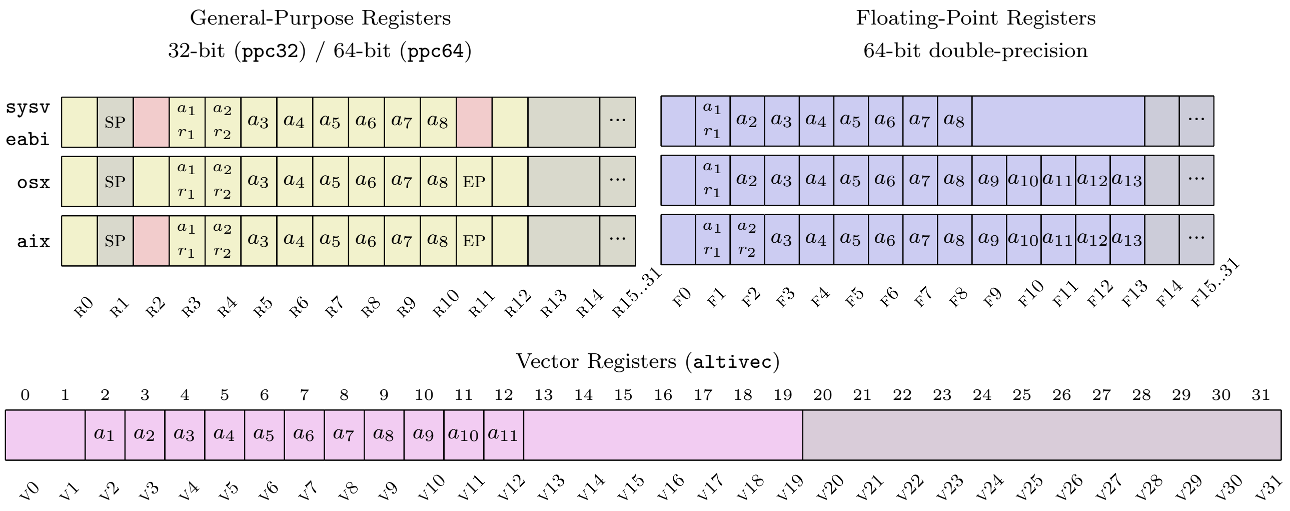

General-Purpose Registers
32-bit (ppc32) / 64-bit (ppc64)
Floating-Point Registers
64-bit double-precision
sysv
eabi
osx
aix
SP
EP
Vector Registers (altivec)

Figure 5.7: Register files of the `ppc` processor architecture family.

Figure 5.7 gives an illustration of register sets as they are utilized in calling conventions of the four ABIs. The PowerPC architecture comprises two large registers sets consisting of 32 registers for general-purpose and floating-point values, respectively. In addition, the architecture supports a SIMD extension named `altivec`. Note the FPU uses a 64-bit double-precision format internally, in contrast to FPUs of other architectures, which typically offer support for 32-bit single-precision. As a consequence for PowerPC, C `float` single-precision floating-point values are promoted to 64-bit double-precision values when they are passed via floating-point registers.

As a common convention to all four ABIs, the eight GPRs R3-R10 are reserved for passing of arguments on all four calling conventions. Return values, up to 64-bit integer, are passed via R3 and R4; single- and double-precision floating-point values are passed via F1; `aix` also supports passing of quad-precision floating-point values via F1 and F2. However, `osx` and `aix` reserve eleven FPRs for passing floating-point values in contrast to eight FPRs on `sysv` and `eabi`.

The calling conventions of `sysv` and `eabi` seem to be straight-forward. First arguments are either passed via GPRs or FPRs. If no register is available for one or the other, the argument is passed in the "parameter list" area of the caller's stack frame in ascending memory addresses i.e. in *right-to-left* push order on downgrowing stacks. The stack frame begins (at the lowest address) with a "back chain" pointer to the previous caller frame on the stack, followed by a reserved cell to save the link register LR (a separate register not part of the GPRs and used by *branch-and-link* instructions), and followed by the parameter list area.

In contrast to `sysv/eabi`, the calling conventions of `osx` and `aix` are more complex. The stack frame contains a homing area which reserves "home location" storage in memory for the eight 32-bit GPRs parameter registers. Arguments passed as 64-bit double-precision values via FPRs are mapped logically to one (for `float` arguments) or two (for `double` arguments) GPRs, which on their part are mapped to the homing area and are marked as used. However, arguments passed via GPRs do not automatically mark a corresponding FPR here and further details need to be considered since only a subset of FPRs may be mapped to GPRs. As the `osx` ABI states "When floating-point registers are exhausted, the caller places floating-point elements in the parameter area."(Apple Inc., 2010b, p.19) In part, the specification of `osx` is inprecise here; a clearer explanation was found in the `aix` specification: "Any floating-point values that extend beyond the first 8 words of the argument list must also be stored at the corresponding location on the stack."(Hoxe et al., 1996, p.163) The layout of stack frames, including the homing area, is further discussed in Section 5.3.7; a comparison of stack frames, including those of `osx` and `sys`, is illustrated in Figure 5.12.

Note the role of register R2 differs among ABIs; it functions as a volatile register by `osx` while it is used as a system-reserved register by the other ABIs. The registers R11 and R12 are reserved for special ABI-specific purposes. `sysv` defines them as volatile registers which may be modified during function linkage (Zucker and Karhi, 1995, p.3-14). On the `osx` and `aix` ABIs registerR11 is volatile for caller and callee of functions. However, some C compilers support an extension for *nested* function definitions where this register becomes an environment pointer (EP) that points to the top-level function's environment i.e. for access to local variables from within the nested functions. `osx` defines that R12 is set to the address of the branch target before indirect calls for dynamic code generation, whereas `aix` defines R12 to be used for special exception handling and global linkage routines.

The utilization of `altivec` registers is incorporated by the `osx` ABI; up to eleven registers are utilized for passing vector data types. A corresponding C data type, named `vector`, is defined as a C extension for the GNU compiler.

5.3.5 MIPS Processor Architecture Family

"The MIPS architecture was born in the early 1980s from the work done by John Hennessy and his students at Stanford University. Over the course of the next 14 years, the MIPS architecture evolved in a number of ways and its implementations were used very successfully in workstation and server systems. [...] Over that time, the architecture and its implementations were enhanced to support 64-bit addressing and operations, support for complex memory-protected operating systems such as UNIX, and very high performance floating point. Also in that period, MIPS Computer Systems was acquired by Silicon Graphics and MIPS processors became the standard for Silicon Graphics computer systems. With 64-bit processors, high-performance floating point, and the Silicon Graphics heritage, MIPS processors became the solution of choice in high-volume gaming consoles. In 1998, MIPS Technologies emerged from Silicon Graphics as a stand-alone company focused entirely on intellectual property for embedded markets. As a result, the pace of architecture development has increased to address the unique needs of these markets: high-performance computation, code compression, geometry processing for graphics, security, signal processing, and multi-threading. Each architecture development has been matched by processor core implementations of the architecture, making MIPS-based processors the standard for high-performance, low-power applications."(Sweetman, 2007, p.V) "These days MIPS is not the highest-volume 32-bit architecture, but it is in a comfortable second place."(Sweetman, 2007, p.XV) "ARM gets more headlines, but MIPS sales volumes remain healthy enough: 100 M MIPS CPUs were shipped in 2004 into embedded applications."(Sweetman, 2007, p.1) "A piece of equipment built around a MIPS CPU might have cost you $35 for a wireless router or hundreds of thousands of dollars for an SGI supercomputer (though with SGIs insolvency, those have now reached the end of the line). Between those extremes are Sony and Nintendo games machines, many Cisco routers, TV set-top boxes, laser printers, and so on."(Sweetman, 2007, p.XV)

5.3.5.1 ABIs and Calling Conventions of `mips`

The original 32-bit MIPS ISA, named `MIPS-I`, was revised in five rounds until it became the 64-bit ISA named `MIPS64` with a 32-bit subset architecture `MIPS32`; both are referred to as `MIPS32/64`. `MIPS-I` was implemented by the *R2000* and *R3000* CPUs which were used among others in Sony's Playstation 1. The 64-bit architecture of MIPS was introduced with the `MIPS-III` and it was first implemented by the *R4000* CPU. Integer-based SIMD instructions were added with the `MIPS-V` ISA and were simultaneously released as a SIMD extension, named `MDMX`. With a few exceptions MIPS ISAs are backward compatible, for example `MIPS I` 32-bit code can run on a *R4000* CPU. However, "the only ISA version that could cause you trouble is MIPS V, some of which is not available in MIPS64. But then it was never implemented, either."(Sweetman, 2007, p.44) A compressed 16-bit ISA, similar to ARM's `Thumb` mode, was also designed for the MIPS architecture, named `MIPS16e`.

"The most important ABIs in MIPS history are:

- `o32`: Grew from traditional MIPS conventions ('o' for old) [...] `o32` is still pretty much universally used by embedded toolchains and for 32-bit Linux.

- `n64`: New formal ABI for 64-bit programs on 64-bit CPUs running under Silicon Graphics Irix operating system. SGIs 64-bit model makes both pointers and C long integer types into 64-bit data items. However, `n64` also changes the conventions for using registers and the rules for passing parameters; because it puts more arguments in registers, it improves performance slightly.

- `n32`: A partner ABI to `n64`, this is really for "32-bit" programs on 64-bit CPUs. It is mostly the same as `n64`, except for having pointers and the C long data type implemented as 32 bits. That can be useful for applications where a 32-bit memory space is already spacious, 64-bit pointers represent nothing but extra overhead."(Sweetman, 2007, p.311-312)

In addition the 32-bit `eabi` ABI for *Homebrew* development on the PlayStation Portable game console is also included here. An inofficial specification was published via mailing list (Christopher, 2003).

MIPS CPUs comprise 32 general-purpose registers as depicted in Figure 5.8. Register R0 is special as it represents a constant that always returns zero. R1 is an assembly temporary (AT). Furthermore, several registers takes specific roles such as the global pointer (GP), stack pointer (SP), frame pointer (FP) and link register (LR); registers 26 and 27 are reserved for the operating-system kernel.

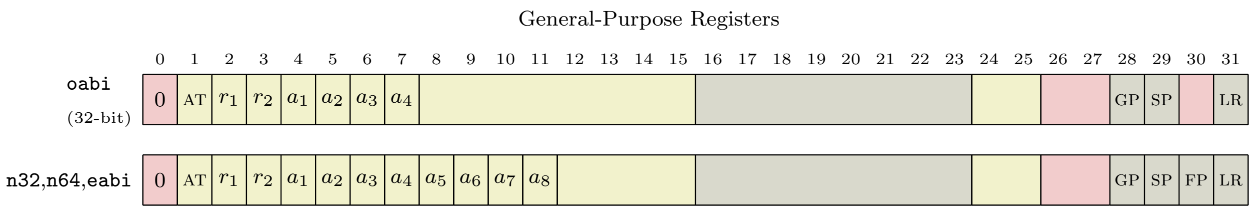

General-Purpose Registers
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
oabi
(32-bit)
0 AT r_1 r_2 a_1 a_2 a_3 a_4 GP SP LR
n32,n64,eabi
0 AT r_1 r_2 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 GP SP FP LR

Figure 5.8: Register files on the `mips` processor architecture family.

The old `o32` ABI is used as the base for the System V ABI supplement for 32-bit MIPS platforms (AT&T, 1991). "Silicon Graphics has defined the following parameter passing convention. The first four "in" parameters are passed to a function in \$a0, \$a1, \$a2, and \$a3 [labeled a_1-a_4 in Figure 5.8]. The convention states that space will be allocated on the stack for the first four parameters even though these input values are not stored on the stack by the caller. All additional "in" parameters are passed on the stack. Register \$v0 [labeled r_1 in Figure 5.8] is used to return a value."(Britton, 2003, p.53) With the support for 64-bit `long long` data types, register R3 (labeled r_2) was also included to pass large integer values. In contrast to the `o32` MIPS, the modern ABIs `n64`, `n32` and `eabi` use eight registers for passing integer arguments and no stack space is reserved.

The first FPU co-processor available in `MIPS-I` was rather complicated and led to odd calling convention rules: "One of the worst faults caused by the age of `o32` is that its use of registers is compatible with the very earliest MIPS floating-point units, which used only the even-numbered registers to hold floating-point values. Double-precision values quietly extended into the adjacent odd-numbered register; the odd-numbered registers were used only when reading or writing FP values from memory, or from integer registers. `o32`'s resulting register conventions do not quite prevent software from using all 32 registers in later CPUs, but they dont make for great efficiency."(Sweetman, 2007, p.322) The FPU registers for `MIPS-I` are illustrated in Figure 5.9: single-precision and double-precision registers are addressed by even-numbered registers; note that the odd-numbered single-precision registers are not considered at all.

The number of floating-point registers for passing arguments of the `o32` ABI is surprisingly small; up to two registers, F12 and F14, are reserved. Due to historic reason the registers are only used if the first arguments in the calling sequence are of floating-point values; otherwise all data is passed via integer registers and then via the stack. "Old-fashioned C had no built-in mechanism for checking that the caller and callee agreed on the type of each argument to a function. To help programmers survive this, the caller converted arguments to fixed types: int for integer values and double for floating point. There was no way of saving a programmer who confused floating-point and integer arguments, but at

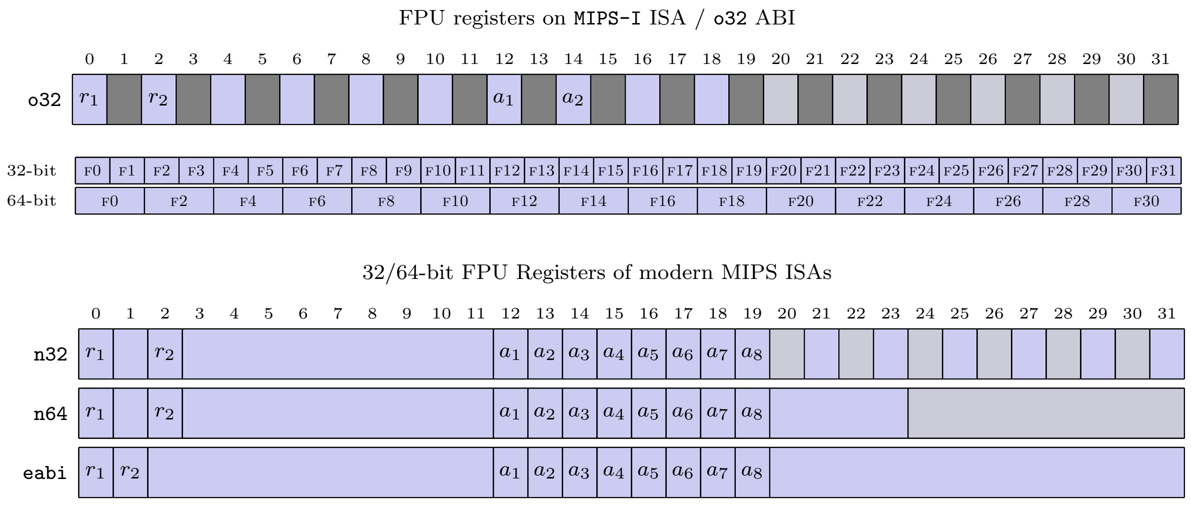

FPU registers on MIPS-I ISA / o32 ABI
o32
r_1
r_2
a_1
a_2
32-bit
64-bit
32/64-bit FPU Registers of modern MIPS ISAs
n32
n64
eabi
a_3
a_4
a_5
a_6
a_7
a_8

Figure 5.9: FPU registers of the mips processor architecture family.

least some possibilities for chaos were averted. Modern C compilers use function prototypes available when the calling function is being compiled, and the prototypes define all the argument types. But even with function prototypes, there are routines – notably the familiar printf() – where the type of argument is unknown at compile time; printf() discovers the number and type of its arguments at run time. MIPS made the following rules. Unless the first argument is a floating-point type, no arguments can be passed in FP registers. This is a kludge that ensures that traditional functions like printf() still work: Its first argument is a pointer, so all arguments are allocated to integer registers, and printf() will be able to find all its argument data regardless of the argument type. The rule is also not going to make common math functions inefficient, because they mostly take only FP arguments. Where the first argument is a floating-point type, it will be passed in an FP register, and in this case so will any other FP types that fit in the first 16 bytes of the argument structure. Two doubles occupy 16 bytes, so only two FP registers are defined for arguments FA0 and FA1, or $f12 and $f14."(Sweetman, 2007, p.321)

"The floating-point register conventions change more dramatically; this is not surprising, since the n32/n64 conventions are for later MIPS CPUs, which have a full 64-bit floating-point unit with 32 fully usable independent registers."(Sweetman, 2007, p.327) Although eabi is a 32-bit MIPS ABI it also takes advantage of a modern 64-bit MIPS FPU; each floating-point register supports two storage formats, single-precision or double-precision, and eight registers are reserved for passing floating-point values.

5.3.6 SPARC Processor Architecture Family

The SPARC (Scalable Processor ARChitecture) processor architecture was introduced by Sun Microsystems in 1987, a company founded in 1982 by three students of Stanford; the acronym "Sun" is derived from the initials of the Stanford University Network. Sun began developing the operating system "SunOS" (later renamed to "Solaris") and built corresponding server/workstation hardware. Early machines were based on the Motorola 68000 processor family. Later in 1985, the company formulated the first SPARC design. Sun was successful with SPARC server and workstation products but more than that it contributed significant technologies such as Java and the ZFS file system.

In January 2010 Oracle Corporation acquired Sun and rebranded its technology assets. However, in mid-2000s Sun decided to open-source major parts of Solaris and the SPARC architecture, known as OpenSolaris and OpenSPARC.

"Small amounts of computer hardware Intellectual Property (IP) have been available for many years in open-source form, typically as circuit descriptions written in an RTL (Register Transfer Level) language such as Verilog or VHDL. However, until now, few large hardware designs have been available in open-source form. One of the most complex designs imaginable is for a complete microprocessor; with the notable exception of the LEON 32-bit SPARC processor, none have been available in open-source form until recently. In March 2006, the complete design of Sun Microsystems "UltraSPARC" T1 microprocessor was released in open-source form, it was named OpenSPARC T1. In early 2008, its successor, OpenSPARC T2, was also released in open-source form. These were the first (and still only) 64-bit microprocessors ever open-sourced."(Weaver, 2008, p.xiii)

5.3.6.1 ABIs and Calling Conventions of `sparc`

"SPARC is based on the RISC I and II designs engineered at the University of California at Berkeley from 1980 through 1982. SPARCs "register window" architecture, pioneered in the UC Berkeley designs, allows for straightforward, high-performance compilers and a reduction in memory load/store instructions."(HAL Computer Systems, 1998, p.19)

Unlike most other processor architectures SPARC uses a very large register file where a small subset of registers is available to a subroutine. "The SPARC architecture provides for a register file with a mapping register that indicates the active registers. Typically, 128 registers are provided, with the programmer having access to the 8 global registers, and only 24 of the mapped registers at any one time. The *save* instruction changes the register mapping so that new registers are provided. A similar instruction, *restore*, restores the register mapping on subroutine return.
The 32 registers are divided into four groups: "in", "local", "out" and "general". The eight general registers G0 - G7 are not mapped and are global to all subroutines. The IN registers are used to pass arguments to closed subroutines, the LOCAL registers are for a subroutine's local variables, and the OUT registers are used to pass arguments to subroutines that are called by the current subroutine. When the "save" instruction is executed the out registers become the in registers, and a new set of local and out registers is provided. The mapping pointer into the register file is changed by 16 registers."(Paul, 2000, p.185)

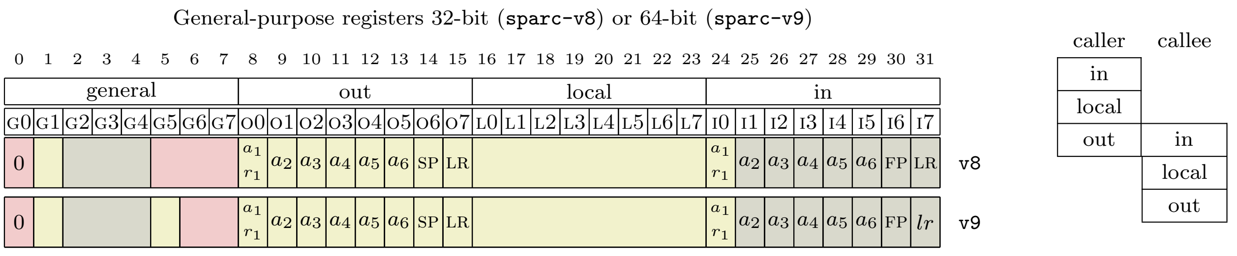
The large register file works as a call stack in hardware; the word 'scalable' in SPARC is due to the variable number of registers that can be packaged on to a core which ranges between 40 and 520 registers i.e. CPU designers can choose between 2 to 32 register windows. However, the execution environment still requires a call stack in main memory for storing additional arguments/local variables beyond the first six arguments and eight local variables in registers. Stack frames also reserve a homing area for I0-I5 registers and a save area for register window content (for the eight LOCAL and OUT registers). The latter is used by a trap handler for register window overflow management during a function call/return where inactive register windows are moved between hardware registers and main memory on the call stack.

The first ISA of the SPARC architecture (for 32-bit) is known as SPARC Version 7 (`sparc-v7`) and was released 1986. In 1990 SPARC Version 8 (`sparc-v8`) was specified in SPARC International (1992) which featured enhanced division/multiply instructions. A corresponding System V ABI supplement for SPARC 32-bit systems is specified in AT&T and SCO (1996). In the mid-1990 SPARC was extended by a 64-bit ISA known as `sparc-v9`; a corresponding System V ABI is specified in SPARC International (1997).

"SPARC-V9 extends the address space of SPARC to 64 bits and adds a number of new instructions and other enhancements to the architecture. SPARC-V9, like its precedessor SPARC-V8, is a microprocessor specification by the SPARC Architecture Commitee of SPARC International. SPARC-V9 is not a specific chip; it is an architecture specification that can be implemented as a microprocessor by anyone securing a license from SPARC International."(Weaver and Germond, 1994, xiii) "The most important SPARC-V9 architectural mandate is binary compatibility of nonprivileged programs across implementations. Binaries executed in nonprivileged mode should behave identically on all SPARC-V9 systems when those systems are running an operating system known to provide a standard execution environment. One example of such a standard environment is the SPARC-V9 Application Binary Interface (ABI)."(HAL Computer Systems, 1998, p.19) The System V ABI for `sparc-v9` is defined in SPARC International (1997).

"A SPARC-V9 processor logically consists of an integer unit (IU) and a floating-point unit (FPU), each with its own registers. This organization allows for implementations with concurrent integer and floating-point instruction execution. Integer registers are 64-bits wide; floating-point registers are 32-, 64-, or 128-bits wide. Instruction operands are single registers, register pairs, register quadruples, or immediate constants."(HAL Computer Systems, 1998, p.31) "An implementation of the SPARC-V9 IU may contain from 64 to 528 general-purpose 64-bit r registers. This corresponds to a grouping of the registers into 8 global r registers, 8 alternate global r registers, plus a circular stack of from 3 to 32 sets of 16 registers each, known as register windows."(HAL Computer Systems, 1998, p.32)

Figure 5.10 gives an overview of SPARC's register utilization. Integer parameters are passed by similar means in 32-bit (`v7`/`v8`) and 64-bit (`v9`) modes: The first six arguments are passed via register O0 to O5; the callee accesses these via register I0 to I5 due to the overlapping register windows as illustrated on the right-hand panel of Figure 5.10; note the overlapping of the caller's "out" and the callee's "in" register group. Similar to MIPS, the G0 register represents the constant zero. G1 is a volatile register



General-purpose registers 32-bit (sparc-v8) or 64-bit (sparc-v9)
general
out
local
in
G0 G1 G2 G3 G4 G5 G6 G7 O0 O1 O2 O3 O4 O5 O6 O7 L0 L1 L2 L3 L4 L5 L6 L7 I0 I1 I2 I3 I4 I5 I6 I7
SP
LR
FP
v8
v9
caller
callee

Figure 5.10: Register files on the sparc processor architecture family.

while Registers G2 to G4 are need to be preserved. Registers G5 to G7 are reserved for the system; note register G5 was redefined as a second volatile register in v9.

However, the handling of floating-point arguments has significantly changed between v8 and v9, as has the FPU hardware. An overview of the FPU register files and its utilization for v8 and v9 ABIs is illustrated in Figure 5.11.

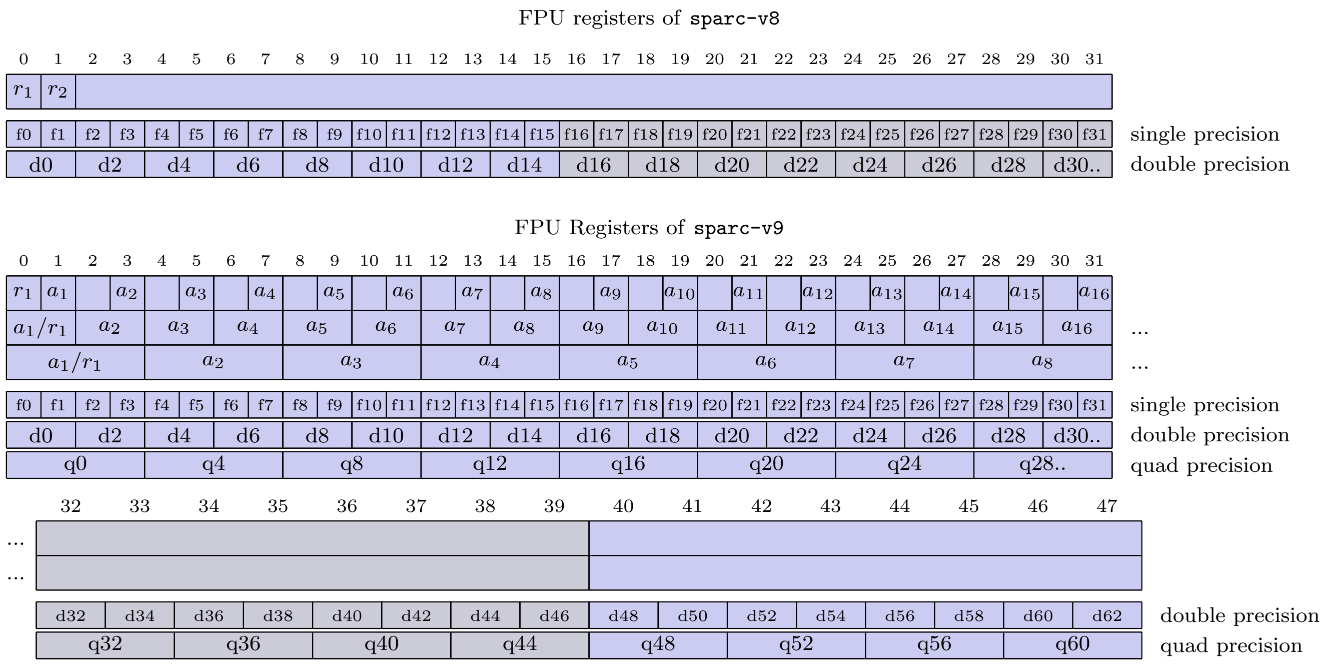


Figure 5.11: FPU registers of the sparc processor architecture family.

The v8 ABI it is defined that “except for floating-point return values, global floating-point registers have no specified role in the standard calling sequence.”(AT&T and SCO, 1996, p.3-13) Floating-point arguments are passed via general-purpose registers and the stack.

Note the v8 FPU comprises 32 single-precision registers that can be paired as 16 double-precision registers. However, the v9 FPU was extended with additional double- and quad-precision registers without extending the set of single-precision registers. “The FPU has thirty-two 32-bit (single-precision) floating-point registers, thirty-two 64-bit (double-precision) floating-point registers, and sixteen 128-bit (quad-precision) floating-point registers, some of which overlap. Double-precision values occupy an even-odd pair of single-precision registers, and quad-precision values occupy a quad-aligned group of four single-precision registers. The 32 single-precision registers, the lower half of the double-precision registers, and the lower half of the quad-precision registers overlay each other. The upper half of the double-precision registers and the upper half of the quad-precision registers overlay each other but do not overlay any of the single-precision registers. Thus, the floating-point registers can hold a maximum of 32 single-precision, 32 double-precision, or 16 quad-precision values." (HAL Computer Systems, 1998, p.32) The value transfer of floating-point data types has also changed in `v9`: "Floating-point arguments are passed in the floating-point registers. Unpromoted single-precision arguments are passed in the first 16 odd-numbered F registers. Double-precision arguments are passed in registers D0 through D30. Quad-precision arguments are passed in registers Q0 through Q28. Floating-point return values appear in the floating-point registers. Single-precision values occupy F0; double-precision values occupy D0; quad-precision values occupy Q0" (SPARC International, 1997).

5.3.7 Stack Frames

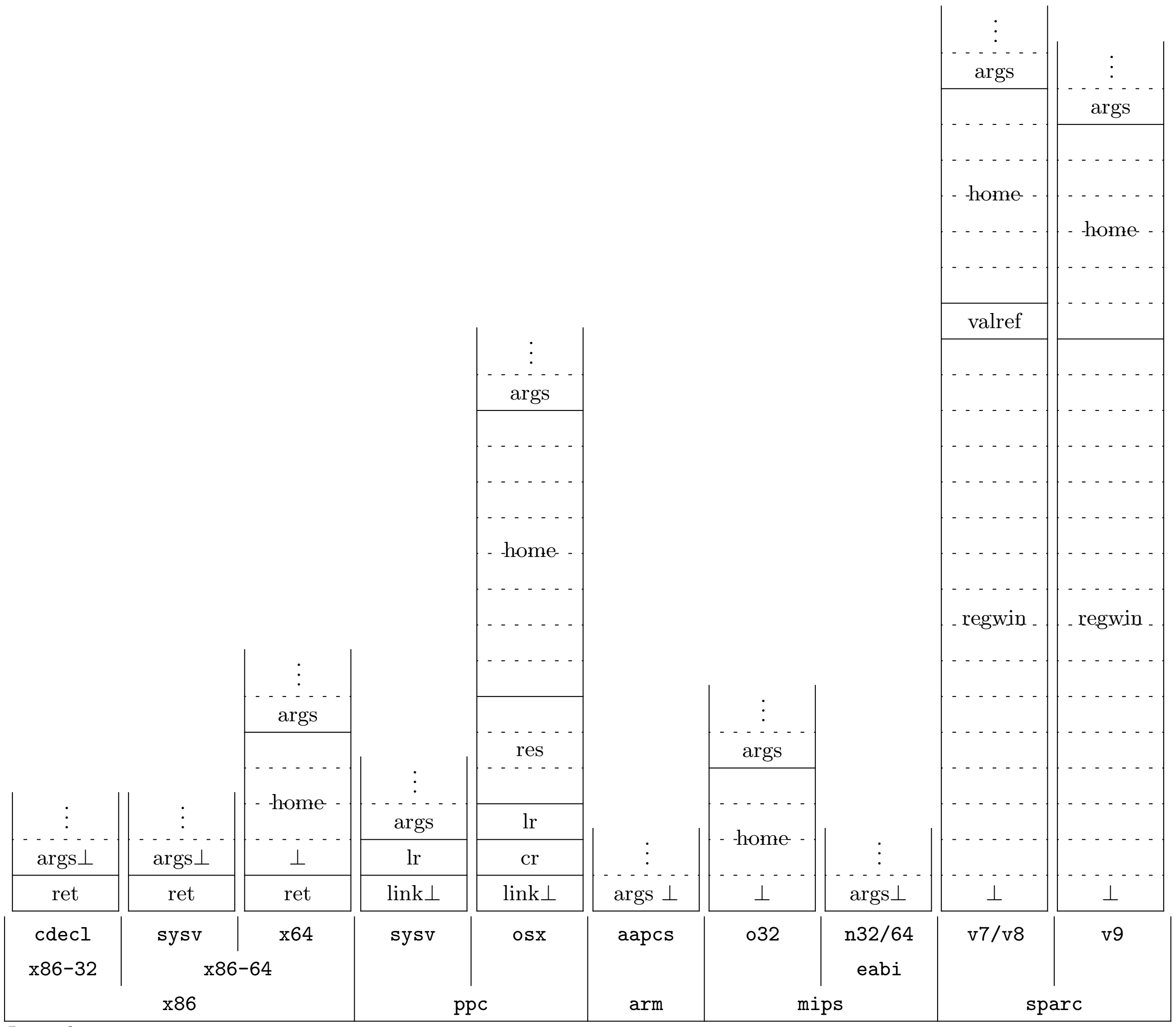
In the previous sections we focused on available registers across processor architectures and their utilization within the calling conventions. Now we give a brief overview on caller stack frames and compare their structure.

Figure 5.12 illustrates the layout structure of the caller stack frame for different processor architectures and calling conventions as they are stored at the top of stack directly after a machine-level function call is executed. The width of each cell in the figure is equal to the bits of the architecture.

A significant difference between the five architecture families is how the return address is passed. RISC architectures use a link register which is loaded with the return address. On the `x86` architecture the return address is pushed as the last element on the stack, denoted "*ret*" in Figure 5.12.

	Alignment	Homing area	
Calling Convention		Offset	Size
`x86-32`	4	-	-
`x86-32-osx`	16	-	-
`x86-64-sysv`	16	-	-
`x86-64-x64`	16	8	32
`ppc32-sysv`	16	-	-
`ppc32-osx`	16	24	32
`arm32-aapcs`	8	-	-
`mips32-o32`	8	0	16
`mips32-eabi`	8	-	-
`mips64-n64`	16	-	-
`mips64-n32`	16	-	-
`sparc-v8`	16	68	24
`sparc-v9`	16	128	48

Table 5.7: Comparison of stack frame characteristics.



Legend:

args	arguments on stack.	lr	save area for link register.
ret	return address.	res	reserved space for kernel.
home	homing area for parameter regs.	regwin	save area for register window.
link	back-chain to previous call frame.	valref	address for large return values.
cr	save area for control register.	⊥	point of alignment.

Figure 5.12: Comparison of the memory layout of stacks.

The topmost element of each stack frame needs to be aligned to a specific memory boundary. Note the point of alignment for `x86` stack frames is above the return address. Table 5.7 gives an overview of different alignment requirements; 8-byte or 16-byte memory boundaries are typically here but `x86-32` simply uses 4-byte alignment and thus has effectively no alignment requirements. However, the Mac OS X ABI for IA-32 (Apple Inc., 2010b, p.43), which is based on `x86-32`, prescribes a 16-byte memory boundary. Alignments larger than 16-byte are also possible; as a special case for the `x86-64-sysv` ABI, the stack frame needs to be aligned at a 32-byte memory boundary if the `__m256` data type (256-bit `x86-64`-specific SIMD C data type extension) is used as an argument type (Matz et al., 2012, p.16).

The *homing area* (denoted "*home*" in Figure 5.12) is part of the stack frame of some ABIs, such as `x86-64-x64`, `ppc32-osx`, `mips-o32` and `sparc-v8/v9`, and serves as a place holder for contents of parameter registers. This can be used to give arguments a memory location and reference if needed. For example, when the address of a function's parameter is requested by C code values in the registers can be saved to their home location on the stack. See Table 5.7 for an overview of sizes and offsets (in bytes) to the top of stack. It is worth noting that the homing area is located right before the argument region starts so that when all parameters in registers are saved to their "home location" the complete argument list is available serialized in stack memory, which can be suitable for the purpose of iterating over the arguments (possibly in a sub-routine). For example, the standard C variadic argument macros `va_list`, `va_start`, `va_arg` and `va_end`, defined in the C standard header file `stdarg.h`, provide an argument iterator for C ellipsis function call arguments. However, for architectures with dependencies between integer and floating-point parameter registers it is often infeasible to implement a *generic* assembly routines which save *all* parameter registers to the homing area simultaneously without knowledge of the exact sequence of types for the first arguments. While homing areas have a fixed size that matches the content of only the integer-based parameter registers, floating-point parameter registers are usually mapped to the integer parameters registers. In order to support integer and, in particular, `float` and `double` arguments passed via floating-point registers, a specific sequence of instructions is needed that matches the types of arguments.

Some ABIs use a homing area while others omit them, even on the same architecture. There exist alternative techniques that can be used for dynamic allocation of homing areas (e.g. by using local variable storage below the callee stack frame). A generic callback handler framework is discuss in Section 5.5.

A notable detail of the stack frame of `sparc-v9` is that stack and frame pointer registers are biased by 2047 bytes towards lower addresses. "This changes the reach of stack relative addressing that uses a 13-bit signed immediate operand of load and store instructions. With the bias, the forward reach is sacrificed but the code can address more of the older stack content."(Jermár, 2007, p.27) Another percularity of stack frames on `sparc-v7/v8` ABI is the *valref* field which explicitly reserves storage for passing memory references for large return values to the callee; the convention in other ABIs is to inject a first hidden parameter in the calling sequence.

In summary the layout of stack frames vary among architectures. While some calling conventions are simple others can be quite complex. Even on a single architecture, such as PowerPC, the stack frames can differ significantly, depending on the software convention if, for example, a homing area is defined or whether further management information is part of a stack frame (for example, the system-reserved fields on `ppc32-osx` denoted "*res*" in Figure 5.12). It is also noticeable that SPARC uses a very large stack frame structure due to the register window (the large region at the top denoted "*regwin*" in Figure 5.12).

5.3.8 Data Representation

The mapping scheme between C and machine data types has a significant influence on the calling convention for the selection of registers and the allocation of the stack location, as well as for the memory layout of composite data types.

For a long time the machine-level data type of the C `int` data type was decided by the recommendation that " `int` will normally be the natural size of a particular machine"(Kernighan and Ritchie, 1988, p.36) and it was widely adopted on 16- and 32-bit architectures. However, with the rise of 64-bit computing, this rule was discontinued for practical reasons. The changeover from 32-bit to 64-bit offers a much larger address space for handling large data but it also doubles the size of C pointers and thus leads to a significant increase of memory consumption. Although the average amount of available main memory also increased, the memory bus still remains the bottleneck of performance. CPUs use fast multi-level cache memory to speed up the processing of data with good locality. "A much larger practical effect in some commercially important applications comes from the consumption of additional memory and the costs of transporting that memory throughout the system. 64-bit integers require twice as much space as 32-bit integers. Further, the latency penalty can be enormous, especially to disk, where it can exceed 1,000,000 CPU cycles (3 nsec to 3 msec). `int` is by far the most frequent data type to be found (statically and sometimes dynamically) within C and C++ programs."(The Open Group, 1997) `int` is also frequently used as the base type of arrays and within the fields of data records in countless programs so that data objects that once were fitting in a single cache line would be spreaded across several cache lines due to 64-bit `int` types.

C Data Model	`short`	`int`	`long`	`long long`	`void*`	Platforms
`ILP32`	16	32	32	64	32	32-bit Execution Environment on IA64.
`LLP64`	16	32	32	64	64	Microsoft Windows 64-bit.
`LP64`	16	32	64	64	64	System V AMD64 (Linux, Mac OS X, BSD)
`ILP64`	16	64	64	64	64	HAL Computer Systems port of Solaris to SPARC 64
`SILP64`	64	64	64	64	64	Unicos

Table 5.8: Overview of 64-bit C programming models.

As a compromise several 64-bit C programming models were designed as depicted in Table 5.8. Each data model uses a different 32/64-bit mapping of fundamental C integer-based data types. The data models are abbreviated using a type signature scheme that gives a subset of data types and followed by a particular bit size mapping. The most common programming models are `LP64` on Unix-based systems and `LLP64` on Microsoft Windows system; both differ by the size of the `long` data type while the `int` data type remains a 32-bit machine word.

However the data model does not reveal the *alignment* in memory. The *alignment* of a data type specifies restrictions for the memory address location imposed by memory caching and bus hardware. Most processor architectures permit explicit memory access on *unaligned* addresses. However, some other architectures trigger a memory bus exception such as SPARC architectures.

ABI specifications provide information about the size, alignment and encoding of machine data types. However, alignment and sizes can also be determined emipirically. Listing 18 gives the C code of a program that determines the size and alignment properties for basic C data types. We make use of the built-in C function `sizeof` to determine the size of a data type. For the computation of the alignment of a data type we use the following schema: For each data type an inline C structure data type is defined that begins with a 1-byte field and followed by a second field with the corresponding data type whose alignment needs to be detected. The compiler is forced to align the offset of the second field to the alignment requirements of its data type. Thus, the alignment is equal to the offset of the second field which can be retrieved via the C macro `offsetof`.

```c
#include <stdio.h>
#include <stddef.h>
#define alignof(X) offsetof( struct { char c; X member; }, member)
int main(int argc, char* argv[]) {
#define X(Y) printf( #Y "\t%ld\t%ld\n", sizeof(Y), alignof(Y));
  X(_Bool)
  X(char)
  X(short)
  X(int)
  X(long)
  X(long long)
  X(double)
  X(float)
  X(void*)
  X(long double)
return 0;
}
```

Listing 18: Program to determine the size and alignment of C data types.

We give an overview of sizes and alignments of C data type implementations for several platforms in Table 5.9. Both properties are given with the format "⟨*size*⟩:⟨*alignment*⟩"; a single number implies natural alignment i.e. the alignment is equal to the size.

Across the spectrum of 32- and 64-bit target platforms of our survey a subset of C data types is consistently mapped to machine data types with a common size and natural alignment, such as 8 bit `char`, 16 bit `short` and 32-bit `int` and `float`. `long long` and `double` are mapped as 64 bit data types.

ABI	`_Bool`	`char`	`short`	`int`	`long`	`long long`	`float`	`double`	pointer	`long double`
`x86-32-sysv`	1	1	2	4	4	8:4	4	8:4	4	12:4
`x86-32-osx`	1	1	2	4	4	8:4	4	8:4	4	16
`x86-32-win32`	1	1	2	4	4	8	4	8	4	8
`x86-32-mingw32`	1	1	2	4	4	8	4	8	4	12:4
`x86-64-sysv`	1	1	2	4	8	8	4	8	8	16
`x86-64-x64`	1	1	2	4	4	8	4	8	8	8
`arm32-aapcs`	1	1	2	4	4	8	4	8	4	8
`arm32-apcs`	1	1	2	4	4	8:4	4	8:4	4	8:4
`arm32-ios`	1	1	2	4	4	8:4	4	8:4	4	8:4
`arm64-aapcs`	1	1	2	4	4/8	8	4	8	8	16
`sparc32-v8`	1	1	2	4	4	8	4	8	4	16:8
`sparc64-v9`	1	1	2	4	8	8	4	8	8	16
`ppc32-sysv`	1	1	2	4	4	8	4	8	4	8
`ppc32-osx`	4	1	2	4	4	8:4	4	8:4	4	16
`ppc32-eabi`	1	1	2	4	4	8	4	8	4	16:8
`ppc64-sysv`	1	1	2	4	8	8	4	8	8	16
`mips32-o32`	1	1	2	4	4	8	4	8	4	8
`mips32-eabi`	1	1	2	4	4	8	4	8	4	8
`mips64-n64`	1	1	2	4	8	8	4	8	8	16
`mips64-n32`	1	1	2	4	4	8	4	8	4	16
`ia64-linux`	1	1	2	4	8	8	4	8	8	16
`hppa-linux`	1	1	2	4	4	8	4	8	4	8

Table 5.9: Overview of data representations of C Data Types. The quantities give the size in bytes. Alignment requirements other than natural alignments (i.e $\neq$ size) are given separated by ':'.

However, on several platforms the alignment is less restrictive i.e. 4-byte instead of 8-byte alignment. Due to different 64-bit programming models, the `long` type significantly differs across ABIs of 64-bit architectures, such as on the `x86-64` architecture; `long` occupies 4 bytes on 64-bit Windows and 8 bytes on 64-bit System V platforms. The `_Bool` data type is usually implemented as a single byte except on Mac OS X for PowerPC 32-bit architectures where it occupies 4 bytes of memory. The size of C pointers (i.e. memory addresses) is directly related to the bits of an architecture.

We also include the size and alignment properties of the C `long double` data type; the different sizes among platforms indicates that different floating-point formats are used. `x86-32` platforms utilize the `x87` FPU which uses an 80-bit extended-precision format internally. Since the stack is always 4-byte aligned on `x86` 12 bytes instead of 10 bytes are used for storage. Mac OS X on `x86` also uses extended-precision as storage format but adds a padding of 6 bytes to ensure 16-byte alignment. Microsoft Windows platforms, whether on 32- or 64-bit architectures, use `double` for the implementation of `long double`. SPARC and PowerPC architectures have support for 128-bit quad-precision floating-point arithmetic and thus use 16 bytes of storage. MIPS 32-bit uses 64-bit double-precision formats for the implementation of `long double`, while on MIPS 64-bit architectures 128-bit quad-precision floating-point data types are used.

5.3.8.1 Computation of offsets

The C standard does not specify the size and alignment of basic C data types nor how the memory layout of member fields is computed. However, analogous to the implementation of function calls at machine-level, ABI specification give details for the layout of composite data types. Figure 5.13 illustrates the variety of memory layouts of a `struct` data type on three platforms.

```
struct {
  short     a;
  long      b;
  int       c;
  long long d;
  char      e;
};
```

	x86-32			x86-64-x64			x86-64-sysv		
Fields	size	align	offset	size	align	offset	size	align	offset
a	2	2	0	2	2	0	2	2	0
b	4	4	4	4	4	4	8	8	8
c	4	4	8	4	4	8	4	4	16
d	8	4	12	8	8	16	8	8	24
e	1	1	20	1	1	24	1	1	32
end			21			25			33
total	24	4		32	8		40	8	

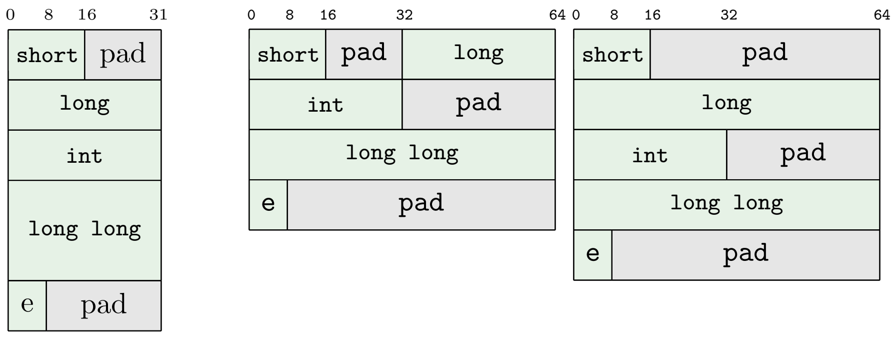


Figure 5.13: Layout of C struct data types on different architectures.

In general, the offset of a member within a C `struct` data structure depends on the layout of previous members in the sequence. The compiler has to account for a location next to the nearest free byte offset that matches the alignment of its member type. Furthermore, the maximum alignment requirement of its members is used as the total alignment requirement for the data structure object.

The following algorithm is used in the `rdyncall` R package for the computation of `struct` and `union` fields from type signatures as discussed in Section 4.5:

Algorithm Given a definition of a C `struct` data type T with an ordered sequence of n field types $FT = \{ft_1, ft_2, .., ft_n\}$ and a platform-specific type information database with two query functions, $Q_s(t)$ and $Q_a(t)$, to obtain the *size* and *alignment* of the type t, we will compute corresponding field offsets $o_1, o_2, .., o_n$ (in bytes) and the size $Q_s(T)$ and alignment $Q_a(T)$ of the `struct` data type T.

For the set of fundamental C types B (`char` , `int` , `float` etc..) the size and alignment properties are predefined i.e. the query functions $Q_s(t)$ and $Q_a(t)$ are predefined $\forall\ t \in B$.

For example, on the `x86-32-sysv` platform, the following query functions are predefined for `int` and `double` :

$$\begin{aligned} Q_s(\texttt{int}) &= 4 \\ Q_a(\texttt{int}) &= 4 \\ Q_s(\texttt{double}) &= 8 \\ Q_a(\texttt{double}) &= 4 \end{aligned}$$

Let $align(m, a)$ be the aligned address of memory address m to the alignment requirement a:

$$align(m, a) = \left\lfloor \frac{m + a - 1}{a} \right\rfloor a$$

Let $O = o_1, o_2, .., o_n$ be the output sequence of field offsets, and o_1 be initialized to zero:

$$o_1 = 0$$

The field offsets $o_2..o_n$ are then computed in sequence as follows:

$$o_i = align(o_{i-1} + Q_s(t_{i-1}), Q_a(t_i)) \ , \ i = 2..n$$

Then the alignment and size of the `struct` type T can be computed as follows:

$$\begin{aligned} Q_a(T) &= max(Q_a(t_1), Q_a(t_2), ..., Q_a(t_n)) \\ Q_s(T) &= align(o_n + Q_s(t_n), Q_a(T)) \end{aligned}$$

In order to compute the size and alignment of `struct` T, the alignment and size of all field types t_i need to be known or have to be computed in advance. When the computation of `struct` T is completed so that the size $Q_s(T)$ and alignment $Q_a(T)$ is known, other data `struct` or `union` data types, which contain `struct` T as field type, can be computed given that the size and alignment requirements of all other field types are also known.

Analoguous to the above, the size and alignment of C `union` types can be computed; however, all field offsets need to be set to zero.

Note that compilers also offer flags and `#pragma` instructions to enable *structure packing* which removes any paddings but these is not used very frequently. If multiple packing schemes need to be supported, we suggest to use several profiles of base type size/alignment configurations which can be changed at run time.

An implementation of this algorithm, written in R, is available in the `rdyncall` source package on CRAN, in file `R/dynstruct.R`.

5.4 Dynamic Function Calls

In this section we discuss the open-source `dyncall` software library, which offers a portable C API for making calls to precompiled native functions of diverse function type with support for several calling conventions. The library was ported across several platforms and includes support for a variety of ABIs and calling conventions, which we discussed in Section 5.3. `dyncall` serves as a portability layer for the implementation of dynamic call facilities, such as dynamic FFIs, in scripting languages.

We give an introduction to the challenging problem of implementing a dynamic FFI portably in C using the example of R's Static FFI function `.C`. We then discuss our approach, the design of the library and API, followed by examples of its usage. Included here is a brief description of the libraries use as an implementation of `.dyncall` for the Dynamic R FFI (see Section 4.3). We then describe the implementation architecture and discuss ports to different processor architectures and ABIs, including details of implementation in C and assembly languages.

5.4.1 Introduction

FFIs require a flexible "caller code" that supports diverse function types. In addition, it is a desireable goal to base the implementation on a portable language, such as C.

Since C is a static typed language, the target function type of a C call expression is of constant type and is unchangeable after compilation. As a result, a specific caller code is needed to support a particular function type. However, for a generic solution in C, code is needed for all different kinds of variations of function types; the code size would *explode*. While the C standard offers a couple of "portable" interfaces for advanced language features, such as variadic function types, there does not exist a C standard interface for dynamic *composition* of calls at run-time.

Static FFIs can be implemented in C by limiting the range of supported C function types. For example, `.C` offers support for calling C functions with return type `void`, and up to 65 parameter types; the latter must all be of type "data pointer" such as `int*` (for `logical` and `integer`), and `double*` for (`numeric`). Such a solution is a compromise.

We now take more closer look at the implementation of `.C` by using an excerpt of the source code given in Listing 19. In a first step all R atomic vector arguments to be passed to C are saved in a C array of data pointers, referencing the first element – recall that R value objects are vectors, represented as C arrays. In a second step caller code is selected (in a switch) based on the number of arguments. Each caller code handles a function type with a different number of arguments, all of which have a parameter type of `void*`. The actual arguments are picked from the array of step 1. Note that the R argument types and C parameter types are not considered in this process for selection of a caller code. This makes the number of cases to be handled relatively small.

In general, arguments of type "data pointer", such as `void*`, `int*` or `float*`, are passed consistently at machine-code level, regardless of the base type. Thus, a specific caller code for a particular number

```c
/* .C() {op=0}  or  .Fortran() {op=1} */
SEXP attribute_hidden do_dotCode(SEXP call, SEXP op, SEXP args, SEXP env)
{
    void **cargs;
    /* ... */
    cargs = (void**) R_alloc(nargs, sizeof(void*));
    nargs = 0;
    /* ... */
    for(pargs = args ; pargs != R_NilValue; pargs = CDR(pargs)) {
        /* ... */
        cargs[nargs] = RObjToCPtr(CAR(pargs), naok, dup, nargs + 1,
                                  Fort, symName, argConverters + nargs,
                                  checkTypes ? checkTypes[nargs] : 0,
                                  encname);
        nargs++;
    }
    /* ... */
    switch (nargs) {
    case 2:
        fun(cargs[0], cargs[1]);
        break;
    case 3:
        fun(cargs[0], cargs[1], cargs[2]);
        break;
    /* ... */
    case 10:
        fun(cargs[0],  cargs[1],  cargs[2],  cargs[3],  cargs[4],
            cargs[5],  cargs[6],  cargs[7],  cargs[8],  cargs[9]);
        break;
    /* ... */
    case 30:
        fun(cargs[0],  cargs[1],  cargs[2],  cargs[3],  cargs[4],
            cargs[5],  cargs[6],  cargs[7],  cargs[8],  cargs[9],
            cargs[10], cargs[11], cargs[12], cargs[13], cargs[14],
            cargs[15], cargs[16], cargs[17], cargs[18], cargs[19],
            cargs[20], cargs[21], cargs[22], cargs[23], cargs[24],
            cargs[25], cargs[26], cargs[27], cargs[28], cargs[29]);
        break;
    /* ... */
    case 65:
    /* ... */
    default:
        errorcall(call, _("too many arguments, sorry"));
    }
    /* ... */
}
```

Listing 19: C Implementation of R's Static FFI `.C`; we give an excerpt from file `src/main/dotcode.c` at line 1600 of R 2.14 and omit several blocks of code in between.

of arguments can make calls to *all* functions where the parameter types are all data pointers, regardless of the base type of each data pointer; the value of a pointer is just a memory address. However, each argument needs to be coerced to a specific R object type in advanced in order to achieve type-safety on the callee side in C.

In summary, `.C` passes all R objects as a C data pointers, referencing the first data element. The argument vector of S-Expressions is transformed to a C array in a generic way by using a fixed byte offset from a header of an S-Expression to its first element; no detailed type checking on a per argument basis is needed in order to build up a generic sequence of C arguments. In a second step a generic call code for data pointers is select from of 65 cases. This leads to a relatively small and portable implementation in C.

This simplified implementation of a FFI causes difficulties for extension developers; type checking and value conversion need to be implemented by R wrapper code, and function calls need to be implemented by C wrapper code.

Dynamic FFIs overcome the limitations of static FFIs since they significantly expand the number of supported C function types, and thereby, they can also take care of type checking and value conversion. However, it is not feasible to write a generic implementation in C. While data pointers are all handled in the same way for a particular calling convention, this is not the case for other basic data types of C. Thus a *dynamic call* interface needs to take account of ABI-specific machine-level details.

5.4.2 Objective

We discuss a portable software library with a C interface that provides an abstraction layer to the calling sequence at machine level. Our objective is to offer this layer as a *dynamic* service for native execution environments that carries out conforming calls to precompiled functions of arbitrary function type, where the latter is specified as part of the interface by the user.

In this context, the term *dynamic* applies in two senses: First, the service should offer a generic solution that functions *dynamically* at run time with support for a large spectrum of function types. Second, the service should operate through a C API comprising functions to compose function calls in a step-wise, successive, *dynamic* manner, ideal for dynamic execution environments such as interpreters. Since several calling conventions may exist on a single native platform, we consider a software architecture that encapsulates a driver of a specific calling convention.

5.4.3 Approach

The compiler implements a function call as a sequence of machine code instructions where the selection of instructions and the exact sequence is not fixed. Some platforms host a variety of C compilers where each code generator can produce different sequences of machine code. Even a single compiler can be operated in a specific mode for optimization that leads to different series of instructions depending on whether the aim is to debug the code, increase execution performance or reduce the code size. But as long as both sides of the call comply with a common calling convention, the caller and callee are compatible and the function call should work.

Effectively, at the point of the machine-level `call` instruction, the state of the machine-level resources related to the passing of arguments is precisely defined. For the callee side of a function call, it is important that argument values are to be found in specific locations on the call stack and/or in registers. More precisely, a conforming function call is given to the callee if there is a stack frame at the top of stack with an exact layout of data fields, and a number of registers of the set of registers for passing arguments are loaded with arguments values, possibly promoted and converted to a platform-specific encoding format. The precise layout of the frame, rules for promotion and alignment of argument values and assignment of values to data fields on the stack and to registers is determined by an algorithm (defined by a calling convention) and the C function type (for a specific case of an implementation of a function or a call).

Thus, since the calling convention does not prescribe a *specific* sequence of instrutions right before the final call instruction, and there is a deterministic algorithm that specifies the exact locations for arguments based on the sequence of argument types and values, we can use a programming interface that allows users to *compose* a function call by given the sequence of arguments in a step-wise manner where the values are buffered. We then work out the call by setting up a stack frame and loading of registers that correspond to the current function type and calling convention on the platform.

While we cannot address precise manipulations of the call stack and register content in C, we can do this in assembly language. Since the stack frame for a function call is a linear block of memory and a fixed subset of registers are used for passing arguments, we use a small call routine written in assembly language, named the *Call Kernel*, to carry out function calls in a generic manner. It implements copying transactions to registers and the stack, followed by the `call` machine instruction.

The Call Kernel is passed an image of the content for the call stack and registers to be copied to the top of the call stack and to be loaded into register files, respectively. The preparation of image data for the call stack frame and register contents is based on the sequence of argument values and paramater types. This is implemented in C and includes details for assignment of values to stack and/or register buffers, value promotion and encoding, and alignment of data fields on the stack.

5.4.4 Architecture

We define an abstract object, named the *Call VM (Call Virtual Machine)*, which encapsulates the details for carrying out calls that conform to a specific calling convention. The interface consists of a portable C API for the specification of calls with support for diverse function types. An implementation of this abstract object provides support for one or more specific calling conventions, implemented for a particular processor architecture. Internally, a Call VM can switch between different *drivers* to support multiple calling conventions. We give a detailed description of the internals in Section 5.4.7.

We now look closer at the interface. In general, a "call" to a function can be characterized by the following attributes:

- Function type specific information:
  - Calling convention
  - Return type
  - Sequence of parameter types
- Call specific information:

- Sequence of argument values
- Address of target function

An implementation of a Call VM represents a state machine, where the parameters of the above are specified via a sequence of interface function calls. The total sequence for the specification of a single call is decomposed into three phases:

1. Preparation:

   (a) Mode: Specification of the calling convention. This step is optional. A Call VM is initially configured to perform function calls that conform to the default C calling convention of the target platform. Users may change this configuration and choose a language-specific (e.g. C++) or platform-specific (e.g. system calls or System DLLs on Microsoft Windows) calling convention for subsequent function calls. As a consequence, the current *driver* of a Call VM object is switched to a different implementation.

   (b) Reset: A Call VM is a reusable object for making numerous consecutive calls to different functions. Before performing the next function call the Call VM state needs to be reset, so that the internal buffers for argument values are cleared and a specification of a new sequence of arguments (Step 2) can be started.

2. Load arguments: A group of interface functions is declared to specify the argument value and corresponding (argument and parameter) type. The sequence of arguments is specified by a corresponding sequence of calls to these interface functions, in accordance with the declaration order of parameter types of the target function from "left to right".

3. Call the target function: A group of "call" functions is defined in the interface of the Call VM. For each supported return type a corresponding interface function is available. The return type and target address are specified as a last step. As a result, the foreign function call is carried out by the service and the return value is passed back via the interface function.

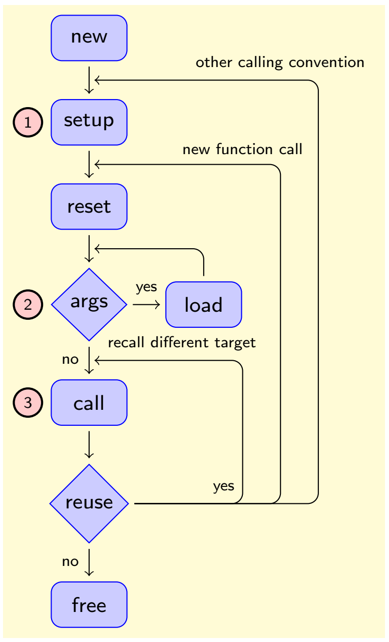
Figure 5.14 gives the automata state machine diagram and the corresponding C interface calls for carrying out foreign function calls via `dyncall`. At the bottom of the figure we give an illustration to derive the sequence of `dyncall` API calls from the C declaration of the foreign function type.

5.4.5 Interface

The API of `dyncall` is accessible via inclusion of a single C header file `dyncall.h`:

```
#include "dyncall.h"
```

We use a consistent naming scheme for elements of the C interface in order to prevent naming conflicts with other C APIs; the following table gives an overview:



```
new:
  DCCallVM *vm;
  vm = dcNewCallVM(4096);   // use 4096 bytes for buffer
setup:
  dcMode(vm, DC_CALL_C_DEFAULT);
reset:
  dcReset(vm);

// Example C Call:
//   pSurf = SDL_SetVideoMode(640,480,32,SDL_OPENGL)
load:
  dcArgInt(vm, 640);         // 1. arg 'width'
  dcArgInt(vm, 480);         // 2. arg 'height'
  dcArgInt(vm, 32);          // 3. arg 'bpp'
  dcArgInt(vm, SDL_OPENGL); // 4. arg 'flags'
call:
  pSurf = dcCallPointer(vm, &SDL_SetVideoMode);

if (again) {
  if (other_callconv)  goto setup;
  if (new_args)        goto reset;
  if (recall)          goto call;
}
free:
  dcFree(vm);
```

Matching the calling sequence using C prototypes:

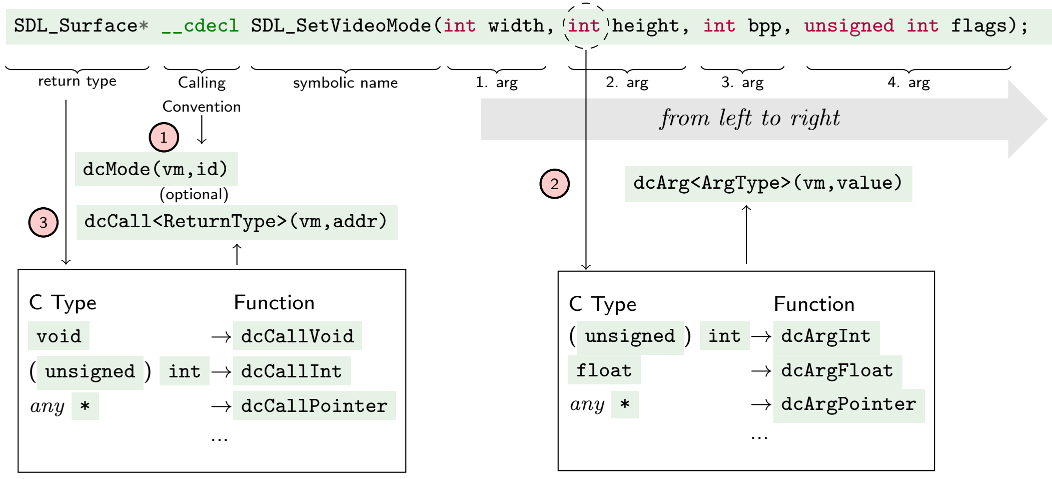


Figure 5.14: State Machine diagram for calling functions via the Call VM (*left*), an example of using the C API (*right*) and matching the calling sequence from the C function prototype definition(*below*).

API elements	Verb chaining	Prefix	Example
Functions	camel-case	`dc`	`dcArgPointer`
Types	camel-case	`DC`	`DCCallVM`
Symbolic constants	upper-case and underscore	`DC_`	`DC_CALL_C_DEFAULT`

All C API function names are prefixed `dc`, derived from 'd'yn-'c'all. Names of types and symbolic constants use an upper-case variant. The chain of verbs in function and type names use a camel-case encoding[2]. Symbolic constants are defined with upper-case letters using underscores to separate verbs.

We declare `typedef` alias types for all supported C types (e.g. `DCint` is an alias to `int`); space separators in C standard type are ignored in this mapping (e.g. `DClonglong` is an alias to `long long`).

Call VMs are self-contained service objects, so that no global variables are used and multiple Call VM objects can co-exist e.g. one for each thread of execution in a multi-threaded execution context.

5.4.5.1 Constructor

The constructor function `dcNewCallVM` creates a new Call VM object and returns a pointer of type `DCCallVM*` as a reference; for the rest of the Call VM API functions this pointer is always passed as the first argument.

```
DCCallVM * dcNewCallVM (DCsize size);
```

We use an internal memory buffer for the preparation of foreign function calls. The `size` parameter gives the size in bytes of internal memory that should be allocated internally; it effectively limits the maximum number of argument values that can be buffered (a size of 4096 bytes is typically used which is large enough to prebuffer arguments for almost all cases i.e. 512-1024 arguments).

5.4.5.2 Phase 1: Preparation

The initial state of a newly created Call VM object is an empty sequence of arguments. After loading arguments and making a call, the buffered arguments are not automatically removed; any subsequent use of the Call VM for making calls to other functions using a different sequence of arguments requires resetting the state machine to the initial state. This is performed by the function `dcReset`:

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

Call VMs are configured for a specific calling convention which is initially set to the default C calling convention of the platform. However, users can change the calling convention as part of the preparation phase via the API function `dcMode`:

[2]A single upper-case character indicates a new verb in the symbol name.

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

The argument `mode` gives the identifier number; for each supported calling convention a corresponding symbolic constant is defined. Thereby two kinds of symbolic constants can be identified: language specific identifiers and calling convention specific identifiers. The former is available to *all* platforms while the latter should be used on particular target platform only.

Currently, three language-specific identifiers are defined:

- `DC_CALL_C_DEFAULT` refers to the standard C calling convention,
- `DC_CALL_C_ELLIPSIS` refers to the ellipsis calling convention and
- `DC_CALL_SYS_DEFAULT` refers to the system call calling convention.

In addition, symbolic constants for the following platform-specific calling conventions exists.

```
#define DC_CALL_C_X86_CDECL            1
#define DC_CALL_C_X86_WIN32_STD        2
#define DC_CALL_C_X86_WIN32_FAST_MS    3
#define DC_CALL_C_X86_WIN32_FAST_GNU   4
#define DC_CALL_C_X86_WIN32_THIS_MS    5 // C++ this calls
#define DC_CALL_C_X86_WIN32_THIS_GNU   6 // C++ this calls
#define DC_CALL_C_X86_PLAN9           19
#define DC_CALL_C_X64_WIN64            7
#define DC_CALL_C_X64_SYSV             8
#define DC_CALL_C_PPC32_DARWIN         9
#define DC_CALL_C_PPC32_SYSV          13
#define DC_CALL_C_ARM_ARM             14
#define DC_CALL_C_ARM_ARM_EABI        10
#define DC_CALL_C_ARM_THUMB           15
#define DC_CALL_C_ARM_THUMB_EABI      11
#define DC_CALL_C_MIPS32_O32          16
#define DC_CALL_C_MIPS32_EABI         12
#define DC_CALL_C_MIPS64_N32          17
#define DC_CALL_C_MIPS64_N64          18
#define DC_CALL_C_SPARC32             20
#define DC_CALL_C_SPARC64             21
#define DC_CALL_SYS_X86_INT80H_LINUX 201
#define DC_CALL_SYS_X86_INT80H_BSD   202
```

5.4.5.3 Phase 2: Load argument

Arguments are "loaded" via functions; their name begins with a common prefix `dcArg` followed by the C type name (in camel-case notation).

For each supported parameter type there is a interface function to specify the next argument value in the calling sequence; the parameter type is implicitly specified by the chosen function. The direction is always *left*-to-*right* in accordance to the declaration of the function type in the foreign language. During buffering of the sequence of argument types and values, internals of the Call VM perform the necessary operations, such as assignment to stack and register buffers, promotion of argument values to larger sizes, and also updating of internal state variables.

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

All integer data are regarded as `signed` data types; `unsigned` parameter types are indirectly supported via type casting to corresponding `signed` types in advance.

Intensive testing verified that this method is sufficient in practice except for one very specific test case[3] that was identified only recently and only for one specific platform (`ppc32`), compiler settings and value transfer (to provoke an overflow). Therefore, it is planed to extend the interface for `unsigned` data types in a new version of `dyncall`.

5.4.5.4 Phase 3: Call function

After all arguments have been buffered, the target function can be called by selecting the return type. For each supported return type there is a type-specific interface function to specify the return type (implied by the chosen function) and the address of the target function.

```
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);
```

Phase two and three of the interface are summarized in Table 5.10. For each supported C type corresponding “load argument” and “call” functions are offered by the interface, except for `void`, which gives no argument value.

5.4.5.5 Freeing resources

When a Call VM object is not needed, it can be destroyed via `dcFree`:

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

[3]The test case is available in the `test/sign` project folder in the source repository.

C Type	Argument Functions	Call Functions
parameters:	Parameter Type, Argument Value	Return Type, Function Address
prototyp pattern:	`void dcArg<TYPE>(state,<VALUE>)`	`<RTYPE> dcCall<RTYPE>(state,<ADDR>)`
`void`	-	`dcCallVoid`
`_Bool`	`dcArgBool`	`dcCallBool`
`char`	`dcArgChar`	`dcCallChar`
`short`	`dcArgShort`	`dcCallShort`
`int`	`dcArgInt`	`dcCallInt`
`long`	`dcArgLong`	`dcCallLong`
`long long`	`dcArgLongLong`	`dcCallLongLong`
`float`	`dcArgFloat`	`dcCallFloat`
`double`	`dcArgDouble`	`dcCallDouble`
any `*`	`dcArgPointer`	`dcCallPointer`

Table 5.10: C API of `dyncall` for the specification of calling sequences.

5.4.5.6 Invocation of C++ code

`dyncall` has support for C++ member function calls. The user has to pass the `this` pointer as first parameter. On most platforms, C++ member function calls are implemented using the `this` pointer as hidden first argument to function (the C++ method). However, on the `x86-32` platform, the `this` pointer needs to be placed in a specific register (ECX). Therefore, two calling conventions (and Call Kernels) are available that need to be selected in advance, namely the symbolic constants `DC_CALL_C_X86_THIS_MS` and `DC_CALL_C_X86_THIS_GNU`, which refer to calling conventions used by code generated with Microsoft's Visual C++ and GNU's GCC compilers, respectively.

5.4.5.7 Invocation of Variadic C Functions

Variadic C functions are supported by a separate calling convention driver, selected via `DC_CALL_C_ELLIPSIS`. After positional arguments are loaded, the mode needs to change to `DC_CALL_C_ELLIPSIS_VARARG`, which indicates that the arguments that follow are variadic. Note that, as of the current implementation, the user is still responsible for passing C types that are valid as the variable argument types. For the variadic part of a calling sequence, `float` is automatically promoted to `double` at the caller side. We have not yet implemented auto-promotion for ellipsis calls (as this would also double the amount of driver code).

5.4.5.8 Invocation of System Calls

Preliminary support for system calls is given for a small number of `x86-32` operating systems. System calls are selected by a numeric identifier instead of a function pointer; the number is specific to a particular operating system and identifies a system service. Currently, the user passes the number via type casting as a function pointer to `dcCallVoid` or `dcCallInt` functions. Note that the system call interface often uses a very limited number of arguments and often does only support a subset of return and argument types i.e. floating-point types are usually not supported as value types.

5.4.6 Sample Applications

In this section we give two examples for the implementation of a Dynamic FFI in C using the `dyncall` API.

5.4.6.1 Variadic function call interface for C

We give a simple example for a dynamic call interface for C. The interface is defined as a function named `tiny_dyncall`; the implementation is as follows:

```
#include "dyncall.h"
#include <stdarg.h>
typedef union { int i; float f; double d; void* p; } Result;
void tiny_dyncall(DCCallVM* vm, Result* res, void (*fun)(), char const * fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);
  dcReset(vm);
  while ( *fmt != ')' ) {
    switch(*fmt++) {
      case 'i': dcArgInt    (vm,         va_arg(ap, int   )); break;
      case 'f': dcArgFloat  (vm, (float) va_arg(ap, double)); break;
      case 'd': dcArgDouble (vm,         va_arg(ap, double)); break;
      case 'p': dcArgPointer(vm,         va_arg(ap, void* )); break;
    }
  }
  va_end(ap);
  switch(*fmt) {
    case 'v':          dcCallVoid   (vm, fun); break;
    case 'i': res->i = dcCallInt    (vm, fun); break;
    case 'f': res->f = dcCallFloat  (vm, fun); break;
    case 'd': res->d = dcCallDouble (vm, fun); break;
    case 'p': res->p = dcCallPointer(vm, fun); break;
  }
}
```

The interface consists of several positional parameters: a reference to a previously created CallVM, a pointer to a union data object for storing return values, a pointer to a C function to be called and a signature string that encodes the target C function type. Then the arguments to the target function are passed via `...`.

Variadic C functions are implemented by means of macros from the C standard header `stdarg.h` to iterate over the sequence of arguments passed after the last positional argument. The macros can be used to extract the arguments in a step-wise manner, but the type of each argument needs to be known. Therefore a descriptor, typically a C string, is mostly used as a positional parameter that gives the sequence of argument types. We use a signature string that describes the argument types, followed by a '`)`' and the return type as last character. Our interface supports five types: '`i`' for integers, '`p`' for pointers, '`f`' and '`d`' for single- and double-precision floating-point values, and '`v`' for `void` return types.

We briefly discuss the standard C macros for the implementation of variadic functions in C (defined in `stdarg.h`); in particular, we explain the iteration over the arguments indicated by `...`. The data type `va_list` is used as an iterator object for the variable argument list; it is initialized via the macro `va_start(va_list ap, last)` where `last` refers to the last positional argument. Arguments are extraced via the macro `va_arg(va_list ap, type)` in a step-wise manner. Finally the `va_end(va_list ap)` macro gets called to terminate the iteration.

During processing of the arguments, we call corresponding functions to load arguments to the Call VM. Note that we need to treat `float` argument types in a special way due to value promotion rules for ellipsis function calls of the C standard: the set of data types that are passed as `...` is limited to a subset of (promotion target) types; small-sized integer data types are promoted to `int` while floating-point values are promoted to `double`. Support for very large integers (e.g. `long long`) and floating-point values (e.g. `long double`) or non-standard data types of a particular platform is not specified by the C standard. As we aim to support `float` via `tiny_dyncall`, we use a small convention here: when a `float` is specified for a function call (as of a '`f`'), we assume it is passed auto-promoted as a `double` to `tiny_dyncall`, so that it is converted to `float` and then to be passed to `dyncall`'s Call VM.

After all arguments are loaded and the iteration terminates, we select an appropriate call function that corresponds to the final signature character and invoke the function call; we store the return value in a user-allocated union given by the pointer `res`.

A sample user code for our interface is given below:

```c
#include <math.h>
#include "SDL.h"
void usercode() {
  Result r;
  DCCallVM* vm = dcNewCallVM(4096);
  dcMode(vm, DC_CALL_C_DEFAULT);

  tiny_dyncall(vm, &r, sqrt, "d)d", 144.0);
  tiny_dyncall(vm, &r, SDL_Init, "i)i", SDL_INIT_VIDEO);
  tiny_dyncall(vm, &r, SDL_SetVideoMode, "iiii)p", 640,480,32,SDL_OPENGL);

  dcFreeCallVM(vm);
}
```

We have kept this example relatively simple to give a brief overview of the implementation structure. At first, the sequence of argument types and values are processed, and then the call is performed. Real-world implementations of Dynamic FFIs include support for more data types than were used in this example.

5.4.6.2 Application to the R interpreter

In this section we describe the implementation of the `.dyncall` FFI to R. As we outlined in Section 4.3.4, `.dyncall` is a front-end that delegates to `dyncall.<MODE>` depending on the `callmode` argument. In the following code we give the implementation for the default callmode.

```
.dyncall.default <- function( address, signature, ... )
{
  .External("dyncall", callvm.default, address, signature, ..., PACKAGE="rdyncall")
}
```

It is a front end to the C implementation, which specifies the Call VM as a second argument and delegates to the C implementation. For example, `.dyncall.stdcall` is very similar; `callvm.default` would be replaced with `callvm.stdcall` which refers to a Call VM configured to use the `stdcall` calling convention (on `x86-32` platforms). In the current implementation of the R package `rdyncall`, we use a set of `callvm.*` external pointer objects that represent a reference to a Call VM, i.e. `DCCallVM*`.

`.External` is used as the extension interface to C for passing a variable number of arguments to an R function implemented in C, named `"dyncall"` in R. Note that the parameter `PACKAGE` specifies the R package (and corresponding shared library) for resolving `"dyncall"` to the address of the C implementation `r_dyncall` and it is then dropped from argument list.

The following code gives the head of `r_dyncall` for extraction of positional arguments, such as `callvm.default`, `address` and `signature`:

```
SEXP r_dyncall(SEXP args) // args -> "dyncall", callvm, address, signature, ...
{
  args = CDR(args);       // NEXT ARG   args -> callvm, address, signature, ...
  DCCallVM *pvm=(DCCallVM*)R_ExternalPtrAddr(CAR(args)); // extract callvm
  args = CDR(args);       // NEXT ARG           args -> address, signature, ...
  if (!pvm) {                                        // ensure
    error("Argument 'callvm' is null"); return R_NilValue; }
  if (TYPEOF(CAR(args))!=EXTPTRSXP) {
    error("Target address must be external pointer."); return R_NilValue; }
  void *addr=R_ExternalPtrAddr(CAR(args));             // extract address
  if (!addr) {                                       // ensure addr is not NULL pointer
    error("Argument 'addr' is null"); return R_NilValue; }
  const char* signature=CHAR(STRING_ELT(CAR(args)0));    // extract signature
  args = CDR(args);       // NEXT ARG                       args -> ...
  const char* sig = signature;                       // init signature iterator
  dcReset(pvm);                                      // init callvm
  SEXP        arg;                                   //
  int         ptrcnt = 0;                            // init pointer counter (for prefixes)
  int         argpos = 0;                            // init positional counter
```

In the code the macros `CAR` and `CDR` are used to iterate over the S-Expression call object and for the extraction of arguments, respectively, followed by resetting the Call VM and assignment of local variables (counter for '`*`' prefixes and argument position counter).

In the following code section the head of the iteration loop is listed; the loop scans the signature string and stops when it encounters a '`)`' signature character (the end of the arguments). Prefixes of '`*`' increment a pointer prefix counter for the support of typed pointers. Otherwise an iteration step for the argument list is performed (we expect an object signature character).

```
  for(;;) {
    char ch = *sig++;           // next char
    if (ch == ')') break;       // ')' -> argument terminator
    if (ch == '\0') {           // end of string? (but we are still in the middle!)
      error("invalid signature '%s': missing ')'.", signature); return R_NilValue;
    }
    if (args == R_NilValue) {  // end of arguments? (but we already read a type code!)
      error("signature '%s': not enough args.",     signature); return R_NilValue;
    }
    else if (ch == '*') {      // pointer prefix?
      ptrcnt++; continue;      // increment ptrcnt and continue scanning
    }
    arg = CAR(args); args = CDR(args); argpos++;  // iteration-step over R args
    int type_id = TYPEOF(arg); // get R object type
```

If `ptrcnt` is greater than zero the type of the next argument is a pointer, otherwise it is a scalar specified by the next type signature character. First we handle the cases for scalar types; the conversion of R to C values is implemented via a nesting of two switch/cases blocks, where the outer block determines the target C type and the inner block determines the source R argument object type, so that a precise conversion can be implemented for each case.

Next we give the conversion to C `_Bool`, `int` and `float`, including a guard code that ensures that the length of atomic vectors is greater than zero:

```
    if (ptrcnt == 0) {                                 // base type (no pointer prefixes)
      // Assert that length() > 0 for anything other than NULL or externalptr.
      if ( type_id != NILSXP && type_id != EXTPTRSXP && LENGTH(arg) == 0 ) {
        error("arg type mismatch at pos %d: length() must be >0.", argpos);
        return R_NilValue;
      }
      switch(ch) {
        case 'B': { // Convert as _Bool
          DCbool boolValue;
          switch(type_id) {
            case LGLSXP:  boolValue = ( LOGICAL(arg)[0] == 0   ) ? DC_FALSE : DC_TRUE; break;
            case INTSXP:  boolValue = ( INTEGER(arg)[0] == 0   ) ? DC_FALSE : DC_TRUE; break;
            case REALSXP: boolValue = ( REAL(arg)[0]    == 0.0 ) ? DC_FALSE : DC_TRUE; break;
            case RAWSXP:  boolValue = ( RAW(arg)[0]     == 0   ) ? DC_FALSE : DC_TRUE; break;
            default:
             error("arg type mismatch at pos %d: expected C bool convertable value", argpos);
             return R_NilValue;
          }
          dcArgBool(pvm, boolValue );
        } break;
        case 'i': { // Convert as 'int'
          int intValue;
          switch(type_id) {
            case LGLSXP:  intValue = (int) LOGICAL(arg)[0]; break;
            case INTSXP:  intValue = INTEGER(arg)[0]; break;
            case REALSXP: intValue = (int) REAL(arg)[0]; break;
            case RAWSXP:  intValue = (int) RAW(arg)[0]; break;
            default:
             error("arg type mismatch at pos %d: expected C int convertable value", argpos);
             return R_NilValue;
          }
          dcArgInt(pvm, intValue);
        } break;
        case 'f': { // Convert as 'float'
          float floatValue;
          switch(type_id)
          {
            case LGLSXP:  floatValue = (float) LOGICAL(arg)[0]; break;
            case INTSXP:  floatValue = (float) INTEGER(arg)[0]; break;
            case REALSXP: floatValue = (float) REAL(arg)[0]; break;
            case RAWSXP:  floatValue = (float) RAW(arg)[0]; break;
            default:
              error("arg type mismatch at pos %d: expected C float convertable value",argpos);
              return R_NilValue;
          }
          dcArgFloat( pvm, floatValue );
        }
        break;
```

Note, that no R object is created during value conversion; the goal is to minimize interactions with a garbage collector for each call. Thus no objects are created due to value conversion. We omit the handling for other types; its implementation is very similar. The next code section gives the conversion for typed pointers to `struct` / `union` data types that are prefixed via '`*`', followed by a '`<`', followed by a type alias name, and terminated by '`>`'in order to support the framework for handling foreign data types (described in Section 4.5).

```
    else if (ptrcnt == 1) {
      DCpointer ptrValue;                     // temp storage for pointer value

      if (ch == '<') {                        // pattern: '<' .. typename .. '>'
        switch(type_id) {  // set value support NULL, externalptr and RAW values:
          case NILSXP:    ptrValue = (DCpointer) 0; break;
          case EXTPTRSXP: ptrValue = R_ExternalPtrAddr(arg); break;
          case RAWSXP:    ptrValue = (DCpointer) RAW(arg); break;
          default:
            error("internal error: typed-pointer must be NULL,externalptr or raw only.");
            return R_NilValue;
        }

        char const * b = sig;                 // mark begin      ^
        while(isalnum(*sig)||*sig=='_')sig++;  // scan name    ([A-Za-z0-9_]*)
        if (*sig != '>') {                    // we expect a '>' -------------^
          error("Invalid signature '%s' at pos '%d': missing '>'.", signature, argpos);
          return R_NilValue;
        }
        sig++;
                                              // check pointer type:
        SEXP structName = getAttrib(arg, install("struct"));
        if (structName == R_NilValue) {
          error("typed pointer needed here");
          return R_NilValue; /* Dummy */
        }
        char const * e = sig-1;
        int l = e - b;
        char cosnt * n = CHAR(STRING_ELT(structName,0));
        if ( (strlen(n) != l) || (strncmp(b,n,l) != 0) ) {
          error("incompatible pointer types");
          return R_NilValue; /* Dummy */
        }
        dcArgPointer(pvm, ptrValue);         // load pointer value to Call VM
        ptrcnt = 0;                          // reset pointer counter
      }
```

In summary, the code checks that external pointers or raw vectors contain an attribute `'struct'` that equals the name between '`<`' and '`>`' in the signature. Otherwise the case for typed pointers to scalars is handled, i.e. the signature begins with a '`*`', followed by a signature character for a scalar type. The implementation also uses two levels of switch/case blocks, similar to the implementation for passing C scalar values. Since we pass pointers here to the internal C array storage of R atomic vectors, the mapping rules are strict and effectively implement the rules described in Section 4.3.6.1.

We include the case of '`v`' that refers to the `void` non-type and thus is a `void*` argument type which equals to the code for handling just '`p`'. We always accept the `NULL` and `externalptr` as values for any typed pointer signature. The `raw` object is a fallback for any typed pointers where no matching implementation type in R exist, such as for `float*` where users need to pass in R objects created with the helper function `as.floatraw`.

```
      else { // sigchar is not '<'.. so we expect a type character.
        switch(ch) {
          case 'v':
            switch(type_id) {
              case NILSXP:    ptrValue = (DCpointer) 0; break;
              case STRSXP:    ptrValue = (DCpointer) CHAR(STRING_ELT(arg,0)); break;
              case LGLSXP:    ptrValue = (DCpointer) LOGICAL(arg); break;
              case INTSXP:    ptrValue = (DCpointer) INTEGER(arg); break;
              case REALSXP:   ptrValue = (DCpointer) REAL(arg); break;
              case CPLXSXP:   ptrValue = (DCpointer) COMPLEX(arg); break;
              case RAWSXP:    ptrValue = (DCpointer) RAW(arg); break;
              case EXTPTRSXP: ptrValue = R_ExternalPtrAddr(arg); break;
              default:
                error("arg type mismatch at pos %d: expected ptr convertable value", argpos);
                return R_NilValue;
            }
            break;
          case 'f':
            switch(type_id) {
              case NILSXP:  ptrValue = (DCpointer) 0; break;
              case RAWSXP:
                if (strcmp(CHAR(STRING_ELT(getAttrib(arg,install("class")),0)),"floatraw")==0){
                  ptrValue = (DCpointer) REAL(arg);
                }
                break;
              default:
                error("arg type mismatch at pos %d: expected 'floatraw'", argpos);
                return R_NilValue;
            }
            break;
          case 'c':
          case 'C':
            switch(type_id)
            {
              case STRSXP:    ptrValue = (DCpointer) CHAR( STRING_ELT(arg,0) ); break;
              case NILSXP:    ptrValue = (DCpointer) NULL;                      break;
              case RAWSXP:    ptrValue = (DCpointer) RAW(arg);   break;
              case EXTPTRSXP: ptrValue = R_ExternalPtrAddr(arg); break;
              default:
                error("arg type mismatch at pos %d: expected 'C str' convertable value",argpos);
                return R_NilValue;
            }
            break;
          case 'd':
            switch(type_id)
            {
              case NILSXP:  ptrValue = (DCpointer) 0; break;
              case REALSXP: ptrValue = (DCpointer) REAL(arg); break;
              case EXTPTRSXP: ptrValue = R_ExternalPtrAddr(arg);   break;
              default:
                error("arg type mismatch at pos %d: expected 'C double' convertable value", argpos);
                return R_NilValue;
            }
            break;
          case 'I':
          case 'i':
            switch(type_id)
            {
              case NILSXP:    ptrValue = (DCpointer) NULL;         break;
              case INTSXP:    ptrValue = (DCpointer) INTEGER(arg); break;
              case EXTPTRSXP: ptrValue = R_ExternalPtrAddr(arg);   break;
              default:
                error("arg type mismatch at pos %d: expected 'int*' convertable value", argpos);
                return R_NilValue;
            }
            break;
          default:
            error("low-level typed pointer on C char pointer not implemented");
            return R_NilValue;
        }
        dcArgPointer(pvm, ptrValue);
        ptrcnt = 0;
      }
    }
  }
```

At this point the argument iteration is finished as we approached to the '`)`'. The end of arguments is checked next.

```
  if (args != R_NilValue) {
    error ("Too many arguments for signature '%s'.", signature);
    return R_NilValue; /* dummy */
  }
```

Now the Call VM has been successfully loaded with arguments and we can finally invoke the foreign function and return the value as an appropriate R object.

```
  switch(*sig++) {
    case 'B':return ScalarLogical((dcCallBool(pvm,addr)==DC_FALSE)?FALSE:TRUE);
    case 'c':return ScalarInteger((int)dcCallChar(pvm,addr)  );
    case 'C':return ScalarInteger((int)((unsigned char)dcCallChar(pvm, addr ) ) );
    case 's':return ScalarInteger((int)dcCallShort(pvm,addr) );
    case 'S':return ScalarInteger((int)((unsigned short)dcCallShort(pvm,addr) ) );
    case 'i':return ScalarInteger(dcCallInt(pvm,addr) );
    case 'I':return ScalarReal((double)(unsigned int) dcCallInt(pvm, addr) );
    case 'j':return ScalarReal((double)dcCallLong(pvm, addr) );
    case 'J':return ScalarReal((double)((unsigned long) dcCallLong(pvm, addr) ) );
    case 'l':return ScalarReal((double)dcCallLongLong(pvm, addr) );
    case 'L':return ScalarReal((double)dcCallLongLong(pvm, addr) );
    case 'f':return ScalarReal((double)dcCallFloat(pvm,addr) );
    case 'd':return ScalarReal(dcCallDouble(pvm,addr));
    case 'p':return R_MakeExternalPtr(dcCallPointer(pvm,addr),R_NilValue,R_NilValue);
    case 'Z':return mkString(dcCallPointer(pvm,addr));
```

It is also possible that the return type is in prefixed and typed pointer notation. We give the implementation for typed pointers, which returns an external pointer that is tagged with a `struct` attribute.

```
    case '*': {
      SEXP ans;
      ptrcnt = 1;
      while (*sig == '*') { ptrcnt++; sig++; }  // count pointers.
      switch(*sig) {
        case '<': {
          /* struct/union pointers */
          PROTECT(ans = R_MakeExternalPtr(dcCallPointer(pvm,addr),R_NilValue,R_NilValue));
          char buf[128];
          const char* b = ++sig;
          const char* e= strchr(sig, '>');
          size_t n = e - b;
          strncpy(buf, b, n);
          buf[n] = '\0';
          setAttrib(ans, install("struct"), mkString(buf) );
          setAttrib(ans, install("class"), mkString("struct") );
        } break;
        case 'C':
        case 'c': {
          PROTECT(ans = mkString(dcCallPointer(pvm,addr)));
        } break;
        default: error("Unsupported returnt type signature"); return R_NilValue;
      }
      UNPROTECT(1);
      return(ans);
    }
    default: error("Unknown return type specification for signature '%s'.", signature);
             return R_NilValue; /* dummy */
  }
}
```

Since the handling of typed pointers is not completed for all combinations when returning, we aim to offer a much more general approach in a future version, where the returned pointer is always passed as an external pointer that is tagged with the desired pointer type.

5.4.7 Implementation

In this section we discuss the implementation of `dyncall` and details of back-end ports to several calling conventions on the supported processor architectures. For some target platforms only one calling convention is available in `dyncall`; others offer support for multiple calling conventions that can be changed during run time. An overview of available ports is given in Table 5.11.

Arch/OS	Microsoft Windows	Apple	Linux,(Net,Free,Open) BSD	Solaris	Dragonfly BSD	Haiku	Minix 3	Plan9	Nintendo DS	Sony PSP
`x86-32`	`cdecl`,`stdcall`,`fastcall`(ms/gnu),`thiscall`(ms/gnu)							`p9`	-	-
`x86-64`	`x64`	`sysv`				-	-	-	-	-
`ppc32`	-	`osx`	`sysv`	-	-	-	-	-	-	-
`arm32`	-	`aapcs`,`armhf`		-	-	-	-	-	`apcs`	-
`mips32`	-	-	`o32`	-	-	-	-	-	-	`eabi`
`mips64`	-	-	`n64`	-	-	-	-	-	-	-
`sparc32`	-	-	`v7/v8`		-	-	-	-	-	-
`sparc64`	-	-	`v9`		-	-	-	-	-	-

Table 5.11: Overview of supported platforms by `dyncall`.

`dyncall` is implemented mostly in C for the Call VM and sparsely in assembly language for the call kernel. The front-end API functions delegate calls to methods, implemented in C, to promote, align and assign data to internal buffers for registers and/or the stack. Call kernels represent generic call services for arbitrary calling sequences but specific ABI and calling convention. They receive pointers to data buffers and buffer sizes to copy data to registers and the call stack, and a function pointer to execute the call.

Most call-kernel implementations consist of short code blocks that create new stack frames by decrementing the *stack pointer* register, followed by instructions to copy data from a buffer to the newly allocated region on the stack. For some calling conventions alignment restrictions of stack frames are handled by assembly code. For those calling conventions that use registers for passing arguments the call kernels contain a section of code with a fixed size sequences of "`load`" instructions to transfer data from buffers to register storage. Some architectures use a FPU with overlapping floating-point register formats. In those cases type-specific load instructions are embedded in conditional blocks; selection of whether single-precision or double-precision values are loaded is based on a bit mask managed from

C. Furthermore some calling conventions use homing areas that need to be reserved on the stack for some parameter registers.

The source code of `dyncall` comprises

- a common code base, including
  - API front-end delegation to back-ends in C (`dyncall_api.c`) and
  - a common implementation of the argument buffer in C(`dyncall_vector.c`), and
- back-end implementations for various architecture-specific calling conventions, including
  - one or more Call VM state machine(s) in C (`dyncall_callvm_<ABI>.c`)
  - one or more Call kernel(s) in assembly language (`dyncall_call_<ABI>.[S|s|asm]`).

Source code file name patterns to be found in the source package are given in parenthesis.

5.4.7.1 Anatomy of Call VM objects

The Call VM is represented as a fixed size data object that usually comprises only few kilobytes. It begins with a common header structure followed by port-specific data fields. The rest of the object is reserved for a vector to buffer the variable sequence of argument data to be copied to the stack. The header is declared in C as follows:

```
typedef struct {
  DCCallVM_vt* mVTpointer;
  DCint        mError;
} DCCallVM;
```

The first field `mVTpointer` is used to determine the current *mode*, i.e. the *driver* for a specific calling convention. The pointer references a table of function pointers which - on their part - specify the implementation for an API function. The function table `DCCallVM_vt` is declared as follows:

```
typedef struct {
  void       (*free)           (DCCallVM* vm);
  void       (*reset)          (DCCallVM* vm);
  void       (*mode)           (DCCallVM* vm,DCint      modeId);
  void       (*argBool)        (DCCallVM* vm,DCbool      value);
  void       (*argChar)        (DCCallVM* vm,DCchar      value);
  void       (*argShort)       (DCCallVM* vm,DCshort     value);
  void       (*argInt)         (DCCallVM* vm,DCint       value);
  void       (*argLong)        (DCCallVM* vm,DClong      value);
  void       (*argLongLong)    (DCCallVM* vm,DClonglong  value);
  void       (*argFloat)       (DCCallVM* vm,DCfloat     value);
  void       (*argDouble)      (DCCallVM* vm,DCdouble    value);
  void       (*argPointer)     (DCCallVM* vm,DCpointer   value);
  void       (*callVoid)       (DCCallVM* vm,DCpointer funcptr);
  DCbool     (*callBool)       (DCCallVM* vm,DCpointer funcptr);
  DCchar     (*callChar)       (DCCallVM* vm,DCpointer funcptr);
  DCshort    (*callShort)      (DCCallVM* vm,DCpointer funcptr);
  DCint      (*callInt)        (DCCallVM* vm,DCpointer funcptr);
  DClong     (*callLong)       (DCCallVM* vm,DCpointer funcptr);
  DClonglong (*callLongLong)   (DCCallVM* vm,DCpointer funcptr);
  DCfloat    (*callFloat)      (DCCallVM* vm,DCpointer funcptr);
  DCdouble   (*callDouble)     (DCCallVM* vm,DCpointer funcptr);
  DCpointer  (*callPointer)    (DCCallVM* vm,DCpointer funcptr);
} DCCallVM_vt;
```

All functions of the `dyncall` API, except `dcNewCallVM`, are implemented as *methods* of a Call VM object. Front-end API functions delegate to methods referenced by the function pointer table. For example, the `dcArgDouble` API front-end function is defined as follows:

```
void dcArgDouble(DCCallVM* vm, DCdouble x) {
  vm->mVTpointer->argDouble(vm, x);
}
```

For each supported calling convention a separate function pointer table is allocated and initialized statically at compile time. In order to switch between several calling conventions (or modes) the `mVTpointer` field is adjusted to point to a different function table.

The second field of the common header, named `mError`, is used to indicate an error status. Currently only the function `dcMode`, to select the mode, sets this field to indicate an error if the requested calling convention is not supported.

The vector for buffering arguments is implemented in C as a common code base available to all ports of the library. The beginning of the vector within the Call VM data object is marked by a vector header structure, which is declared as follows:

```
typedef struct {
  DCsize mTotal;
  DCsize mSize;
} DCVecHead;
```

Several C macros and functions for vector management, declared in `dyncall_vector.h`, are listed below:

```
#define dcVecInit(p,size)   (p)->mTotal=size;(p)->mSize=0
#define dcVecReset(p)       (p)->mSize=0

void    dcVecAppend(DCVecHead* pHead, const void* source, size_t length);

#define dcVecResize(p,size) (p)->mSize=(size)
#define dcVecSkip(p,size)   (p)->mSize+=(size)
#define dcVecData(p)        ( (unsigned char*) (((DCVecHead*)(p))+1) )
#define dcVecAt(p,index)    ( dcVecData(p)+index )
#define dcVecSize(p)        ( (p)->mSize )
```

The vector is initialized via a macro call to `dcVecInit` with a pointer to the vector header and the total amount of memory available for the buffer; the latter is typically set to

total size of the object − size of all headers.

The field `mSize` keeps track of the current size of data within the buffer and it functions as a *write* pointer that is initialized to zero by `dcVecInit` and `dcVecReset`, and is incremented while appending data via `dcVecAppend`.

The implementation of the vector was intentionally held simple; detection of buffer overflows was omitted and left open to users of the library, such as developers of dynamic FFIs who can decide to limit the number of arguments. However, we suggest to allocate a few kilobytes of memory for a Call VM object, so that the vector can handle all the kinds of sequences that are encountered in practice. For example, 4096 bytes is enough memory to buffer from 512 to 1024 arguments (depending on the type), a much larger number of arguments then exist in public C libraries.

We now discuss several examples of back-end implementations. We begin with simple calling conventions and then proceed to the more complex ones. During this discussion we give excerpts of `dyncall` sources partly in modified form (i.e. shorter symbolic names) to improve legibility of core implementation concepts.

5.4.8 Passing arguments via the stack (x86-32 ports)

One of the most basic calling conventions uses only the stack to pass all arguments. Some calling conventions for the x86-32 architecture, such as cdecl and stdcall, adopted this basic scheme. The Call VM's vector is sufficient as main data structure to buffer different sequences of incoming argument data; no further state variables are need other than the current size of the vector. All arguments are appended to the vector during the loading phase; this matches the *right to left* "stack push order" rule of cdecl and stdcall. Stack values need to be aligned to 4-byte boundaries - a core condition for call stacks on the x86-32 architecture. For small integer arguments (smaller than 4 bytes) the value is promoted to 4-byte integer words in advance. Other supported argument types are already multiples of 4-byte quantities and are appended as-is.

```
typedef struct
{
  DCCallVM  mInterface;   /* callvm header */
  int       mIntRegs;     /* fastcall specific: used integer registers */
  DCVecHead mVecHead;     /* vector header */
} DCCallVM_x86;
```

Listing 20: Common Call VM structure shared by x86-32 call kernels.

Listing 20 gives the C declaration of the Call VM data object for the x86-32 architecture. The head of the structure consists of the common header `DCCallVM mInterface`, a port-specific data field `int mIntRegs` and ends with the header of the vector `DCVecHead mVecHead`. In order to switch between calling conventions on x86-32 at run time all port-specific data fields are collectively held in the data structure. The field `mIntRegs` is included because it is used by driver code of the fastcall calling convention, which is discussed in Section 5.4.10; for now we can ignore this variable.

In dyncall, driver code for calling conventions is implemented by means of function tables. Listing 21 gives the C source code for the implementation of cdecl and stdcall drivers. Both tables - `gVT_x86_cdecl` and `gVT_x86_stdcall` - are preset with a similar set of implementation methods except for the `call<RTYPE>` slot which is set to call-kernel specific delegate functions.

The `reset` method sets the header field `mSize` of the vector to zero via `dcVecReset`. Implementations for loading small integers, such as `aBool`, `aChar` and `aShort`, first promote argument values to 32-bit `int` values and then delegate the call to the `aInt` function, which appends 4 bytes of data to the back of the vector. Other method implementations, such as `aDouble` and `aLongLong`, append data as-is.

The same delegate functions to call kernels, namely `call_cdecl` or `call_stdcall`, are each used for an entire set of `call<RTYPE>` slots regardless of the different return types. Note that the actual return type of the delegate function is `void` in C. However, the passing of return values proceeds here as expected; after the user invokes one of the `dcCall<RTYPE>` API functions the return value is transferred correctly. Recall that C functions usually pass return values via registers and, since both the delegate function and call kernel do not modify these registers after the foreign function call was executed, the caller will find the return value in the register that is reserved for this purpose.

```
void destroy  (DCCallVM* p)                { dcFreeMem(p); }
void reset    (DCCallVM* p)                { dcVecReset(&((DCCallVM_x86*)p)->mVecHead); }
void aInt     (DCCallVM* p, DCint      v) { dcVecAppend( &((DCCallVM_x86*)p)->mVecHead, &v, sizeof(DCint)     );}
void aBool    (DCCallVM* p, DCbool     v) { aInt(p, (DCint) v); }
void aChar    (DCCallVM* p, DCchar     v) { aInt(p, (DCint) v); }
void aShort   (DCCallVM* p, DCshort    v) { aInt(p, (DCint) v); }
void aLong    (DCCallVM* p, DClong     v) { aInt(p, (DCint) v); }
void aLongLong(DCCallVM* p, DClonglong v) { dcVecAppend( &((DCCallVM_x86*)p)->mVecHead, &v, sizeof(DClonglong));}
void aFloat   (DCCallVM* p, DCfloat    v) { dcVecAppend( &((DCCallVM_x86*)p)->mVecHead, &v, sizeof(DCfloat)   );}
void aDouble  (DCCallVM* p, DCdouble   v) { dcVecAppend( &((DCCallVM_x86*)p)->mVecHead, &v, sizeof(DCdouble)  );}
void aPointer (DCCallVM* p, DCpointer  v) { dcVecAppend( &((DCCallVM_x86*)p)->mVecHead, &v, sizeof(DCpointer) );}
void call_cdecl (DCCallVM* p, DCpointer target) {
  DCCallVM_x86* vm = (DCCallVM_x86*) p;
  x86_cdecl  ( target, dcVecData(&vm->mVecHead), dcVecSize(&vm->mVecHead) );
}
void call_stdcall (DCCallVM* x, DCpointer target) {
  DCCallVM_x86* vm = (DCCallVM_x86*) p;
  x86_stdcall( target, dcVecData(&vm->mVecHead), dcVecSize(&vm->mVecHead) );
}
DCCallVM_vt gVT_x86_cdecl = {
  &destroy, &reset, &mode, &aBool, &aChar, &aShort, &aInt, &aLong, &aLongLong, &aFloat, &aDouble, &aPointer
, (DCvoidvmfunc     *) &call_cdecl, (DCboolvmfunc     *) &call_cdecl, (DCcharvmfunc     *) &call_cdecl
, (DCshortvmfunc    *) &call_cdecl, (DCintvmfunc      *) &call_cdecl, (DClongvmfunc     *) &call_cdecl
, (DClonglongvmfunc *) &call_cdecl, (DCfloatvmfunc    *) &call_cdecl, (DCdoublevmfunc   *) &call_cdecl
, (DCpointervmfunc  *) &call_cdecl
};
DCCallVM_vt gVT_x86_stdcall = {
  &destroy, &reset, &mode, &aBool, &aChar, &aShort, &aInt, &aLong, &aLongLong, &aFloat, &aDouble, &aPointer
, (DCvoidvmfunc     *) &call_stdcall, (DCboolvmfunc     *) &call_stdcall, (DCcharvmfunc     *) &call_stdcall
, (DCshortvmfunc    *) &call_stdcall, (DCintvmfunc      *) &call_stdcall, (DClongvmfunc     *) &call_stdcall
, (DClonglongvmfunc *) &call_stdcall, (DCfloatvmfunc    *) &call_stdcall, (DCdoublevmfunc   *) &call_stdcall
, (DCpointervmfunc  *) &call_stdcall
};
```

Listing 21: Call VM for `cdecl` calling convention on `x86-32`.

While most API functions are implemented by means of the function table, the `dcNewCallVM` is the only front-end API function that is implemented directly by a back-end port. We discuss its implementation by example of the `x86-32` port of `dyncall`:

```
DCCallVM* dcNewCallVM(DCsize size)
{
  DCCallVM_x86* p = (DCCallVM_x86*) dcAllocMem( sizeof(DCCallVM_x86)+size );
  p->mVTpointer = &gVT_x86_cdecl;
  p->mError     = DC_ERROR_NONE;
  dcVecInit(&p->mVecHead, size);
  return (DCCallVM*) p;
}
```

Each port provides its own implementation of this API function for the construction of a Call VM object. The total size of the object is determined by the user-defined `size` of the vector adjusted by the size of the header structure. Memory for the object is allocated from heap memory via `dcAllocMem` - an alias to C's dynamic memory allocator `malloc`. The object is initialized by setting the pointer to the function table of the driver for the default C calling convention `cdecl`, and by initializing the vector via `dcVecInit`.

Listing 22 gives the implementation of the `mode` slot for `x86-32` Call VMs to switch between calling convention drivers. Predefined macros are often used to set platform-specific driver code for abstract language-specific modes at compile time. For example, the `DC_CALL_SYS_DEFAULT` identifier selects either a Linux or BSD system-call driver, depending on the target operating-system at compile time, as selected by predefined macros such as `DC__OS_Linux`.

```
void mode(DCCallVM* in_self, DCint mode)
{
  DCCallVM_x86* self = (DCCallVM_x86*) in_self;
  DCCallVM_vt*  vt;
  switch(mode) {
    case DC_CALL_C_ELLIPSIS:
    case DC_CALL_C_ELLIPSIS_VARARGS:
    case DC_CALL_C_DEFAULT:
    case DC_CALL_C_X86_CDECL:          vt = &gVT_x86_cdecl;            break;
    case DC_CALL_C_X86_WIN32_STD:      vt = &gVT_x86_stdcall;          break;
    case DC_CALL_C_X86_WIN32_FAST_MS:  vt = &gVT_x86_fast_ms;          break;
    case DC_CALL_C_X86_WIN32_THIS_MS:  vt = &gVT_x86_this_ms;          break;
    case DC_CALL_C_X86_WIN32_FAST_GNU: vt = &gVT_x86_fast_gnu;         break;
    case DC_CALL_C_X86_WIN32_THIS_GNU: vt = &gVT_x86_cdecl;            break;
# if defined DC_UNIX
    case DC_CALL_SYS_DEFAULT:
#   if defined DC__OS_Linux
      vt = &gVT_x86_sys_int80h_linux; break;
#   else
      vt = &gVT_x86_sys_int80h_bsd; break;
#   endif
#endif
    default:
      self->mInterface.mError = DC_ERROR_UNSUPPORTED_MODE; return;
  }
  self->mInterface.mVTpointer = vt;
}
```

Listing 22: Call VM mode switching on `x86-32`

We now take a closer look at the implementation of the call kernels for `cdecl` and `stdcall`. Their interface from C is declared as follows:

```
void x86_cdecl ( void (*target)(), void* buffer, size_t size);
void x86_stdcall ( void (*target)(), void* buffer, size_t size);
```

Both call kernels receive a pointer to the `target` function, a pointer to the argument `buffer` and the `size` of the vector. At first each routine allocates a new region on the stack via decrementing the stack pointer by the `size`. Then the contents of the argument buffer vector are copied from `buffer` to the newly allocated region on the stack. Finally the call to the `target` function pointer is executed. On return of the call return values are stored in corresponding registers and remain unmodified. Then the caller stack frame is removed (only for `cdecl` call kernels) and finally control flow is passed back to the call kernel's caller.

Listing 23 gives the assembly source for both call kernels.

```
 1    .text
 2    .globl x86_cdecl
 3  x86_cdecl:
 4    pushl %ebp          # prolog
 5    movl  %esp,%ebp
 6    pushl %esi          # preserve
 7    pushl %edi
 8    movl 12(%ebp),%esi # esi = buffer
 9    movl 16(%ebp),%ecx # ecx = size
10    addl $ 15,%ecx      #
11    andl $-16,%ecx      # ecx = align(ecx,16)
12    movl %ecx,16(%ebp) # save ecx to stack
13    subl %ecx,%esp      # alloc stack frame
14    movl %esp,%edi      # edi = top of stack
15    rep movsb           # copy data loop
16    call *8(%ebp)       # call
17    addl 16(%ebp),%esp # caller cleans stack!
18    popl %edi           # restore
19    popl %esi
20    movl %ebp,%esp      # epilog
21    popl %ebp
22    ret
```

```
 1    .text
 2    .globl x86_stdcall
 3  x86_stdcall:
 4    pushl %ebp          # prolog
 5    movl  %esp,%ebp
 6    pushl %esi          # save preserved
 7    pushl %edi
 8    movl 12(%ebp),%esi # esi = buffer
 9    movl 16(%ebp),%ecx # ecx = size
10
11                        # no alignment
12
13    subl %ecx,%esp      # alloc stack frame
14    movl %esp,%edi      # edi = top of stack
15    rep movsb           # copy data loop
16    call *8(%ebp)       # call
17                        # callee cleans!
18    popl %edi           # restore preserved
19    popl %esi
20    movl %ebp,%esp      # epilog
21    popl %ebp
22    ret
```

Listing 23: Call kernel for the `cdecl` (*left*) and the `stdcall` (*right*) on the `x86-32` architecture.

The call kernel begins with a prolog code (line 4-7) to set up a local stack frame. First the frame pointer EBP is saved on the stack (line 4) before it is loaded with the stack pointer ESP (line 5). Then the source and destination registers, ESI and EDI, are saved (line 6-7) since they are needed later in the code for memory copying operations. The three arguments to the call kernels are accessed relative to the frame pointer EBP; the stack layout (after execution of the prolog section) is given in the next table:

Offset	Content
EBP+16	Arg 3: `size`
EBP+12	Arg 2: `buffer`
EBP+8	Arg 1: `target`
EBP+4	Return address
EBP+0	Saved EBP
ESP+4	Saved ESI
ESP+0	Saved EDI

The source register ESI is loaded with the `buffer` pointer argument (line 8), and the counter register ECX is loaded with the total `size` of bytes to be copied (line 9).

Since the `cdecl` calling convention for Mac OS X on `x86-32` is slightly modified and requires that the stack is aligned on a 16-byte boundary, we adopted this scheme for `cdecl` in general here. The call kernel's local frame is of size 16 bytes as depicted in the table above. Therefore the `size` of the new stack frame is adjusted to the next multiple of 16 bytes (line 10 to 11). The adjusted `size` is saved back to the parameter location on the stack (line 12) for later reuse.

A new stack frame is allocated by decrementing the stack pointer ESP (line 13); its location is also loaded in the destination register EDI (line 14). Data is copied using a single *string memory copy* instruction with a loop prefix `rep` (line 15) - a pecularity of the `x86` ISA; bytes are copied in a loop from a source to a destination address, given by ESI and EDI registers, respectively. After each round both registers are incremented by one and the loop counter ECX is decremented by one. The loop breaks when ECX reaches zero.

The `target` function ([EBP+8]) is called (line 16). When the control flow returns (line 17) the return values are stored in corresponding registers such as EAX, EDX and ST(0); it needs to be ensured that these registers remain unmodified until control flow and the results in registers are passed back (line 22) so that return values are passed along via the call delegate to finally reach the user of the `dcCall*` C API functions. The `cdecl` call kernel removes the stack frame of the previous call by incrementing the ESP again (line 17). All preserved registers are restored to their original content, including frame and stack pointers (line 18 to 21). Finally, the stack pointer points to the return address and control flow is returned back via `ret` instruction (line 22).

The code of both call kernels is very similar except for two tasks which are implemented only by `cdecl`: code to ensure the stack is aligned to 16 bytes (line 10 to 12), and code to clean up the stack frame (line 17); the source code lines of `stdcall` are left intentionally blank here.

5.4.9 Passing arguments via registers (`arm32-softfloat` ports)

Major calling conventions for the ARM architecture utilize a subset of general-purpose registers to pass arguments. Even ARM hardware platforms with FPU registers do not use them for passing arguments in this scheme. More recently the `armhf` ABI (ARM hardfloat) was defined which utilizes FPU co-processor registers as was discussed in Section 5.3.3. In this section we focus on the implementation of ARM driver back ends for softfloat calling conventions that use only general-purpose registers.

A group of calling conventions, namely `apcs-gnu` and `linux-aapcs`, logically arranges argument values on a linear region of memory, similar to the `x86-32-cdecl` ABI. However, the first 16 bytes of storage are assigned to four registers, R0 to R3, while the rest (if any) is assigned to stack memory. Note that all kinds of data types are passed via general-purpose registers regardless of the data type such as floating-point values or large data; the latter can be passed eventually split between register and stack memory.

ARM supports two different ISAs, namely `arm` and `thumb`, so that two implementations of call kernels were implemented in two different assembly languages.

The data structure of the Call VM for ARM is very similar to that of `x86-32` and requires only the vector for its implementation:

```
typedef struct {
  DCCallVM  mInterface;
  DCVecHead mVecHead;
} DCCallVM_arm32;
```

The C interface of each call kernel is also similar to that of the `x86-32` architecture:

```
void arm32_arm (void (*target)(), void* buffer, size_t size);
void arm32_thumb(void (*target)(), void* buffer, size_t size);
```

Both call kernels load the first 16 bytes of the `buffer` to registers R0-R3; the rest is copied on the stack. Listing 24 gives the ARM call kernel implementation for `arm` ISA and `thumb` ISA. Both codes implement the same tasks by means of the respective ISA. We discuss the `arm` ISA code (*left* side) first.

As one of the first steps of the `arm` code a set of preserved registers R4-R12 and the return address in the link register (R14) are saved to the stack (line 6). The three arguments to the call kernel are copied to R4-R6 from R0-2 (line 7 to 9) to make the latter available for the next call. Then the first four words of the argument `buffer` (R5) are loaded to parameter registers R0-R3 and R5 is incremented accordingly by 16 bytes (line 11). R6 (`size` parameter) is decremented by 16 bytes to give the number of bytes to be copied to the stack (line 12). If this number is less than or equal to zero, the data copying step can be skipped (line 13). Otherwise the stack pointer is adjusted and aligned to an 8-byte boundary (line 14 to 16). Data are copied from the buffer to the stack in a loop (line 22 to 27). Finally

```
 1    .text
 2    .code 32
 3    .globl arm32_arm
 4  arm32_arm:
 5    mov %r12, %r13 # r12 = stack pointer on entry
 6    stmdb %r13!,{%r4-%r12,%r14} # save registers
 7    mov %r11, %r12 # r11 = stack pointer on entry
 8    mov %r4 , %r0  # r4  = target function (=arg 1:r0)
 9    mov %r5 , %r1  # r5  = argument buffer (=arg 2:r1)
10    mov %r6 , %r2  # r6  = size            (=arg 3:r2)
11    ldmia %r5!, {%r0-%r3} # load arg 1-4 to regs
12    subs %r6, %r6, #16    #  and decrement size
13    ble .call             # is less than zero?
14    sub %r13, %r13, r6    # copy arguments to stack
15    and %r9 , %r6 , #7    # align
16    sub %r13, %r13, r9
17    mov %r8 , %r13
18    mov %r9 , #0          # init copy counter to 0
19
20
21
22  .copy:                  # copy loop
23    ldrb %r7, [%r5, %r9]
24    strb %r7, [%r8, %r9]
25    add  %r9, %r9, #1
26    cmp  %r9, %r6
27    bne .copy
28  .call:
29    mov %r14, %r15        # r14 = return address
30    bx  %r4               # branch-and-link (armv4t)
31
32    ldmdb %r11,{%r4-%r11,%r13,%r15} # restore & return
```

```
 1    .text
 2    .code 16
 3    .globl arm32_thumb
 4  arm32_thumb:
 5    push {%r4-%r7, %r14} # save registers
 6    mov  %r7, %r13       # fp = sp
 7
 8    mov  %r4, %r0
 9    mov  %r5, %r1
10    mov  %r6, %r2
11
12    cmp  %r6, #16
13    ble  .call           # use stack?
14    sub  %r6, #16
15    mov  %r0, %r13
16    sub  %r0, %r0, %r6
17    lsr  %r0, #3
18    lsl  %r0, #3
19    mov  %r13, %r0
20    add  %r1, #16
21    mov  %r2, #0
22  .copy:
23    ldrb %r3, [%r1, %r2]
24    strb %r3, [%r0, %r2]
25    add  %r2, %r2, #1
26    cmp  %r2, %r6
27    bne .copy
28  .call:
29    ldmia        %r5!, {%r0-%r3}
30    blx %r4       # armv5/v6 branch-and-link
31    mov %r13, %r7        # sp = fpr
32    pop {%r4-%r7, %r15} # restore registers
```

Listing 24: Call Kernel for `arm-aapcs-softfloat` calling conventions implemented for `arm` 32-bit (*left*) and `thumb` 16-bit (*right*) ISA.

a *branch-and-link* call is executed to the `target` function (line 29 to 30). On return the preserved registers are restored and the return address is moved to the program counter (15) to return to the caller (line 31).

The `thumb` ISA code is quite similar except that R7 is used as frame pointer to be compilant with the specifications of iOS for `armv6` and `armv7` platforms, and alternative sequences of instructions are used to ensure 8-byte stack frame alignment and copying.

In order to support `arm`/`thumb` interworking it should be noted in particular which *branch-and-link* instruction to choose for indirect function calls (to target addresses stored in registers). The branch-and-link instruction `blx Rm` can be used in `armv5`/`armv6` for making sub-routine function calls with support for interworking. However, in order to support `armv4t` the sequence `mov lr,pc` and `bx Rm` need to be used. See ARM Limited (2009, pp.21) for details. Which sequence to choose depends on the target platform; we currently incorporate a compile-time selection in the `dyncall arm` assembly sources that selects an appropriate method based on the requested target platform.

Both call kernels are used for two different calling conventions, namely `apcs-gnu` and `aapcs`, implemented as separate back-end drivers given in Listing 25. Their rules differ for the alignment of data types larger than 4 bytes i.e. for passing `long long` and `double` C argument data types. Note that the implementation is also duplicated in order to support `thumb` target platforms where a different call delegate, `call_thumb`, is used that calls `arm32_thumb` instead of `arm32_arm`.

```
void aInt(DCCallVM* p, DCint v) {
  dcVecAppend(&((DCCallVM_arm32*)p)->mVecHead,
    &v, sizeof(DCint)); }
void aBool (DCCallVM* p, DCbool  v) { aInt(p,v); }
void aChar (DCCallVM* p, DCchar  v) { aInt(p,v); }
void aShort(DCCallVM* p, DCshort v) { aInt(p,v); }
void aLong (DCCallVM* p, DClong  v) {
  dcVecAppend(&((DCCallVM_arm32*)p)->mVecHead,
    &v, sizeof(DClong));}
void aLongLong(DCCallVM* p, DClonglong v) {
  dcVecAppend(&((DCCallVM_arm32*)p)->mVecHead,
    &v, sizeof(DClonglong));}
void aFloat(DCCallVM* p, DCfloat    v) {
  dcVecAppend(&((DCCallVM_arm32*)p)->mVecHead,
    &v, sizeof(DCfloat   ));}
void aDouble(DCCallVM* p, DCdouble   v) {
  dcVecAppend(&((DCCallVM_arm32*)p)->mVecHead,
    &v, sizeof(DCdouble  ));}
void aPointer(DCCallVM* p, DCpointer  v) {
  dcVecAppend(&((DCCallVM_arm32*)p)->mVecHead,
    &v, sizeof(DCpointer ));}
DCCallVM_vt gVT_apcs_gnu = {
 &destroy,&reset,&mode,
,&aBool,&aChar,&aShort,&aInt,&aLong
,&aLongLong
,&aFloat
,&aDouble
,&aPointer
,(DCvoidvmfunc*) &call_arm
,(DCboolvmfunc*) &call_arm
 /* [ ... lines omitted ... ] */
,(DCpointervmfunc*)&call_arm
};
```

```
void aDoubleAlign8(DCCallVM* p, DCdouble v) {
  /* align vector to 8 byte boundary. */
  dcVecSkip(&((DCCallVM_arm32*)p)->mVecHead,
    dcVecSize(&((DCCalVM_arm32*)p)) & 4 );
  dcVecAppend(&((DCCallVM_arm32*)p)->mVecHead,
    &v, sizeof(DCdouble));
}

void aLongLongAlign8(DCCallVM* p, DClonglong v) {
  /* align vector to 8 byte boundary. */
  dcVecSkip(&((DCCallVM_arm32*)p)->mVecHead,
    dcVecSize(&((DCCalVM_arm32*)p)) & 4 );
  dcVecAppend(&((DCCallVM_arm32*)p)->mVecHead,
    &v, sizeof(DCdouble));
}

void call_arm(DCCallVM* p, DCpointer target) {
  DCCallVM_arm32_arm* vm = (DCCallVM_arm32_arm*)p;
  arm32_arm(target, dcVecData(&vm->mVecHead),
    dcVecSize(&vm->mVecHead));
}
DCCallVM_vt gVT_aapcs_linux = {
 &destroy,&reset,&mode
,&aBool,&aChar,&aShort,&aInt,&aLong
,&aLongLongAlign8  /* <-- 8-byte alignment version */
,&aFloat
,&aDoubleAlign8    /* <-- 8-byte alignment version */
,&aPointer
,(DCvoidvmfunc*) &call_arm
,(DCboolvmfunc*) &call_arm
 /* [ ... lines omitted ... ] */
,(DCpointervmfunc*) &call_arm
};
```

Listing 25: Call VM for `apcs-gnu` and `aapcs-linux` ABIs on ARM for `arm` ISA

5.4.10 Passing leading integer arguments via registers (`x86-32-fast` ports)

The `fastcall` calling conventions on `x86-16` and `x86-32` use two or more integer registers for passing *integer-only* data types. `dyncall` contains two back-end drivers for compiler-specific variants of `fastcall` ABIs for `x86-32`, namely for Microsoft Visual C (`fastcall-ms`) and GNU (`fastcall-gnu`) compilers. Both calling conventions reserve the two registers ECX and EDX for passing integers up to 32 bit. 64-bit integer and floating-point values are passed via the stack. However, in contrast to `cdecl` and `stdcall` a second buffer is needed and the Call VM state machine needs to track the number of available registers. For this purpose the first 8 bytes of the internal vector are reserved for data to be passed via integer registers and a counter variable `int mIntRegs` was added to the Call VM to check the current state depicted in Listing 20. One call kernel is used by both `fastcall` calling conventions. It always loads the first 8 bytes of the buffer to ECX and EDX and after that corresponds with the implementation of the `stdcall` call kernel. The C interface is as follows:

```
void x86_fastcall ( void (*target)(), void* buffer, size_t size);
```

The code, given in Listing 26, is similar to the `stdcall` call kernel (Listing 23) since the callee is responsible for cleaning the stack. In addition, the first 8 bytes of the buffer are copied to the integer registers ECX and EDX (line 10-13 and 18).

```
 1    .text
 2    .globl x86_fastcall
 3  x86_fastcall:
 4    pushl %ebp            # prolog
 5    movl  %esp,%ebp
 6    pushl %esi            # save preserved
 7    pushl %edi
 8    movl 12(%ebp),%esi    # ESI = buf
 9    movl 16(%ebp),%ecx    # ECX = len
10    movl  0(%esi),%eax    # EAX = register argument 1
11    movl  4(%esi),%edx    # EDX = register argument 2
12    addl $8,%esi          # skip first 8 bytes
13    subl $8,%ecx          # for buf and len
14    subl %ecx,%esp        # alloc stack frame
15    movl %esp,%edi        # EDI = top
16    rep movsb             # copy arguments
17    movl %eax,%ecx        # ECX = register argument 1
18    call *8(%ebp)         # call
19    popl %edi             # restore preserved
20    popl %esi
21    movl %ebp,%esp        # epilog
22    popl %ebp
23    ret
```

Listing 26: Call kernel for `fastcall` on `x86-32`.

The drivers `fastcall-gnu` and `fastcall-ms` are integrated with those in the Call VM for `x86-32` and their implementation is given in Listing 27.

Several driver methods of the `fastcall` calling conventions differ from that of `cdecl` and `stdcall`. The reset method `fReset` reserves the first 8 bytes of the vector for use as a separate buffer for the

```
void fInt(DCCallVM* p, DCint v) {
  DCCallVM_x86* x = (DCCallVM_x86*) p;
  if (x->mIntRegs < 2) { /* to register */
    *((int*)dcVecAt(&x->mVecHead,x->mIntRegs*4))=v;
    x->mIntRegs++;
  } else                 /* to stack */
    dcVecAppend(&x->mVecHead,&v,sizeof(DCint));
}
/* Promote small integers to 32 bit. */
void fBool (DCCallVM* p,DCbool  x){fInt(p,(DCint)x);}
void fChar (DCCallVM* p,DCchar  x){fInt(p,(DCint)x);}
void fShort(DCCallVM* p,DCshort x){fInt(p,(DCint)x);}
void fLong (DCCallVM* p,DClong  x){fInt(p,(DCint)x);}
void fPtr(DCCallVM* p,DCpointer x){fInt(p,(DCint)x);}
DCCallVM_vt gVT_x86_fast_ms = {
  &destroy, &fReset, &mode
, &fBool, &fChar, &fShort, &fInt, &fLong
, &aLongLong                    /* <-- different --> */
, &aFloat, &aDouble, &fPtr
, (DCvoidvmfunc     *) &fCall
, (DCboolvmfunc     *) &fCall
/* [ ... lines omitted ... ] */
, (DCpointervmfunc  *) &fCall
};
```

```
void fLongLong_gnu(DCCallVM* p, DClonglong v) {
  self->mIntRegs = 2; /* always skip registers */
  aLongLong(p, v);
}
void fReset(DCCallVM* p) {
  DCCallVM_x86* x = (DCCallVM_x86*) p;
  dcVecResize(&x->mVecHead,8);
  x->mIntRegs=0;
}
void fCall(DCCallVM* p, DCpointer target) {
  DCCallVM_x86* x = (DCCallVM_x86*) p;
  x86_fastcall(target,dcVecData(&x->mVecHead),
           dcVecSize(&x->mVecHead));
}
DCCallVM_vt gVT_x86_fast_gnu = {
  &destroy, &fReset, &mode,
  &fBool, &fChar, &fShort, &fInt, &fLong
, &fLongLong_gnu         /* <-- GNU's fastcall */
, &aFloat, &aDouble, &fPtr,
, (DCvoidvmfunc     *) &fCall
, (DCboolvmfunc     *) &fCall
/* [ ... lines omitted ... ] */
, (DCpointervmfunc  *) &fCall
};
```

Listing 27: Call VM for `fastcall` on `x86-32`.

two integer registers. The method `fInt` implements the core logical to pass integer 32-bit data first via registers where data is buffered in the first 8 bytes of the vector and the state of currently occupied registers is tracked by the `mIntRegs` counter variable. If no register is free data is appended to the vector. For all integer-based data types smaller 32 bit, the data is promoted to 4 bytes. Floating-point values are always appended to vector. 64 bit integer data types are passed via the stack and GNU's variant marks both registers as used.

5.4.11 Passing arguments via registers with homing area (`sparc-v7` port)

Six 32-bit general-purpose registers are reserved for parameters by the `sparc v7` (and `v8`) ABI. Although the 32-bit SPARC calling conventions use large structures for stack frames and specific mechanisms for their allocation and destruction they are relatively simple. All arguments are promoted to 32-bit values and then appended to the stack. However, the first 24 bytes of memory for parameters are reserved as the "homing area" and these data are passed via the corresponding memory of six parameter registers. Due to this static mapping between stack memory and registers the implementation of the Call VM consists only of the vector, as in the case of `x86-32` and `arm-apcs`:

```
typedef struct {
  DCCallVM  mInterface;
  DCVecHead mVecHead;
} DCCallVM_sparc;
```

The implementation of the Call VM, given in Listing 28, is also very similar to that of `x86-32` and `arm-apcs`.

```
void reset    (DCCallVM* p)            {dcVecReset (&((DCCallVM_sparc*)p)->mVecHead);                         }
void aInt     (DCCallVM* p,DCint      v){dcVecAppend(&((DCCallVM_sparc*)p)->mVecHead, &v, sizeof(DCint     ));}
void aBool    (DCCallVM* p,DCbool     v){aInt(p,(DCint)v);                                                    }
void aChar    (DCCallVM* p,DCchar     v){aInt(p,(DCint)v);                                                    }
void aShort   (DCCallVM* p,DCshort    v){aInt(p,(DCint)v);                                                    }
void aLong    (DCCallVM* p,DClong     v){dcVecAppend(&((DCCallVM_sparc*)p)->mVecHead, &x, sizeof(DClong    ));}
void aLongLong(DCCallVM* p,DClonglong v){dcVecAppend(&((DCCallVM_sparc*)p)->mVecHead, &x, sizeof(DClonglong));}
void aPointer (DCCallVM* p,DCpointer  v){dcVecAppend(&((DCCallVM_sparc*)p)->mVecHead, &v, sizeof(DCpointer ));}
void aFloat   (DCCallVM* p,DCfloat    v){dcVecAppend(&((DCCallVM_sparc*)p)->mVecHead, &v, sizeof(DCfloat   ));}
void aDouble  (DCCallVM* p,DCdouble   v){dcVecAppend(&((DCCallVM_sparc*)p)->mVecHead, &v, sizeof(DCdouble  ));}
DCCallVM_vt gVT_v7 = {
  &free, &reset, &mode, &aBool, &aChar, &aShort, &aInt, &aLong, &aLongLong, &aFloat, &aDouble, &aPointer,
  (DCvoidvmfunc*)     &sparc_v7,  (DCboolvmfunc*)     &sparc_v7,  (DCcharvmfunc*)     &sparc_v7,
  (DCshortvmfunc*)    &sparc_v7,  (DCintvmfunc*)      &sparc_v7,  (DClongvmfunc*)     &sparc_v7,
  (DClonglongvmfunc*) &sparc_v7,  (DCfloatvmfunc*)    &sparc_v7,  (DCdoublevmfunc*)   &sparc_v7,
  (DCpointervmfunc*)  &sparc_v7
};
```

Listing 28: Call VM for `sparc-v7`.

However, the call kernel is directly plugged in for all *call* methods (the `&sparc_v7` fields in the function table `gVT_v7`); implementations other than `sparc` use a delegate `call` function here to extract parameters, such as `buffer` pointer and the `size` of the vector, to be passed to the kernel. In this case the C function type interface of call kernels for `sparc` corresponds to that of the front-end `dcCall<RTYPE>` API interface functions:

```
void sparc_v7 (DCCallVM* vm, DCpointer target);
```

This direct assembly interface to the C API front-end was choosen due to SPARC's *register windows* and, in particular, the change of mapping return value registers during function returns. On most platforms a C function can be plugged as a call delegate for all *call* methods regardless of different return types; the return value in registers is passively forwarded by the `void` function. However, on `sparc` the return value could be destroyed.

```
sparc_v7:
  or   %o0, %g0, %o3    ! o3: vm
  or   %o1, %g0, %o0    ! o0: target
  ld   [%o3+12], %o1    ! o1: size
  add  %o3, 16, %o2     ! o2: data
  add  %o1, 92+15,%o3   ! o3: stack frame size:
  and  %o3, -16, %o3    !       92 = (16+1+6)*4
  neg  %o3              !       16-byte aligned
  save %sp, %o3, %sp    ! Allocate register window
  ld   [%i2    ],%o0    ! Load 6 x 32-bit GPRs
  ld   [%i2+4*1],%o1
  ld   [%i2+4*2],%o2
  ld   [%i2+4*3],%o3
  ld   [%i2+4*4],%o4
  ld   [%i2+4*5],%o5
















  sub   %i1, 24, %i1    ! Skip homing area
  cmp   %i1, 0          ! Size > 0 ?
  ble  .do_call
  nop                   ! Copy on stack:
  add  %i2, 24, %i2     ! i2: 7th word of args buffer
  or   %g0,%g0, %l0     ! l0: offset initialized to 0
  add  %sp, 92, %l2     ! l2: argument area on stack
.next:
  ld   [%i2+%l0],%l1    ! Read  4-byte word
  st   %l1, [%l2+%l0]   ! Write 4-byte word
  add  %l0, 4, %l0      ! Update index
  sub  %i1, 4, %i1      ! Update counter
  cmp  %i1, 0
  bgt  .next
  nop
.do_call:
  call %i0              ! Call target
  nop
  or   %o0,%g0,%i0      ! Rescue return value
  or   %o1,%g0,%i1      ! Rescue upper 32-bit
  jmpl %i7+8,%g0        ! Return, but before:
  restore               ! Restore register window
```

```
dcCall_v9:
  or   %o0, %g0   , %o3
  or   %o1, %g0   , %o0
  ldx [%o3+24]    , %o1  ! Offset 24
  add  %o3,    32, %o2   ! Offset 32
  add  %o1,176+15, %o3   ! Stack frame size:
  and  %o3,   -16, %o3   !    176 = (16+0+6)*8
  neg  %o3               !     16-byte alined
  save %sp, %o3, %sp
  ldx  [%i2     ] ,%o0   ! Load 6  x 64-bit GPRs
  ldx  [%i2+8*1] ,%o1
  ldx  [%i2+8*2] ,%o2
  ldx  [%i2+8*3] ,%o3
  ldx  [%i2+8*4] ,%o4
  ldx  [%i2+8*5] ,%o5
  ldd  [%i2     ],%f0    ! Load 16 x 64-bit FPRs
  ldd  [%i2+8*1 ],%f2
  ldd  [%i2+8*2 ],%f4
  ldd  [%i2+8*3 ],%f6
  ldd  [%i2+8*4 ],%f8
  ldd  [%i2+8*5 ],%f10
  ldd  [%i2+8*6 ],%f12
  ldd  [%i2+8*7 ],%f14
  ldd  [%i2+8*8 ],%f16
  ldd  [%i2+8*9 ],%f18
  ldd  [%i2+8*10],%f20
  ldd  [%i2+8*11],%f22
  ldd  [%i2+8*12],%f24
  ldd  [%i2+8*13],%f26
  ldd  [%i2+8*14],%f28
  ldd  [%i2+8*15],%f30
  sub  %i1, 48, %i1      ! Skip 48 bytes homing area
  cmp  %i1, 0
  ble  .do_call
  nop
  add  %i2, 48, %i2      ! Skip 6 x 64-bit
  or   %g0, %g0, %l0
  add  %sp 2047+176,%l2 ! Biased sp (-2047)
.next:
  ldx  [%i2+%l0],%l1     ! Read  8-byte word
  stx  %l1, [%l2+%l0]    ! Write 8-byte word
  add  %l0, 8, %l0
  sub  %i1, 8, %i1
  cmp  %i1, 0
  bgt  .next
  nop
.do_call:
  call %i0
  nop
  or   %o0,%g0,%i0       ! Rescue 64-bit value

  jmpl %i7+8,%g0
  restore
```

Listing 29: Call kernels for `sparc` ABIs `v7/v8` (*left*) and `v9` (*right*).

The implementation of the call kernel is depicted in Listing 29 (*left*). Due to the different C function interface of the assembly routine the parameters `size` and `buffer` need to be retrieved directly from the `vm` base parameter (line 2-5). Before a new stack frame is allocated the size needs to be calculated. The minimum size for a standard stack frame header is 16 words for the register window, 1 word for "large return value" pointer and 6 words for the homing area i.e. 23×4 bytes = 92 bytes (see column "`sparc`" in Figure 5.12 for an illustration). The `size` of the buffer is added and the total size[4] is aligned to a 16-byte boundary.

The stack frame is allocated by the `save` instruction by passing the requested size as a parameter (line 9). As a result the mapping of registers changes. Output registers O0-O7 are mapped to input registers I0-I7 and a free set of local and output registers is made available by the system. The first six words of the `buffer` are loaded to output registers O0-O5 (line 10-15), `size` is reduced by 24 bytes (line 32) to determine the remaining number of bytes, if any (line 33-34), to be copied to the parameter region of the stack frame (starting above the stack header at byte offset 92) (line 36-45). Finally the call to the target address is executed (line 48).

On return of a function integer-based results are passed in register O0 and O1. However, the local stack frame needs to be cleaned up via a `restore` instruction (line 52) which remaps the input registers back to the output register (for the caller) so that results would be destroyed. Therefore registers O0 and O1 need to be saved to the input registers (line 50-51) first. Note the `nop` (no operation) instructions (line 19, 35, and 46); these are *branch delay slots* which get executed as part of the previous instruction. In the case of the function's return (line 52) the `restore` instruction is part of its branch delay slot (line 53) and gets executed *before* the function has returned.

5.4.12 Overlapping FPU registers and homing area (`sparc-v9` port)

The 64-bit ABI for `sparc`, named `v9`, also uses six general-purpose registers and a corresponding homing area as does the 32-bit predecessors `v7/v8`. In addition, registers of the FPU for passing arguments are incorporated in the calling convention. Several floating-point formats are supported by distinct FPU registers but with overlapping register memory; a 64-bit register is shared by a pair of 32-bit registers[5]. The first sixteen 64-bit registers are reserved for passing `double` values. From the corresponding set of thirty-two 32-bit registers the sixteen odd-numbered registers are reserved for passing `float` values. The calling convention puts parameter registers and the stack in relation to each other. When a parameter is passed via a FPU register, the next free location on the stack is marked as occupied for subsequent arguments in the calling sequence. Conversely, parameters passed in general-purpose registers block the next free location in 32/64-bit FPU registers.

Several versions of a port to `v9` were implemented in `dyncall`; the first version incorporates a state

[4]The total size is slightly larger than needed as the first 24 bytes are already included in the standard header size, a trade-off decision between small size overhead versus more calculations for saving 24 bytes stack space reservation.

[5]Only the first half of the 64-bit registers use this mapping; see Figure 5.11 for details.

machine design based on a bit mask for the piecewise overloading of 32-bit floating-point values, distinct buffers for each register type and counters and the vector. More recently a new version was implemented based on the simple mapping scheme described below.

The calling convention prescribes an alignment of stack values at 8-byte boundaries and has a "homing area" for the first six general-purpose registers. The inter-relation between registers and the stack suggests that one regards the 64-bit FPU registers as being also "homed". Since a "homing" location on the stack is either associated with a register of one kind or the other but not both; the same head of the stack can be transferred to both sets (for a relatively small kernel in assembly). Furthermore, if the sixteen 64-bit FPU registers for passing `double` values are stored to memory in sequence, all `float` values in 32-bit FPU registers are covered by this operation and the 4-byte gap, due to unused even registers, ensures 8-byte alignment.

The current implementation uses a similarly simple Call VM as in `v7` but with 64-bit value promotion/padding for 8-byte alignment, as depicted in Listing 30. Function `argFloat` implements buffering of `float` by passing the value in a 8-byte `union` that matches the overlapping scheme of the floating-point registers before being append to the vector.

```
void argLongLong(DCCallVM* p, DClonglong x) {
  DCCallVM_v9* self = (DCCallVM_v9*)p;
  dcVecAppend(&self->mVecHead, &x, sizeof(DClonglong));
}
void argLong   (DCCallVM* p, DClong    x) { argLongLong(p, (DClonglong) x ); }
void argInt    (DCCallVM* p, DCint     x) { argLongLong(p, (DClonglong) x ); }
void argBool   (DCCallVM* p, DCbool    x) { argLongLong(p, (DClonglong) x ); }
void argChar   (DCCallVM* p, DCchar    x) { argLongLong(p, (DClonglong) x ); }
void argShort  (DCCallVM* p, DCshort   x) { argLongLong(p, (DClonglong) x ); }
void argPointer(DCCallVM* p, DCpointer x) { argLongLong(p, (DClonglong) x ); }
void argDouble (DCCallVM* p, DCdouble x)  {
  DCCallVM_v9* self = (DCCallVM_v9*)p;
  dcVecAppend(&self->mVecHead, &x, sizeof(DCdouble));
}
void argFloat  (DCCallVM* p, DCfloat x) {
  union { DClonglong l, float  f[2]; } u;
  u.f[1] = x;
  argLongLong(p, u.l);
}
```

Listing 30: Call VM for `sparc-v9`.

The call kernel loads the first six 64-bit words to general-purpose registers, but also the first sixteen 64-bit words to corresponding FPU registers. In accordance to `v7`, the C interface of the call kernel is similar to the front-end API:

```
void sparc_v9 (DCCallVM* vm, DCpointer target);
```

Listing 29 (*right* panel) gives the implementation for the `v9` call kernel. The differences to the `v7` kernel (*left* panel) are as follows:

- A sequence of `ldx` instructions is used (instead of `ld` instructions for loading the first 48 bytes of buffer data to six 64-bit general-purpose registers (line 10-15).

- A sequence of `ldd` instructions is added for loading the first 128 bytes of buffer data to the sixteen 64-bit floating-point registers for passing `float` and `double` values (line 16-31).
- The size of the standard header is not exactly a multiple of 2 (due to 64-bit) but adjusted by one element on the stack which is used for the computation of a stack frame size of 176 bytes (line 6). This is due to a slightly different stack frame structure of `v9`: The *valref* field, reserved for passing a reference to memory objects for large return values in `v7`, was removed from the standard structure of `v9` (compare columns '`v7/v8`' and '`v9`' in Figure 5.12).
- By convention, the `v9` stack pointer is always biased by -2047 bytes and needs to be adjusted to get the effective address (line 38).
- One output register was copied to one input register for saving the 64-bit integer return value (line 50).

5.4.13 Passing floating-point arguments via multiform FPU registers (`x86-64` ports)

ABIs on the `x86-64` utilize general-purpose and SIMD vector registers for passing arguments. The latter are used for passing 32-bit and 64-bit floating-point values. The two major ABIs, namely for Microsoft Windows 64-bit (`x64`) and for System V Unix-based systems (`sysv`), use different calling conventions. While six general-purpose and eight floating-point registers for 64-bit values are reserved for arguments on `sysv`, only four of each kind are used in `x64`.

In addition, `x64` also incorporates a homing area on the stack with the capacity for 4× 64-bit register storage = 32 bytes of register data. This byte quantity corresponds to the capacity of register-memory for passing arguments of one kind but not for both. On `x64` the four SIMD registers are logically mapped to the four general-purpose registers so that an argument value assigned to one register marks a corresponding register of the other kind as occupied.

`sysv` uses a simple calling convention. Depending on the data type kind a corresponding register file is subsequently filled with argument data. If no space is left the value is assigned to stack memory. Furthermore, the two ABIs use a different 64-bit C programming model; the size of a `long` is mapped to 32-bit on `x64` and 64-bit on `sysv`). However the alignment and promotion rules for smaller argument data types less than or equal to 64-bit are effectively the same. In summary, four arguments can be passed via registers on `x64` in contrast to a maximum number of fourteen arguments via registers on `sysv`.

A common code base was used for the implementation of the methods, while the different calling conventions are manifested in specific call kernels and configurations for the data structure by using two buffers and two counter variables for `sysv` and a unified buffer and counter for `x64` as depicted in Listing 31. The configuration is based on preprocessor macros, `DC_WINDOWS` (`x64`) and `DC_UNIX` (`sysv`), that indicate the target ABI platform.

```
#if defined(DC_WINDOWS)
typedef long long int64;        /* llp64 */
#define numIntRegs   4
#define numFloatRegs 4
#define DCRegCount_x64 DCRegCount_x64_u
#define DCRegData_x64  DCRegData_x64_u

#elif defined(DC_UNIX)
typedef long      int64;        /* lp64 */
#define numIntRegs   6
#define numFloatRegs 8
#define DCRegCount_x64 DCRegCount_x64_s
#define DCRegData_x64  DCRegData_x64_s
#endif
```

```
typedef union  { int i; int f; } DCRegCount_x64_u;
typedef struct { int i; int f; } DCRegCount_x64_s;
typedef union  {
  int64 i[numIntRegs]; double f[numFloatRegs];
} DCRegData_x64_u;
typedef struct {
  int64 i[numIntRegs]; double f[numFloatRegs];
} DCRegData_x64_s;
typedef struct {
  DCCallVM       mInterface;
  DCRegCount_x64 mRegCount;
  DCRegData_x64  mRegData;
  DCVecHead      mVecHead;
} DCCallVM_x64;
```

Listing 31: Call VM data structure for `x86-64` calling conventions.

The shared implementation of both Call VM is given in Listing 32.

However, the call method is slightly different as the interface of each call kernel differs as follows:

```
void x86_64_sysv(size_t size,void* buf,void* idata,void* fdata,(void (*target)()));
void a_B(DCCallVM* p, DCbool v){ a_l(p,(long long)v) };
void a_c(DCCallVM* p, char   v){ a_l(p,(long long)v) };
void a_s(DCCallVM* p, short  v){ a_l(p,(long long)v) };
void a_i(DCCallVM* p, int    v){ a_l(p,(long long)v) };
void a_j(DCCallVM* p, int    v){ a_l(p,(long long)v) };
void a_p(DCCallVM* p, void*  v){ a_l(p,(long long)v) };
void a_l(DCCallVM* p, long long v) {
  DCCallVM_x64* x = (DCCallVM_x64*)p;
  if(x->mRegCount.i < numIntRegs)
    x->mRegData.i[x->mRegCount.i++] = v;
  else
    dcVecAppend(&x->mVecHead, &v, 8);
}
void a_f(DCCallVM* p, float  v){
  DCCallVM_x64* x  = (DCCallVM_x64*)p;
  union { /* floats are passed as 64-bit words */
    DCdouble d;
    DCfloat  f;
  } f;
  f.f = v;
  if(x->mRegCount.f < numFloatRegs)
    *(DCfloat*)&x->mRegData.f[x->mRegCount.f++] = v;
  else
    dcVecAppend(&x->mVecHead, &f.f, sizeof(DCdouble));
}
void a_d(DCCallVM* p, double v)
{
  DCCallVM_x64* x = (DCCallVM_x64*)p;
  if(x->mRegCount.f < numFloatRegs)
    x->mRegData.f[x->mRegCount.f++] = v;
  else
    dcVecAppend(&x->mVecHead, &v, sizeof(DCdouble));
}
```

```
void reset(DCCallVM* p) {
  DCCallVM_x64* x = (DCCallVM_x64*)p;
  dcVecReset(&x->mVecHead);
  x->mRegCount.i = x->mRegCount.f =  0;
}
void call(DCCallVM* p, DCpointer target)
{
  DCCallVM_x64* x = (DCCallVM_x64*)p;
#if defined(DC_UNIX)
  x86_64_sysv(
#else
  x86_64_win64(
#endif
    dcVecSize(&x->mVecHead), dcVecData(&x->mVecHead),
    x->mRegData.i,
#if defined(DC_UNIX)
    x->mRegData.f,
#endif
    target
  );
}

DCCallVM_vt gVT_x86_64 =
{
  &free, &reset, &mode
, &a_B, &a_c, &a_s, &a_i, &a_j, &a_l
, &a_f, &a_d, &a_p
, (DCvoidvmfunc*)     &call, (DCboolvmfunc*)     &call
, (DCcharvmfunc*)     &call, (DCshortvmfunc*)    &call
, (DCintvmfunc*)      &call, (DClongvmfunc*)     &call
, (DClonglongvmfunc*) &call, (DCfloatvmfunc*)    &call
, (DCdoublevmfunc*)   &call, (DCpointervmfunc*)  &call
};
```

Listing 32: Call VM for `x86-64` calling conventions.

```
void x86_64_x64 (size_t size,void* buf,void* regdata ,(void (*target)()));
```

Listing 33 gives the assembly source for both call kernels. We discuss the `sysv` first. Preserved registers are saved to the stack (line 2-4) and the stack pointer is saved to the frame pointer (line 5). Floating-point register data (`fdata`) are loaded to XMM0-XMM7 (line 7-14). A new stack frame is allocated (line 15) and aligned to a lower 32-byte boundary (line 16). Similar to `x86-32` data are copied using a string move operation with a loop prefix (line 17-19). Integer data are loaded to the six integer registers (line 20-25). In order to support calls to variadic functions the AL register is set to 8 (line 26) to indicate the maximum number of XMM registers that are potentially filled with data as specified in Matz et al. (2012, p.20). Then the call is performed (line 27). Finally the stack frame is removed (line 28) and preserved registers are restored (line 29-30) before returning to the caller (line 31).

The `x64` calling convention is significantly different. The code begins with a standard prolog code for saving preserved registers (line 2-5). The size of the buffer is aligned to the next 16-byte boundary (line 6-7). Then a new stack frame is allocated (line 8) followed by a standard string copy (line 9-11). R9 (`target` argument) is saved to RAX (line 12) to make all registers available for loading 32 bytes of `regdata` to integer registers (line 13-16). The same data are transferred as-is (without conversion or type cast) to corresponding floating-point registers (line 17-20). The homing area is allocated at

```
 1  x86_64_sysv:
 2    pushq %rbp          # prolog
 3    pushq %rbx
 4
 5    movq  %rsp,%rbp
 6    movq  %r8, %rbx     # RBX = fun
 7    movsd  0(%rcx),%xmm0 # load float regs
 8    movsd  8(%rcx),%xmm1
 9    movsd 16(%rcx),%xmm2
10    movsd 24(%rcx),%xmm3
11    movsd 32(%rcx),%xmm4
12    movsd 40(%rcx),%xmm5
13    movsd 48(%rcx),%xmm6
14    movsd 56(%rcx),%xmm7
15    sub   %rdi, %rsp    # create stack frame
16    and   %rsp, #-32    # align to 32 byte
17    mov   %rdi, %rcx    # set loop count
18    mov   %rsp, %rdi    #
19    rep movsb           # copy on stack
20    movq  0(%rdx),%rdi  # load int regs
21    movq  8(%rdx),%rsi
22    movq 24(%rdx),%rcx
23    movq 32(%rdx),%r8
24    movq 40(%rdx),%r9
25    movq 16(%rdx),%rdx
26    movb $8,%al         # AL = 8 XMM regs used
27    call *%rbx          # call
28    movq %rbp,%rsp      # epilog
29    popq %rbx
30    popq %rbp
31
32    ret
```

```
 1  x86_64_x64:
 2    pushq %rbp         # Prolog
 3    pushq %rsi
 4    pushq %rdi
 5    movq %rsp,%rbp
 6    addq $ 15,%rcx     # RCX = align(RCX,16)
 7    andq $-16,%rcx
 8    subq %rcx,%rsp
 9    movq %rdx,%rsi     # RSI = sdata
10    movq %rsp,%rdi     # RDI = stack frame
11    rep movsb          # copy data
12    movq  %r9,%rax     # RAX = fun
13    movq 0 (%r8),%rcx # load int regs
14    movq 8 (%r8),%rdx
15    movq 24(%r8),%r9
16    movq 16(%r8),%r8
17    movd %rcx,%xmm0    # load float regs
18    movd %rdx,%xmm1
19    movd %r8 ,%xmm2
20    movd %r9 ,%xmm3
21    pushq %r9          # push first arguments
22    pushq %r8          #
23    pushq %rdx         #
24    pushq %rcx         #
25
26
27    call *%rax         # call
28    movq %rbp,%rsp     # epilog
29    popq %rdi
30    popq %rsi
31    popq %rbp
32    ret
```

Listing 33: Call kernels for `x86-64` ABIs: System V `sysv` (*left*) and Microsoft Windows `x64` (*right*).

the top of stack, but with content by push instructions to copy `regdata` also to stack memory (line 21-24) in order to support varadic function calls. Finally the call is performed (line 27), followed by a standard epilog (line 28-32).

5.4.14 Passing floating-point arguments via uniform FPU registers (ppc32 ports)

PowerPC offers two sets[6] of register files for passing arguments. Specific to this architecture the FPU uses a uniform 64-bit floating-point register format so that `float` 32-bit values need to be passed converted as 64-bit `double` values when being passed via FPU registers.

Two ABI ports for the PowerPC 32-bit architecture, namely System V (sysv) and Mac OS X (osx), are currently available in dyncall. Both ABIs use a stack frame header with a slightly different structure as illustrated in column "ppc" of Figure 5.12.

On sysv arguments are assigned to be passed via registers of one of register file depending on the data type. Eight registers are reserved for both register classes. If all parameter registers of one file are exhausted the argument is assigned to be passed via the parameter region of the stack frame. Subsequent arguments in the sequence can still be assigned to the other file if possible. Specific to this calling convention 64-bit integer values are passed via odd-even register pairs (in the range of R3 and R10) and this can lead to gaps of unused even-numbered registers. Aside from that, the sysv utilizes a relatively simple scheme in contrast to osx.

On osx the same range of integer-based parameter registers are utilized but the range of floating-point registers is extended by five registers to thirteen in total for passing floating-point arguments. Furthermore, stack space is reserved for a homing area. The combination of a homing area for eight 32-bit integer registers, which are partly linked with thirteen 64-bit floating-point registers, effectively leads to a more complex calling convention rule set.

One Call VM was implemented for ppc32 that integrates both calling conventions and supports switching modes similar to the design of the x86-32 port. Thus the data structure contains buffers for eight 32-bit integer registers and thirteen 64-bit floating-point registers.

```c
typedef struct {
  DCint          gpr[8];
  DCdouble       fpr[13];
} DCRegData_ppc32;
typedef struct {
  DCCallVM        mInterface;
  int             iregs;
  int             fregs;
  DCRegData_ppc32 mRegData;
  DCVecHead       mVecHead;
} DCCallVM_ppc32;
```

The implementation is given in Listing 34. Note that the two function tables (at the bottom of the listing) are initialized with methods, some of which are shared by both drivers while others are specific to either sysv or osx and prefixed 's' or 'o' in the C code, respectively.

[6]This port considers a standard PowerPC platform which offers a FPU. Note that variants exist, such as embedded PowerPC platforms without FPU, and Apple's ABI supports a SIMD extension, named altivec, which offers a third register file for passing intrinsic vector data types.

```
/* -------------------------------------------- Common methods -------------------------------------------- */
void reset(DCCallVM* p) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p; dcVecReset(&x->mVecHead); x->iregs=0; x->fregs=0; }
                                  /* -- methods below promote to int (either delegated to sysv_Int or osx_Int) */
void aBool    (DCCallVM* p, DCbool    v) { dcArgInt(p, (v == 0) ? DC_FALSE : DC_TRUE );}
void aChar    (DCCallVM* p, DCchar    v) { dcArgInt(p, (DCint) v ); }
void aShort   (DCCallVM* p, DCshort   v) { dcArgInt(p, (DCint) v ); }
void aLong    (DCCallVM* p, DClong    v) { dcArgInt(p, (DCint) v ); }
void aPointer (DCCallVM* p, DCpointer v) { dcArgInt(p, *(DCint*) &v ) };
```

```
/* --- System V methods: --------------------------- */
void sysv_Call(DCCallVM* p, DCpointer target) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*) p;
  ppc32_sysv(t, &x->mRegData,
    dcVecSize(&x->mVecHead),
    dcVecData(&x->mVecHead));
}
void sysv_Int(DCCallVM* p, DCint i) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p;
  if (x->iregs < 8) /* assign to register ? */
    x->mRegData.gpr[x->iregs++] = i;
  else /* OR: to stack      */
    dcVecAppend(&x->mVecHead,&i,sizeof(DCint));
}
void sysv_Float(DCCallVM* p, DCfloat f) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p;
  if (x->fregs < 8) /* as 'double' to register ? */
    x->mRegData.fpr[x->fregs++] = (DCdouble) (f);
  else /* OR to stack as-is */
    dcVecAppend(&x->mVecHead,&f,sizeof(DCfloat));
}


void sysv_Double(DCCallVM* p, DCdouble d) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p;
  if (x->fregs < 8) /* assign to FPR */
    x->mRegData.fpr[x->fregs++] = d;
  else { /* OR: to stack 8-byte aligned */
    dcVecResize(&x->mVecHead,
      (dcVecSize(&x->mVecHead)+7UL)&-8UL);
    dcVecAppend(&x->mVecHead,&d,sizeof(DCdouble));
  }
}
void sysv_LongLong(DCCallVM* p, DClonglong v) {
  DCint* d = (DCint*) &v;
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p;
  /* fillup integer register file */
  if (x->iregs < 7) { /* 2 GPRs available ? */
    if (x->iregs & 1) x->iregs++;
    x->mRegData.gpr[x->iregs++] = d[0];
    x->mRegData.gpr[x->iregs++] = d[1];
  } else {      /* OR: to stack 8-byte aligned */
    x->iregs=8; /* force all GPRs occupied. */
    dcVecResize(&x->mVecHead,
      (dcVecSize(&x->mVecHead)+7)&(-8UL));
    dcVecAppend(&x->mVecHead,&L,sizeof(DClonglong));
  }
}
DCCallVM_vt gVT_sysv = {
  &free, &reset, &mode
, &aBool, &aChar, &aShort
, &sysv_Int,
, &aLong,
, &sysv_LongLong,&sysv_Float,&sysv_Double,
, &aPointer
,(DCvoidvmfunc*)    &sysv_Call
,(DCboolvmfunc*)    &sysv_Call
/* [ ... lines omitted ... ] */
,(DCpointervmfunc*) &sysv_Call
};
```

```
/* --- Mac OS X methods: --------------------------- */
void osx_Call(DCCallVM* p, DCpointer t) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p;
  ppc32_osx(t,&x->mRegData,
    DC_MAX(dcVecSize(&x->mVecHead), 8*4),
    dcVecData(&x->mVecHead));
}
void osx_Int(DCCallVM* p, DCint i) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p;
  if (x->iregs < 8) /* to GPR ? */
    x->mRegData.gpr[x->iregs++] = i;
  /* AND: push onto stack (for ellipsis) */
  dcVecAppend(&x->mVecHead,&i,sizeof(DCint));
}
void osx_Float(DCCallVM* p, DCfloat f) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p;
  if (x->fregs < 13) /* to register? */
    x->mRegData.fpr[x->fregs++] = (DCdouble) (f);
  if (x->iregs < 8) /* skip 1 GPR */
    x->mRegData.gpr[x->iregs++] = *( (DCint*) &f );
  /* AND: push on stack */
  dcVecAppend(&x->mVecHead, &f, sizeof(DCfloat));
}
void osx_Double(DCCallVM* p, DCdouble d) {
  DCCallVM_ppc32* x = (DCCallVM_ppc32*)p;
  if (x->fregs < 13) {
    x->mRegData.fpr[x->fregs++] = d;
    /* skip two integer register file entries */
    if (x->iregs < 8)
      x->mRegData.gpr[x->iregs++]=((DCint*)&d)[0];
    if (x->iregs < 8)
      x->mRegData.gpr[x->iregs++]=((DCint*)&d)[1];
  }
  /* AND: push on stack */
  dcVecAppend(&x->mVecHead, &d, sizeof(DCdouble));
}
void osx_LongLong(DCCallVM* in_x, DClonglong L) {
  DCint* d = (DCint*) &L;
  dcArgInt(p, d[0]);
  dcArgInt(p, d[1]);
}

DCCallVM_vt gVT_osx = {
  &free, &reset, &mode
, &aBool, &aChar, &aShort
, &osx_Int,
, &aLong,
, &osx_LongLong,&osx_Float,&osx_Double,
, &aPointer
,(DCvoidvmfunc*)    &osx_Call
,(DCboolvmfunc*)    &osx_Call
/* [ ... lines omitted ... ] */
,(DCpointervmfunc*) &osx_Call
};
```

Listing 34: Call VM for ppc32.

The C interface of both call kernels is given below:

```
void ppc32_osx (void* target, DCRegData_ppc32* rdata, size_t size, void* buffer);
void ppc32_sysv(void* target, DCRegData_ppc32* rdata, size_t size, void* buffer);
```

The implementation of both call kernels is given in Listing 35. As their implementations are similarly structured we discuss them once.

```
 1  .globl ppc32_sysv:
 2  ppc32_sysv:        # prolog:
 3    mflr r0          # load return address to r0
 4    stw  r0,4(r1)    #   and save to link area
 5                   # allocate stack frame
 6    addi r0,r5,23    # r0 = size (=arg 2/r5)
 7                     #    + link area size (=8)
 8                     #    + align-1(=15)
 9    rlwinm r0,r0,0,0,27 # r0 = r0 and -16
10    neg r0,r0        # r0 := - align(size+link,16)
11    stwux r1,r1,r0   # store r1 and decrement
12                     # copy stack data
13    subi r6,r6,4     #   r6 = argbuf(arg 3:r6)-4
14    addi r7,r1,4     #   r7 = dest+8(link)-4
15    srwi r5,r5,2     #   r5 = size in words (*4)
16    cmpi cr0,r5,0    #   is size zero ?
17    beq  cr0,.done
18    mtctr r5         #   copy loop:
19  .next:
20    lwzu r0, 4(r6)
21    stwu r0, 4(r7)
22    bdnz .next
23  .done:
24    mr   r12,r3      # r12 = target func (arg 1:r3)
25    mtctr r12        # ctr = r12
26    mr   r11,r4      # r11 = register data buf
27    lwz r3 , 0(r11)  # load 8 integer 32-bit regs
28    lwz r4 , 4(r11)  #
29    lwz r5 , 8(r11)
30    lwz r6 ,12(r11)
31    lwz r7 ,16(r11)
32    lwz r8 ,20(r11)
33    lwz r9 ,24(r11)
34    lwz r10,28(r11)
35    lfd f1 ,32(r11)  # load 8 float 64-bit regs
36    lfd f2 ,40(r11)
37    lfd f3 ,48(r11)
38    lfd f4 ,56(r11)
39    lfd f5 ,64(r11)
40    lfd f6 ,72(r11)
41    lfd f7 ,80(r11)
42    lfd f8 ,88(r11)
43
44
45
46    creqv 6,6,6      # support for variadic args
47
48    bctrl            # call target func
49    lwz  r1, 0(r1)   # restore stack ptr to r1
50    lwz  r0, 4(r1)   # r0 = return address
51    mtlr r0          # store in link reg
52    blr              # return (jump link reg)
```

```
 1  .globl ppc32_osx
 2  .ppc32_osx:        # prolog:
 3    mflr r0          #   r0 = ret addr (link reg)
 4    stw  r0,8(r1)    #   save to link area
 5                     # allocate stack frame:
 6    addi r0,r5,39    #   r0 = size (=arg 2/r5)
 7                     #      + link area (=24)
 8                     #      + align-1   (=15)
 9    rlwinm r0,r0,0,0,27 # r0 = r0 and -16
10    neg r0,r0        # r0 := - align(size+link,16)
11    stwux r1,r1,r0   # store r1 and decrement stack
12                     # copy stack data:
13    subi r6,r6,4     #   r6 = argbuf(arg 3:r6)-4
14    addi r7,r1,20    #   r7 = dest+24(link)-4
15    srwi r5,r5,2     #   r5 = size in words (*4)
16    cmpi cr0,r5,0    #   is size zero ?
17    beq  cr0,.done
18    mtctr r5         #   copy loop:
19  .next:
20    lwzu r0, 4(r6)
21    stwu r0, 4(r7)
22    bdnz .next
23  .done:
24    mr   r12,r3      # r12 = target func (arg 1:r3)
25    mtctr r12        # ctr = r12
26    mr   r2, r4      # r2 = register data buf
27    lwz r3 , 0(r2)   # load 8 integer 32-bit regs
28    lwz r4 , 4(r2)
29    lwz r5 , 8(r2)
30    lwz r6 ,12(r2)
31    lwz r7 ,16(r2)
32    lwz r8 ,20(r2)
33    lwz r9 ,24(r2)
34    lwz r10,28(r2)
35    lfd f1 ,32(r2)   # load 13 float 64-bit regs
36    lfd f2 ,40(r2)
37    lfd f3 ,48(r2)
38    lfd f4 ,56(r2)
39    lfd f5 ,64(r2)
40    lfd f6 ,72(r2)
41    lfd f7 ,80(r2)
42    lfd f8 ,88(r2)
43    lfd f9 ,96(r2)
44    lfd f10,104(r2)
45    lfd f11,112(r2)
46    lfd f12,120(r2)
47    lfd f13,128(r2)
48    bctrl            # call target func
49    lwz r1, 0(r1)    # restore stack ptr to r1
50    lwz r0, 8(r1)    # r0 = return address
51    mtlr r0          # store in link reg
52    blr              # return (jump link reg)
```

Listing 35: Call kernel for `sysv` and `osx` calling convention on the `ppc32` architecture.

The return address in the link register is saved to the “link area” of the caller’s stack frame (line 3-4). The size of the new stack frame is computed based on the `size` adjusted by the head of the stack frame (line 6) and aligned to a 16-byte boundary (line 9-10). Finally the stack pointer is decremented to allocate a new stack frame (line 11). Note the different offsets for the "link area" and different sizes of the header structures with byte offset 4 within a 8-byte structure for `sysv`, and byte offset 8 in a 24-byte structure for `osx`. Data from `buffer` (in Register R6) are then copied to the stack frame (line 13-23). The `target` function's address (in Register R3) is saved to Register R12 and the control register (line 24-25). The pointer `rdata`, to register data `DCRegData_ppc32` (in Register R4), is loaded to a free volatile register, namely R11 for `sysv` or R2 for `osx`. All parameter registers are now reusable; integer (line 27-34) and floating-point registers (line 35-42 [`sysv`], 35-47 [`osx`]) are loaded with argument data. Note the instruction `creqv 6,6,6` for `sysv` (line 46) is required to support C variadic function calls, as specified in Zucker and Karhi (1995, Section 3.21). Finally the call is executed (line 48) using the target address stored in the control register. On function return the stack pointer and link register are restored (line 49-50); the latter is moved to register R0 and a *branch to link register* instruction passes back control to the caller (line 51-52).

5.4.15 Passing leading floating-point arguments via FPU registers (`mips-o32` port)

The `o32` ABI of `mips` uses a complex calling convention for handling floating-point arguments. The FPU comprises a set of 32 registers with each 32 bits of register storage. Two floating-point formats are supported, namely 32-bit and 64-bit floating-point values, which can be addressed by only the even numbered registers. For 64-bit values the storage of an even/odd numbered register pair is used. However, except for memory transfer, the odd-numbered registers are otherwise not used.

The calling convention reserves four FPU registers, F12 to F15, for passing the leading two arguments for `float` and `double` data types. Four 32-bit registers, R4 to R7, are utilized for passing the first 16 bytes of data (regardless of the argument type) and a corresponding homing area forms the top of the stack frame similar to the scheme used on `sparc-v7`. Small values are promoted to 32-bit values and all data objects are naturally aligned i.e. `double` and `long long` 64-bit objects are aligned to 8-byte boundaries and/or to an even numbered general-purpose register.

As the memory size for argument data of both register types equals to 16 bytes this could suggest a simple mapping between FPU and CPU register memory with regard to the homing area. Such a mapping was incorporated, for example, on `x86-64-x64`. However, `float` arguments passed in FPU registers cause skipping the next register which does not match the memory image of CPU registers. The following examples of argument types and corresponding mappings to CPU/FPU registers, given in Table 5.12, are intended to illustrate the assignment of arguments to registers in more detail.

Type	CPU registers				FPU registers			
sequence	R4	R5	R6	R7	F12	F13	F14	F15
f_1,f_2	f_1	f_2	...		f_1	-	f_2	-
f_1,d_1	f_1	-	d_1		f_1	-	d_1	
d_1,f_1	d_1		f_1	...	d_1		f_1	-
d_1,d_2	d_1		d_2		d_1		d_2	
i_1,f_1	i_1	f_1	...		-	-	f_1	-
l_1,f_1	l_1		f_1	...	-	-	f_1	-
l_1,d_1	l_1		d_1		-	-	d_1	
i_1,i_2,f_1,f_2	i_1	i_2	f_1	f_2	-	-	-	-

Table 5.12: Handling of floating-point arguments on the `mips-o32` ABI.

Each row gives an argument-type sequence and its corresponding assignment to the four general-purpose and floating-point parameter registers. Arguments are denoted as $\langle type \rangle_{index}$ with '*f*' for `float`, '*d*' for `double`, '*i*' for `int` and '*l*' for `long long`, and the index is used to distinguish objects of the same type. The "..." indicates the next free general-purpose register for subsequently following third arguments (although alignment restrictions may lead to a skip). While a mapping of CPU registers to corresponding FPU registers works for some cases, this scheme would fail for others. For example, for the sequences (f_1,f_2) in Row 1 and (i_1,f_1) in Row 5 the second argument would need to be passed in general-purpose register R6, instead of R5, in order to utilize a simple mapping scheme.

For the implementation a Call VM for `mips-o32` a separate buffer was incorporated to layout and transfer data to floating-point registers separately; the decision regarding exactly where to place data is based on an argument number counter `mArgCount`. The data structure of the Call VM for `mips-o32` is given below:

```
typedef struct {
  union {
    double d;
    float  f[2];
  } u[2];
} DCRegData_o32;
typedef struct {
  DCCallVM      mInterface;
  int           mArgCount;
  DCRegData_o32 mRegData;
  DCVecHead     mVecHead;
} DCCallVM_o32;
```

Listing 36 gives the implementation of the Call VM. Note that all *load argument* methods (prefixed 'a') update the counter by one. The variable is evaluated in the methods `aFloat` and `aDouble` where data are appended to the vector and also assigned to the register buffer for the first two arguments.

```
void aFloat(DCCallVM* p, DCfloat x) {
  DCCallVM_o32* x = (DCCallVM_o32*)p;
  dcVecAppend(&x->mVecHead,&x,sizeof(DCfloat));
  if (x->mArgCount < 2) {
#if defined(__MIPSEL__)
    x->mRegData.u[mArgCount].f[1] = x;
#else
    x->mRegData.u[mArgCount].f[0] = x;
#endif
  }
  x->mArgCount++;
}
void aInt(DCCallVM* p, DCint i) {
  DCCallVM_o32* x = (DCCallVM_o32*)p;
  dcVecAppend(&x->mVecHead, &i, sizeof(DCint));
  x->mArgCount++; }
void aBool (DCCallVM* p, DCbool  v){aInt(p,(DCint)v);}
void aChar (DCCallVM* p, DCchar  v){aInt(p,(DCint)v);}
void aShort(DCCallVM* p, DCshort v){aInt(p,(DCint)v);}
void aLong (DCCallVM* p, DClong  v){aInt(p,(DCint)v);}
void aLongLong(DCCallVM* p, DClonglong v) {
  DCCallVM_o32* x = (DCCallVM_o32*)p;
  /* align to 8-byte */
  dcVecSkip(&x->mVecHead,dcVecSize(&x->mVecHead)&4);
  dcVecAppend(&x->mVecHead,&v,sizeof(DClonglong));
  x->mArgCount++; }
```

```
void aDouble(DCCallVM* p, DCdouble x) {
  DCCallVM_o32* x = (DCCallVM_o32*)p;
  /* align 8-byte */
  dcVecSkip(&x->mVecHead, dcVecSize(&x->mVecHead) & 4);
  dcVecAppend(&x->mVecHead, &x, sizeof(DCdouble) );
  if (x->mArgCount < 2)
    x->mRegData.d[x->mArgCount] = x;
  x->mArgCount++;
}
void reset(DCCallVM* p) {
  DCCallVM_o32* x = (DCCallVM_o32*)p;
  dcVecReset(&x->mVecHead); x->mArgCount = 0; }
void call(DCCallVM* p, DCpointer target) {
  DCCallVM_o32* x = (DCCallVM_o32*)p;
  mips_o32(target,&x->mRegData,
  DC_MAX(16,((dcVecSize(&x->mVecHead))+7UL)&(-8UL));
  ,dcVecData(&x->mVecHead));
}
DCCallVM_vt gVT_o32 = {
  &free, &reset, &mode, &aBool, &aChar, &aShort, &aInt
, &aLong, &aLongLong, &aFloat, &aDouble, &aPointer
, (DCvoidvmfunc*)    &call
, (DCboolvmfunc*)    &call,
  /* [...] */
, (DCpointervmfunc*) &call
};
```

Listing 36: Call VM for `mips-o32`.

The C interface to the call kernel is given below:

```
void mips_o32( void* target, DCRegData_o32* regdata, size_t size, void* buffer );
```

The call kernel, depicted in Listing 37, transfers the `buffer` data to the stack and copies the first 16 bytes to registers R4 to R7 while `regdata` is passed to FPU registers F12 to F15. Since data are also written to the homing area of the stack the call kernel is also suitable for calling standard and variadic functions.

```
 1 mips_o32:
 2   addiu $sp,$sp,-8    # Alloc local stack frame
 3   sw    $31,4($sp)    # Save link register
 4   sw    $fp,0($sp)    # Save frame pointer register
 5   move  $fp,$sp       # fp = sp
 6   sub   $sp,$sp,$6    # Alloc call stack frame
 7   move  $12,$7        # r12 = buffer (r7)
 8   move  $14,$sp       # r14 = sp
 9 .next:                # 'Copy' loop:
10   beq   $6 , $0,.skip #   size > 0 ?
11   nop
12   lw    $2 , 0($12)   #   read word
13   nop
14   sw    $2 , 0($14)   #   write word
15   addiu $12,$12,  4   #   update read/write pointers
16   addiu $14,$14,  4
17   addiu $6 , $6, -4   #   update size counter
18   j     .next
19   nop
20 .skip:                # Load FPRs (f12-f15)
21   l.d   $f12, 0($5)   #   load 64-bit data in f12/f13
22   l.d   $f14, 8($5)   #   load 64-bit data in f14/f15
23   move  $12, $7       # r12 = stack data
24   move  $25, $4       # r25 = target function
25   lw    $4 , 0($12)   # Load GPRs (r4-r7)
26   lw    $5 , 4($12)
27   lw    $6 , 8($12)
28   lw    $7 ,12($12)
29   jalr  $25           # Call target function
30   nop
31   move  $sp,$fp       # Restore stack pointer
32   lw    $31,4($sp)    # Restore link register
33   lw    $fp,0($sp)    # Restore frame pointer
34   addiu $sp,$sp,8     # Clean up local stack frame
35   j     $31           # Jump to link register (return)
36   nop
```

Listing 37: Call kernel for `mips-o32`.

At first a local stack frame is allocated (line 2) in which the link register and frame pointer are stored (line 3-4). The stack pointer is saved as the new frame pointer (line 5) and then decremented by `size` to allocate a new stack frame for the next call (line 6). Note that `size` (Register R6) is guaranteed to be equal or larger than 16 bytes and aligned to 8 bytes (see Function `call` in Listing 36). The `buffer` (in Register R7) is then copied to the stack including the homing area (line 9-20). FPU register data are loaded from `regdata` (in Register R5) into four floating-point parameter registers F12 to F15 (line 21-22). The first four words of `buffer` are loaded to the four general-purpose parameter registers R4 to R7 (line 25-28). Finally the call is executed (line 29). After that the stack frame of the call is cleaned up by moving the frame pointer to the stack pointer (line 31). The original frame pointer and link register are restored (line 32-33), the local stack frame is cleaned up (line 34) and, finally, control is passed back to the caller (line 35-36). Note the `nop` instructions which need to be used for branch delay slots (line 11, 19, 30 and 36); also one `nop` is used to prevent *load delays* between direct dependencies when loading/storing data (line 13) which is suggested in Sweetman (2007, p.52).

The MIPS architecture supports two endian data models. Typically "`mips`" implies a *big endian* model also referred to as "`mipseb`". The little endian variant is known as "`mipsel`". For the implementation of the Call VM and the call kernel the endian model has an influence for storing `float` data in the correct location in the register buffers (see Function `aFloat` in Listing 36) and for transfer to the FPU registers which needs to change the 32-bit word transfer order for passing data to F12/F13 and F14/F15; this is managed by the pseudo-instruction `l.d` (line 21-22 of Listing 37) which generates two instructions of the form `lwc1 Fn, offset(Rn)` and sets up the "offset" that corresponds to the endian model of the target machine during assembling.

5.4.16 Passing arguments via type-specific load instructions (`mips-n64` port)

MIPS CPUs, based on the `MIPS32/64` ISA, incorporate a modern FPU programming model that improves on the `mips-o32` ABI. It offers support for 64-bit registers which support 32-bit and 64-bit data types. The calling convention of the `n64` ABI reserves eight 64-bit general-purpose and floating-point registers for passing of arguments. The two register files are logically inter-connected. When one argument is assigned to be passed via one register, the corresponding register of the other set is marked as occupied, similar to the `x86-64-x64` ABI. However, no homing area is reserved on the stack. Effectively, eight arguments can be passed via registers.

The data structure of the Call VM for `n64` contains two buffers for each register type; for the floating-point buffer a union structure for `float` and `double` values is used:

```
typedef struct {
  DClonglong   mGPR[8];
  union {
    DCfloat  f;
    DCdouble d;
  } mFPR[8];
  DClonglong   mBitMask;
} DCRegData_n64;
typedef struct {
  DCCallVM      mInterface;
  int           mRegCount;
  DCRegData_n64 mRegData;
  DCVecHead     mVecHead;
} DCCallVM_n64;
```

The call kernel loads both sets of eight registers. However, in order to load floating-point registers we needed type-specific load instructions to distinguish 32-bit and 64-bit value transfer. A bit mask `mBitMask` is used as part of the state machine, where the lower 8 bits indicate the specific sub-type whether a value needs to be transferred as a `float` (bit cleared) or as a `double` (bit set). The implementation of the Call VM is given in Listing 38.

During initialization and resetting of the Call VM the bit mask is cleared. Small integer argument types are promoted to 64 bits. If 32-bit floating-point values need to be passed via stack (if `mRegCount` > 7) the value is append to the vector and a 4-byte padding is added so that the stack remains 8-byte aligned. The method implementation for `dcArgDouble`, named `aDouble`, updates the bit mask accordingly for the first eight arguments. The call kernel has the following C interface:

```
void mips_n64(void* target, DCRegData_n64* regdata, size_t size, void* buffer);
```

The assembly source for the call kernel is given in Listing 39.

At first a local stack frame is allocated on the stack (line 2) in which link register, frame pointer and global pointer are saved (line 3-5). The stack pointer is copied to the frame pointer (line 6) and then decremented by `size` (Register R6) to allocate a new stack frame (line 7). Then `buffer` data are

```
void aLongLong(DCCallVM* p, DClonglong v) {
  DCCallVM_mips_n64* x = (DCCallVM_mips_n64*)p;
  if (x->mRegCount < 8)
    x->mRegData.mGPR[x->mRegCount++] = v;
  else
    dcVecAppend(&x->mVecHead,&v,sizeof(DClonglong));
}
void aInt(DCCallVM* p,DCint     v){
  aLongLong(p,(DClonglong)v); }
void aBool (DCCallVM* p,DCbool    v){
  aLongLong(p,(DClonglong)v); }
void argChar (DCCallVM* p,DCchar    v){
  aLongLong(p,(DClonglong)v); }
void aShort(DCCallVM* p,DCshort   v){
  aLongLong(p,(DClonglong)v); }
void aLong (DCCallVM* p,DClong    v){
  aLongLong(p,(DClonglong)v); }
void aPtr  (DCCallVM* p,DCpointer v){
  aLongLong(p, *(DClonglong*)&v);}
void reset(DCCallVM* p) {
  DCCallVM_mips_n64* x = (DCCallVM_mips_n64*)p;
  dcVecReset(&x->mVecHead);
  x->mRegCount = 0;
  x->mRegData.mBitMask = 0LL;
}
```

```
void aFloat(DCCallVM* p, DCfloat v) {
  DCCallVM_n64* x = (DCCallVM_n64*)p;
  if (x->mRegCount < 8) {
    x->mRegData.mFPR[x->mRegCount++].f = x;
  } else {
    dcVecAppend(&x->mVecHead, &v, sizeof(DCfloat) );
    dcVecSkip  (&x->mVecHead, sizeof(DCfloat) );
  }
}
void aDouble(DCCallVM* p, DCdouble v) {
  DCCallVM_n64* x = (DCCallVM_n64*)p;
  if (x->mRegCount < 8) {
    x->mRegData.mBitMask |= 1<<( x->mRegCount );
    x->mRegData.mFPR[x->mRegCount++].d = v;
  } else
    dcVecAppend(&x->mVecHead, &v, sizeof(DCdouble) );
}
void call(DCCallVM* p, DCpointer t) {
  DCCallVM_mips_n64* x = (DCCallVM_mips_n64*)p;
  mips_n64(t, &x->mRegData,
    dcVecSize(&x->mVecHead), dcVecData(&x->mVecHead));
}
```

Listing 38: Call VM for `mips-n64`.

copied to the stack frame (line 8-18) and the eight general-purpose registers are loaded (line 21-28). The bit mask is loaded to Register R14 (line 34) followed by a sequence of eight conditional blocks for loading floating-point registers. In each block a bit is tested (e.g. line 35) in order to load a 32-bit (e.g. line 37) or 64-bit (e.g. line 39) value, respectively. Code is given for the first (line 35-39) and the last conditional block (line 41-45) and is omitted for blocks in between. After loading all floating-point registers the preparation of arguments is complete and the call can be executed (line 47). On function return the stack pointer is restored to the local frame (line 47) from where other registers are restored (line 48-50). The local frame is removed (line 51) and control is passed back to the caller (line 53).

```
mips64_n64:
      dsubu  $sp,$sp,64    # Alloc local frame
      sd     $ra,48($sp)   # Save link register
      sd     $fp,40($sp)   # Save frame pointer
      sd     $gp,32($sp)   # Save global pointer
      move   $fp,$sp       # Set fp = stack pointer
      dsubu  $sp,$sp,$6    # Alloc call frame
      move   $12,$7        # Set r12 = buffer
      move   $14,$sp       # Set r14 = sp
.next:                     # 'Copy' loop:
      beq    $6,$0,.skip   #   size == 0 ?
      daddiu $6,$6,-8
      ld     $2, 0($12)
      sd     $2, 0($14)
      daddiu $12,$12,8
      daddiu $14,$14,8
      b .next
.skip:                     # Free GPRs:
      move   $25,$4        #   r25 = target
      move   $13,$5        #   r13 = regdata
      ld     $4 ,  0($13)  # Load 8 x 64-bit GPRs
      ld     $5 ,  8($13)
      ld     $6 , 16($13)
      ld     $7 , 24($13)
      ld     $8 , 32($13)
      ld     $9 , 40($13)
      ld     $10, 48($13)
      ld     $11, 56($13)




                           # Load 8 x 32/64-bit FPRs
      ld     $14 ,128($13) #   r14 = bit mask
.f0:  and    $15 ,$14,1    # Bit 0 set ?
      bgtz   $15 ,.d0
      l.s    $f12,64($13)  #   load 32-bit float
      j      .f1
.d0:  l.d    $f12,64($13)  #   load 64-bit double
   #  [...] -- CODE OMITTED FOR BIT 2-6 --
.f7:  and    $15, $14,128  # Bit 7 set ?
      bgtz   $15, .d7
      l.s    $f19,120($13) #   load 32-bit float
      j      .call
.d7:  l.d    $f19,120($13) #   load 64-bit float
.call:jal    $ra , $25     # Call function
      move   $sp , $fp     # Restore stack pointer
      ld     $ra ,48($sp)  # Restore link register
      ld     $fp ,40($sp)  # Restore frame pointer
      ld     $gp ,32($sp)  # Restore global pointer
      daddu  $sp ,$sp,64   # Free local frame
      j      $ra           # Jump to return address
```

Listing 39: Call kernel for `mips-n64` ABI.

5.5 Dynamic Callback Handlers

The previous section covered a generic technique for calling precompiled code from within a dynamic execution context. Our focus remains on function calls, but in this section we regard the complementary problem.

We discuss a portable software framework to execute scripting functions on behalf of C callbacks. This was motivated by the aim to allow scripting users to implement callbacks of C libraries by means of user-defined scripting functions. The framework is provided as an open-source C library, named `dyncallback`, contributed to the *DynCall* project. In this sense `dyncallback` complements `dyncall` to offer a dynamic *bi-directional* inter-language call bridge between compiled and interpreted programming languages.

5.5.1 Introduction

Callbacks are offered by C APIs as a complementary interface method to the usual function call interfaces. APIs make use of callbacks to allow users to define *behaviour* for library-specific events that are activated during execution of a library's function. Figure 5.15 illustrates the basic components of a callback in C libraries.

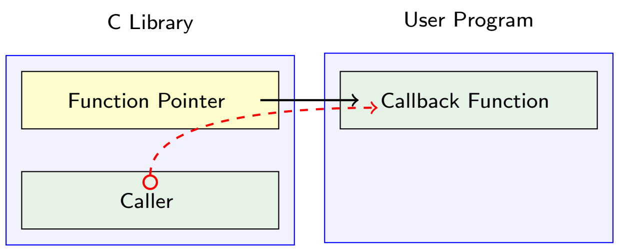


Figure 5.15: Anatomy of C Callbacks.

The figure shows a running program comprising two run-time modules (light blue), namely a shared "C library" and an executable "user program". At a finer level two code objects (green) and one data object (yellow) are depicted. The "Callback Function" is defined as part of the application's code but for the implementation of a library's callback. At first the application code *registers* this C function as a *callback* via a library-specific mechanism so that the function's code address is stored in a "Function Pointer" variable of the library; the reference is shown as the black solid arrow. Subsequently the library code activates the callback, for example to compare elements when sorting a user-defined data type (e.g. the `qsort` function) or to notify an event during parsing an XML file (e.g. the `expat` XML parser library) as illustrated in Section 4.6. The control flow (dashed red line) of the callback is activated by the "Caller" code section. In preparation for the call arguments are loaded to the stack and registers in accordance with the C function type of the callback and the calling conventions of the platform. Finally a machine-level `call` instruction is executed by using the "Function Pointer" variable as the target address and control flow is passed from the library to the application's "Callback Function". Note that the callee side needs to conform with the function type and calling convention of the callback caller otherwise the process possibly crashes.

Interpreters usually offer an *embedding interface* for executing scripting functions from C. Language bindings to C libraries make use of this facility to provide support for callbacks. Typically a *bindings* interface is offered to specify a user-defined scripting function to be registered as a C callback. The *bindings* package provides C callback functions that take the role of adapters which are registered as callbacks; when they are activated the calls are forwarded at interpreter-level to corresponding scripting functions. However, for each distinct function type a separate C function needs to be implemented as adapter by the *bindings* package. Furthermore context information about the interpreter and the target scripting function needs to be incorporated by different means depending on the library's registration and callback interface.

5.5.2 Objective

Function pointer objects represent the key interface objects for registration and activation of callbacks in C. If a generic method is made available to scripting languages for *wrapping* user-defined scripting functions as plain C function pointers a much broader spectrum of C API services can be accessed dynamically than by a foreign function call interface alone. However, a C function can not be used as a *generic* callback adapter *directly* for receiving incoming calls from call-outs of arbitrary function types.

The problem is as challenging as is the case of `dyncall`. Callbacks with arbitrary calling sequences are executed at machine level. In order to handle them from a scripting language in a generic way they need to be processed by a generic handler code that translates the function call to the interpreter; it is desirable that this interface can be implemented in C.

We discuss a portable software library with a C interface that provides an abstraction layer for handling of function calls on the callee side. Included here is an abstraction for processing arbitrary sequences of arguments.

Our objective is to offer this layer as a generic and dynamic service for wrapping function objects of a dynamic scripting language as native function pointer objects. When calling the latter from precompiled code control is delegated to a *callback handler* function where an interpreter-based call to the scripting function is executed with access to the sequence of arguments and further context information.

5.5.3 Approach

In order to handle arbitrary function calls on the callee side the state of machine resources for passing arguments can be captured and encapsulated as an abstract argument iterator object. This allows argument processing to be delegated to a handler in C later. Such a generic function entry handler can be implemented in assembly language, similarly to call kernels of `dyncall`. However, a pointer to this function is not suitable as a *wrapper* since additional context information needs to be made available to the handler, such as the C function type of the callback and the reference to the target scripting function. Therefore we use a hybrid "code" and "data" adapter object for this purpose as depicted in Figure 5.16.

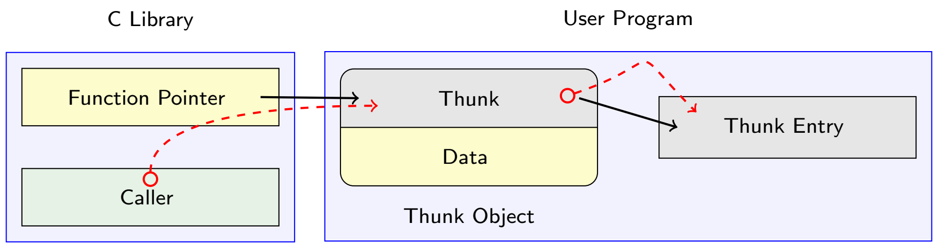


Figure 5.16: Anatomy of Thunks Objects.

The object begins with a short section of machine code (gray), named a *Thunk* header, which makes it a *callable* object, followed by a data section (yellow) which provides an environment for context data. When calling *Thunk* objects control flow is forwarded to a user-defined *Thunk Entry* routine regardless of the type of the function call.

Dynamic creation of natively callable objects at run time poses a challenge in C. *Callable* objects are typically created at build time by compilers or assemblers. However, the code of a thunk is relatively small (between 12 to 20 bytes) and we can use a tiny code generator (approximately 15 lines of C) for each platform port. A thunk code header performs two tasks:

1. A reference of the *Thunk* object is loaded in a specific volatile register that is not otherwise reserved as a parameter register. In this framework the register is named the *thunk register*.
2. Control flow is passed to the *Thunk Entry* via a `jump` instruction.

On entry to the thunk entry function the execution environment is that after the caller's `call` except that additional context "Data" are available via the thunk register. Thus a specific routine can be specified as "thunk entry" which captures the current state of the call and then forwards this information to a C handler function for further evaluation of the call.

5.5.4 Architecture

The `dyncallback` framework provides a C interface for dynamic run-time creation of *Callback* object types. The object type is special because it is *callable* and contains context-specific *data*. As depicted in Figure 5.17, a pointer to this object can be used as a plain function pointer for registration of C callbacks regardless of the required function type but the caller code needs to comply with the calling convention supported by the framework.

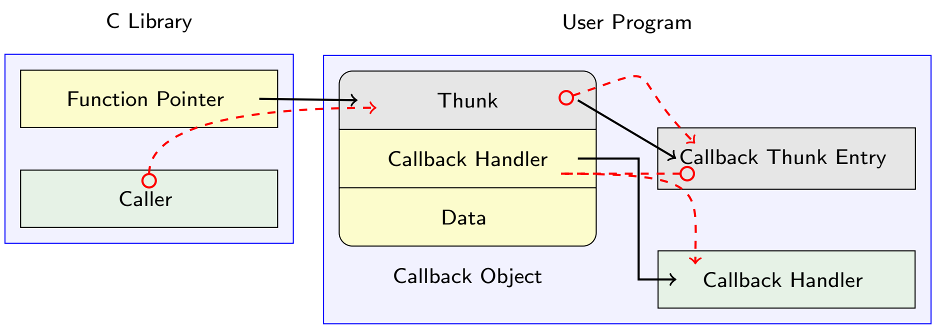


Figure 5.17: Anatomy of Callback Objects.

Callback objects are associated with a C "Callback Handler" function and a user "Data" pointer. The framework ensures that calls to callback objects are delegated to the handler function. Several parameters are passed to the handler, such as the sequence of arguments, storage for passing callback-specific return values and the user data pointer.

Since a handler function needs access to the arguments of arbitrary function types, the sequence needs to be captured before entry to the C handler function. Technically this needs to be done very early after the call. Since arguments are loaded in registers and stack memory their values need to be preserved for later iteration, in particular to free parameter registers for the next C call to the handler function.

Callback objects are based on Thunk objects configured to bypass calls to a "Callback Thunk Entry" function. The main task of the callback thunk entry is to delegate the call as data arguments to the handler. As part of this task arguments are encapsulated by an object, named *Argument Iterator*, that will be dynamically allocated on the stack as depicted in Figure 5.18.

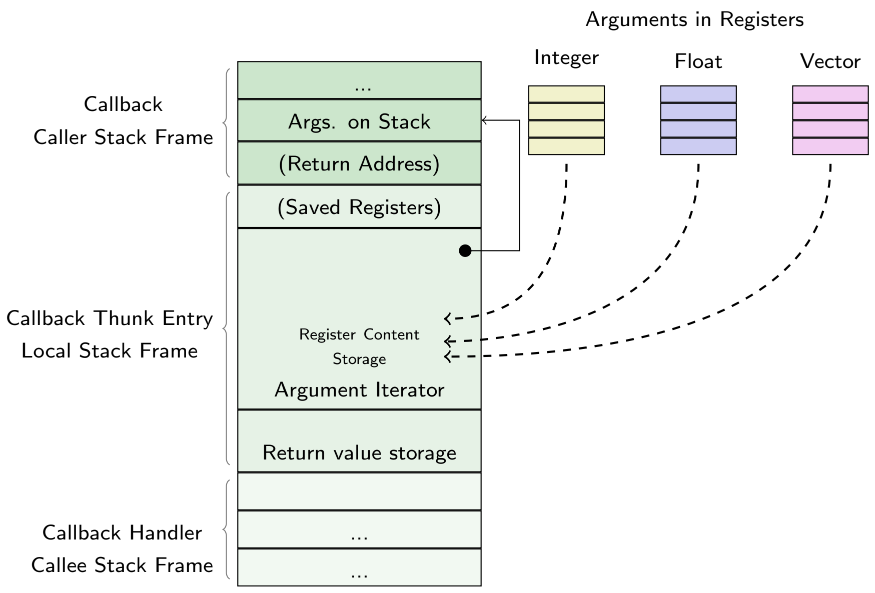


Figure 5.18: Dynamic allocation of Argument Iterator on Callback Thunk Entry's stack frame.

The argument iterator is a platform-specific data object that needs to take the characteristics of calling conventions and ABIs into account. For all parameter registers corresponding buffers need to be reserved in that object. The callback thunk entry code is responsible for its allocation and initialization; the contents of parameter registers are saved to buffers; the parameter region on the callback's call stack frame is saved to a pointer variable and counter variables are initialized.

In addition a second object is dynamically allocated on the stack for storing return values. Finally the handler of the"Callback object" is called. The return value of the handler indicates the return type of the callback by a type code so that corresponding registers are loaded with return values from stack storage. Finally, control flow returns back to the "Caller".

Type information about the function type of a callback is typically stored in user data so that the handler can iterate type-safe over arguments and return appropriate value types. The argument iterator object offers a C interface for type-specific iteration. Note that the implementation of these functions is also specific to a calling convention; the argument iterator represents a state machine similar to the Call VM of `dyncall` but for the opposite direction - to *read* arguments in sequence.

For the implementation of a *dynamic callback handling* facility as an extension to a dynamic programming language three tasks need to be accomplished in C:

1. User data: Definition of a user-defined structure for information about the callback type (e.g. fuction type signature) and the target execution environment (e.g reference to a scripting function).

2. Handler function: Implementation of a handler function that iterates over incoming arguments to prepare a call to a scripting function and then to execute it. According to the function type, a return value is passed back via a data structure and its value type is returned via a return code.

3. Wrapper interface: Implementation of a scripting interface to wrap user-defined scripting functions and corresponding C function types of the callback (e.g. function type signature) as a pointer object. A user data object is created and initialized accordingly. Finally the callback object is created, linked with the handler and user data, and then returned as an external pointer object.

Figure 5.19 illustrates the application of this framework to R contributed by the `rdyncall` package as discussed in Section 4.6. The figure shows a "Callback object", created in R via a call to `new.callback` that *wrapped* a user-defined "R Function" as a plain function pointer registered to a C library as a callback and stored in a "Function Pointer" object. Callback objects, created by `rdyncall`, are set to the global "R Callback Handler" function. This function implements the bridge between calls from statically compiled C code to dynamically interpreted function calls in R. Note the "R Callback Userdata" which supplies the handler with callback-specific information such as the reference to the "R function" and the C function type (not depicted).

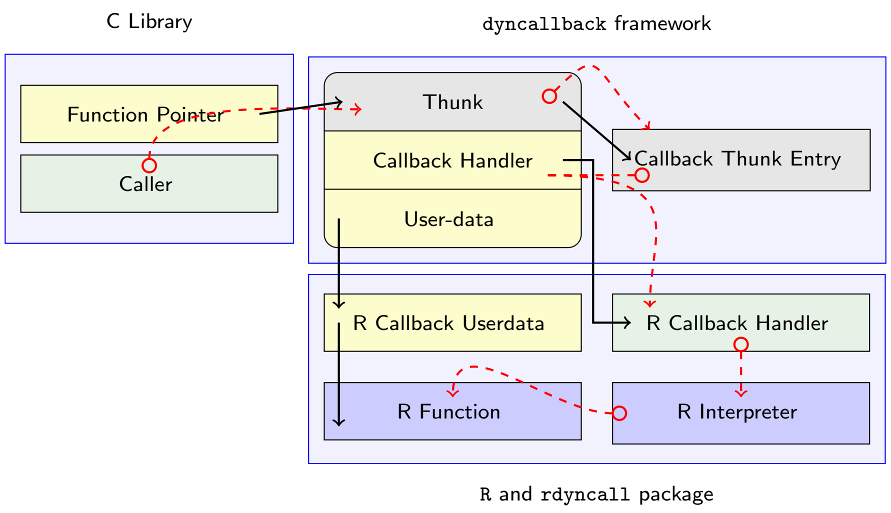


Figure 5.19: Anatomy of C Callbacks to R based on `rdyncall` and `dyncallback`.

5.5.5 Interface

The C API of `dyncallback` is accessible via inclusion of the C header file `dyncall_callback.h`. A similar API name scheme as for `dyncall` (see Section 5.4.5) is used for `dyncallback` except for callback and argument iterator functions which are prefixed `dcb` instead of `dc`.

5.5.5.1 Creation of Callback objects

Callback objects are created dynamically at run time via the following interface:

```
DCCallback* dcbNewCallback(const char* sig, DCCallbackHandler* hf, void* ud);
```

`hf` specifies the handler function, which is called on behalf of a callback call.

`ud` specifies a user data pointer which is associated with the callback and passed as an argument to the handler function during callback activation. The function returns a pointer to `DCCallback`; in order to call it from C code it can be type cast to a function pointer.

The signature `sig` is currently used on `x86-32` for the the selection of the calling convention and possibly the computation of stack clean-up sizes, such as required by the `stdcall` ABI. Other architectures currently ignore this parameter.

5.5.5.2 Callback Handler Interface

Handler functions need to conform with the following function type as interface convention:

```
char MyCallbackHandler (DCCallback* p, DCArgs* a, DCValue* r, void* ud);
```

As a response to a function call on a callback object a corresponding handler is called with four arguments: a pointer to the callback object, given by `p`, a pointer to the argument iterator, given by `a`, a pointer to a buffer to store return values, given by `r` and a pointer to the user data associated with the callback object during creation, given `ud`.

Handlers should return different return values via `r` according to the callback's function type. In addition the handler's return value is a *return type code* which indicates the return type of the callback to the underlying framework; the latter moves the return value from the storage to appropriate registers according to this type code. The type code is a single character code based on the notation for basic C types of *DynPort* (see Section 3.7.3.2).

Note that, if no other means are used for function type information, the data field `sig` of `p`, gives the signature `sig` as it was passed during creation of the callback object.

5.5.5.3 Argument Iterator

A handler processes arguments by using the *Argument Iterator* and a function type of the callback (e.g. a signature). The interface for the argument iterator comprises a set of C functions, one for each supported argument type. All functions have a common naming and function type pattern as follows:

$\langle type \rangle$ `dcbArg` $\langle type \rangle$ `(DCArgs* pArgIter)`

where $\langle type \rangle$ is one of `Bool`, `Char`, `UChar` and so on. The following gives a complete list of the interface:

```
typedef struct DCArgs DCArgs;
DCbool      dcbArgBool      (DCArgs*);
DCchar      dcbArgChar      (DCArgs*);
DCshort     dcbArgShort     (DCArgs*);
DCint       dcbArgInt       (DCArgs*);
DClong      dcbArgLong      (DCArgs*);
DClonglong  dcbArgLongLong  (DCArgs*);
DCuchar     dcbArgUChar     (DCArgs*);
DCushort    dcbArgUShort    (DCArgs*);
DCuint      dcbArgUInt      (DCArgs*);
DCulong     dcbArgULong     (DCArgs*);
DCulonglong dcbArgULongLong (DCArgs*);
DCfloat     dcbArgFloat     (DCArgs*);
DCdouble    dcbArgDouble    (DCArgs*);
DCpointer   dcbArgPointer   (DCArgs*);
```

The arguments are traversed from *left-to-right* according to the function type of the callback.

5.5.5.4 Return Value Storage

The `DCValue` data structure defines a variant union data type that contains a data value field for each supported return type.

```
typedef union DCValue_ DCValue;
union DCValue_
{
  DCbool        B;
  DCchar        c;
  DCuchar       C;
  DCshort       s;
  DCushort      S;
  DCint         i;
  DCuint        I;
  DClong        j;
  DCulong       J;
  DClonglong    l;
  DCulonglong   L;
  DCfloat       f;
  DCdouble      d;
  DCpointer     p;
  DCstring      Z;
};
```

The field name corresponds with the return type code of the handler.

5.5.5.5 Destroying Callback Objects

Callback objects that are no longer needed can be destroyed by calling a destructor as follows:

```
void dcbFreeCallback(DCCallback* pcb);
```

5.5.5.6 Querying parameters

The user data of a callback can be queried as follows:

```
void* dcbGetUserData(DCCallback* pcb);
```

5.5.6 Sample Application

In Section 4.6 we discussed a facility of the `rdyncall` R package for wrapping R scripting functions as callback objects to C callbacks. We now give its implementation which is based on `dyncallback`.

The R constructor function `new.callback( signature, fun, envir )` is implemented as a small R wrapper that delegates a call to the C implementation:

```
new.callback <- function(signature, fun, envir=new.env())
{
  stopifnot( is.character(signature) )
  stopifnot( is.function(fun) )
  stopifnot( is.environment(envir) )
  .Call("new_callback", signature, fun, envir, PACKAGE="rdyncall")
}
```

Each callback object is associated with a user-defined data structure that contains R-specific callback data such as the reference to the R scripting function and a reference to the environment `envir` where the R function should be evaluated:

```
typedef struct
{
  int         disabled;   /* disabled marker */
  SEXP        fun;        /* R function */
  SEXP        rho;        /* environment */
  int         nargs;      /* number of arguments (cached) */
  const char* signature;  /* signature without callmode prefix (cached) */
} RData;
```

The field `fun` holds the reference to the R function closure object that will be called upon callback activation by the R callback handler. The callback handler creates a call expression that holds this S-Expression as a first object. Further S-Expressions in the list are appended with atomic R value objects that are initialized by incoming C arguments via the argument iterator. The field `rho` holds the reference to the environment in which the call expression is evaluated. The field `nargs` is used as a cached number of arguments derived from the signature. The field `signature` comprises the function type signature without any calling convention prefixes.

The callback object constructor in C is given below:

```
SEXP new_callback(SEXP sig, SEXP fun, SEXP rho) {
  const char* s = CHAR( STRING_ELT( sig, 0 ) );          // get signature C string pointer
  RData* rdata = Calloc(1, RData);                       // allocate RData object
  DCCallback* cb = dcbNewCallback( s, &handler, rdata);  // create DCCallback object
  rdata->disabled = 0;                                   // unmark as disabled
  rdata->fun = fun; R_PreserveObject(fun);               // protect fun and rho
  rdata->rho = rho; R_PreserveObject(rho);
  if ( *s == '_') s += 2;                                // skip any calling convention prefix
  rdata->signature = s;                                  // store signature string
  int nargs = 0; while( (*s++) != ')') nargs++;          // count arguments and store
  rdata->nargs = nargs;
  SEXP ans = R_MakeExternalPtr(cb,R_NilValue,R_NilValue); // create external pointer
  R_RegisterCFinalizerEx(ans, &finalizer, TRUE);         // register finalizer
  return ans;
}
```

The data structure `RData` will be allocated as user data for the callback and `rdyncall`'s callback handler. A full scan over the signature is performed to determine the beginning of the arguments and the number of arguments. The function and environment are referenced for external use via `R_PreserveObject` to protect them from garbage collection.

The finalizer function will be called if there are no references to a callback object in R and the garbage collector is running.

```
void finalizer(SEXP x) {
  DCCallback* cb = R_ExternalPtrAddr(x);                // obtain DCCallback object
  RData* rdata = (RData*) dcbGetUserData(cb);        // obtain R user data
  R_ReleaseObject(rdata->fun);                          // release fun and rho
  R_ReleaseObject(rdata->rho);
  Free(rdata);                                          // free R_Callback object
  dcbFreeCallback(cb);                                  // free Callback object
}
```

The finalizer reverses all allocations of the constructor and releases the function and environment for garbage collection.

When calling a callback from C the callback handler “`handler`” is activated. As a convention of the `rdyncall` package if an R callback script function failed during execution it is disabled for subsequent callbacks. If the callback’s user data field `disabled` is set control is passed back immediately.

```
char handler(DCCallback* pcb, DCArgs* args, DCValue* rval, void* userdata){
  RData* rdata = (RData*) userdata;
  if (rdata->disabled) return 'v';
```

A call expression is allocated to prepare the function call.

```
  SEXP call;
  PROTECT( call = allocList(rdata->nargs + 1) );
  SET_TYPEOF(call, LANGSXP); SETCAR( call, rdata->fun );
```

`call` is a root S-Expression object that needs to be protected from garbage collection. It follows the iteration loop to create a sequence of R argument objects.

```
  SEXP        x    = CDR(s);
  char const *ptr;
  char        ch;
  for(ptr = rdata->signature ; (ch = *ptr) != ')' ; ptr++ ) {
    SEXP item;
    switch(ch) {
      case 'B': item = ScalarLogical((dcbArgBool(args)==DC_FALSE)?FALSE:TRUE); break;
      case 'c': item = ScalarInteger( (int) dcbArgChar(args) ); break;
      case 'C': item = ScalarInteger( (int) dcbArgUChar(args) ); break;
      case 's': item = ScalarInteger( (int) dcbArgShort(args) ); break;
      case 'S': item = ScalarInteger( (int) dcbArgUShort(args) ); break;
      case 'i': item = ScalarInteger( (int) dcbArgInt(args) ); break;
      case 'I': item = ScalarReal( (double) dcbArgUInt(args) ); break;
      case 'j': item = ScalarReal( (double) dcbArgLong(args) ); break;
      case 'J': item = ScalarReal( (double) dcbArgULong(args) ); break;
      case 'l': item = ScalarReal( (double) dcbArgLongLong(args) ); break;
      case 'L': item = ScalarReal( (double) dcbArgULongLong(args) ); break;
      case 'f': item = ScalarReal( (double) dcbArgFloat(args) ); break;
      case 'd': item = ScalarReal( dcbArgDouble(args) ); break;
      case 'z': item = mkString( dcbArgPointer(args) ); break;
      case 'p': item = R_MakeExternalPtr( dcbArgPointer(args), R_NilValue, R_NilValue ); break;
    }
    SETCAR( x, item );
    x = CDR(x);
  }
```

The argument iterator is operated according to the signature. At each step atomic R scalar values are constructed and appended to the call expression object. The objects need not be protected since they are inserted in the call expression which is already protected from garbage collection. The call object is complete and can be evaluated as follows:

```
  int error = 0;
  SEXP ans;
  PROTECT( ans = R_tryEval( call, rdata->rho, &error ) );
```

If an error occures the callback object is disabled for subsequent callbacks.

```
  if (error) {
do_error:
    warning("error during R callback (disabled).");
    rdata->disabled = 1;                                    // mark as 'disabled'
    UNPROTECT(2);
    return 'v';
  }
```

Otherwise the return value is returned as follows:

```
  ch = *ptr;
  switch(ch) {
    case 'v': break;
    case 'B':
      ch = 'i';
      switch( TYPEOF(ans) ) {
        case INTSXP:  rval->i = (INTEGER(ans)[0] == 0     ) ? DC_FALSE : DC_TRUE; break;
        case LGLSXP:  rval->i = (LOGICAL(ans)[0] == FALSE ) ? DC_FALSE : DC_TRUE; break;
        case REALSXP: rval->i = (   REAL(ans)[0] == 0.0   ) ? DC_FALSE : DC_TRUE; break;
        default:      goto do_error;
      } break;
    case 'c': case 'C': case 's': case 'S': case 'i': case 'I':
      ch = 'i';
      switch( TYPEOF(ans) ) {
        case INTSXP:  rval->i = (int) INTEGER(ans)[0]; break;
        case REALSXP: rval->i = (int)    REAL(ans)[0]; break;
        case LGLSXP:  rval->i = (int) LOGICAL(ans)[0]; break;
        default:      goto do_error;
      } break;
    case 'j': case 'J':
      ch = 'j';
      switch( TYPEOF(ans) ) {
        case INTSXP:  rval->j = (long) INTEGER(ans)[0]; break;
        case REALSXP: rval->j = (long)    REAL(ans)[0]; break;
        case LGLSXP:  rval->j = (long) LOGICAL(ans)[0]; break;
        default:      goto do_error;
      } break;
    case 'l': case 'L':
      ch = 'l';
      switch( TYPEOF(ans) ) {
        case INTSXP:  rval->l = (long) INTEGER(ans)[0]; break;
        case REALSXP: rval->l = (long)    REAL(ans)[0]; break;
        default:      goto do_error;
      } break;
    case 'f':
      ch = 'f';
      switch( TYPEOF(ans) ) {
        case INTSXP:  rval->f = (float) INTEGER(ans)[0]; break;
        case REALSXP: rval->f = (float)    REAL(ans)[0]; break;
        default:      goto do_error;
      } break;
    case 'd':
      ch = 'd';
      switch( TYPEOF(ans) ) {
        case INTSXP:  rval->d = (float) INTEGER(ans)[0]; break;
        case REALSXP: rval->d = (float)    REAL(ans)[0]; break;
        default:      goto do_error;
      } break;
    case 'p': switch( TYPEOF(ans) ) {
      ch = 'p';
      case NILSXP:    rval->p = NULL; break;
      case EXTPTRSXP: rval->p = R_ExternalPtrAddr(ans); break;
      default:        goto do_error;
    } break;
  }
  UNPROTECT(2);
  return ch;
}
```

The result of the evaluation is cast to the corresponding C return value, as indicated by the signature, and stored in the object pointed to by the argument `DCValue* rval` (3rd argument) and a corresponding return type character is passed back.

5.5.7 Implementation

In this section we discuss the implementation of the `dyncallback` library on a number of platforms. Table 5.13 gives an overview of available ports of the library. Note that a subset of ports in `dyncall` have been made available to `dyncallback`; `ppc32-sysv` and `x86-32-p9` ports as well as support for `mips` and `sparc` architectures are currently work in progress.

Arch/OS	Microsoft Windows	Apple	Linux,(Net,Free,Open) BSD	Solaris	Dragonfly BSD	Haiku	Minix 3	Plan9	Nintendo DS	Sony PSP
`x86-32`	`cdecl`,`stdcall`,`fastcall`(ms/gnu),`thiscall`(ms/gnu)							[p9]	-	-
`x86-64`	`x64`	`sysv`				-	-	-	-	-
`ppc32`	-	`osx`	[`sysv`]	-	-	-	-	-	-	-
`arm32`	-	`aapcs`		-	-	-	-	-	`aapcs`	-

Table 5.13: Overview of supported platforms by `dyncallback`.

Table 5.14 gives an overview of the components of a "port" of `dyncallback`. Base components are listed first, followed by dependent components. The order gives an orientation for development of a port i.e. we suggest to first check for the existence of a suitable implementation of a "Memory allocator" before continuing with the development of a "Thunk" port. It is worth noting that unit tests are available for various components in order to support the porting process; an overview is given in Section 5.7.2.

Component	Language	API Header	Implementation files
Memory allocator	C	`dyncall_alloc_wx.h`	`dyncall_alloc_wx_<OS|API>.c`
Thunk	C, machine code	`dyncall_thunk.h`	`dyncall_thunk_<ARCH>.[c|h]`
Callback object	C	`dyncall_callback.h`	`dyncall_callback_<ARCH>.[c|h]`
Argument iterator	C	`dyncall_args.h`	`dyncall_args_<ARCH>.[c|h]`
Callback entry	Assembly	-	`dyncall_callback_<ARCH>.[S|s|asm]`

Table 5.14: Components of `dyncallback`.

Except for the memory allocator, which needs to be ported to a particular operating-system platform, the other components are processor-architecture specific (i.e. ISA, ABI and calling convention). We now discuss the components and their implementation in more details.

5.5.7.1 Memory Allocator

In order to "call" an object so that the program counter can jump to its address, the memory need to be marked as "executable". For security reasons current platforms typically forbid execution of code for certain default memory regions such as the stack and heap memory. Several CPUs incorporate a protection mechanism on their MMU which is advertised as a special protection bit, such as the NX bit (Never eXecute) by AMD, or XD bit (eXecute Diasabled) by AMD Intel, for `x86-64` and the XN bit (eXecute Never) for the `arm` architecture. Standard C functions for dynamic allocation of memory, such as `malloc`, allocate memory for data objects on the heap so that execution of code is forbidden. However, there are operating-system specific functions for allocating memory in which users can specify security and access features. The "Memory allocator" interface in `dyncallback` represents a portable abstraction layer for allocation of `Writable-eXecutable` memory. The C API interface is given below:

```
DCerror dcAllocWX(DCsize size, void** pp);
void dcFreeWX (void* p, DCsize size);
```

`dcAllocWX` allocates a memory block of `size` bytes. On success the address is written as a user-defined pointer variable, given by `pp`, and an error code of zero is returned. Otherwise an error is indicated by a return value less than zero. `dcFreeWX` is used to free the memory.

Currently two ports are included; both regarded together cover a broad spectrum of operating system platforms. One implementation uses the windows-specific memory allocator function `VirtualAlloc` and the other uses the POSIX function `mmap`. There is also a *dummy* implementation, based on `malloc`, for use on target platforms which allow execution of code in heap memory.

5.5.7.2 Anatomy of Thunks

Thunks are provided as a reusable component via a C API available from `dyncall_thunk.h`:

```
typedef struct {
  /* platform-specific implementation */
} DCThunk;
void dcbInitThunk(DCThunk* t, void (*e)() );
```

The API defines a C `struct` data object, named `DCThunk`, and a function `dcbInitThunk` for its initialization. Both are platform-specific, i.e. each port gives their definition and implementation.

Users can use the `DCThunk` structure as a header member field of their user-defined data structure but need to be aware that the definition and its size is platform-specific.

The function `dcbInitThunk` initializes objects with a thunk header by writing a machine-code sequence to the memory referenced by `t`. It is therefore important that thunk objects and derived user-defined objects are allocated in memory regions that are writable and executable such as provided by `dcAllocWX`.

The *thunk entry* routine, given by `e`, is executed in response to arbitrary function calls on a thunk object. Due to the special conditions on entry, i.e. the environment of the "call" is accessible via the *thunk register*, the routine is usually implemented in assembly by the user. Although the `dyncallback` framework uses one implementation for the delegation to a handler function the thunk API allows for other applications. For example, we plan to use thunks for dynamic inheritance and overloading of C++ classes by scripting languages.

An implementation of the thunk software framework comprises:

1. Selection of a *Thunk Register* or some other means (i.e. by pushing a value on the stack) to transfer an address value between thunk code and thunk entry routine without destroying the resources used by function calls.

2. Definition of a C API "data" structure, named `DCThunk`, which corresponds to the size of the machine code needed to implement the two basic tasks of a thunk, namely:

    - to `load` a self-reference address, given by `t`, to the common variable (determined in Step 1) and then

    - to `jump` to a *thunk entry function*, given by `e`.

3. Implementation of a code generator C API function, named `dcbInitThunk`, which initializes the data structure the above with the machine code.

Examples of thunk implementations, in terms of assembly code and corresponding memory block layout, C data structure and code generator for four architectures, are depicted in Listing 40. Note that the machine code in the C function was determined by development of prototypes in assembly code, translated and disassembled or outputted as hex dumps via image format tools, such as `objdump` (for ELF and PE) and `otool` (Mach-O).

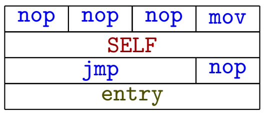
The selection of the *thunk register* depends on the architecture and the range of calling conventions on the platform to be supported. As a requirement the register needs to be volatile without taking the role of a parameter register. However, it may be a register reserved for return values since it is soley used for data transfer between thunk code and thunk entry. A list of architectures and suggested thunk registers is given in Table 5.15.

We give a brief summary of implementation methods and we outline a fallback alternative.

- The *thunk object address* is loaded in the thunk register

    - relative to the program counter using a `lea` (*load effective address*) instruction (`x86-64`) x86-32

```
nop
nop
nop
mov SELF, %eax
jmp 12(%eax)
nop
```

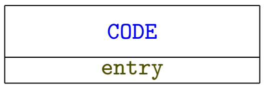


Thunk-Register: EAX

```
typedef struct {
  uint32_t mov;
  void     *SELF;
  uint32_t jmp;
  void    (*entry)();
} DCThunk;
void dcbInitThunk(DCThunk *t, void (*e)()) {
  t->mov   =0xB8909090;
  t->SELF =t;
  t->jmp   =0x900C60FF;
  t->entry=e;
}
```

x86-64

```
lea 0(%rip), %rax
jmp 16(%rax)
```

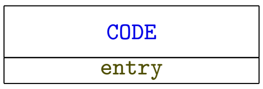


Thunk-Register: RAX

```
typedef struct {
  uint64_t code[2];
  void     (*entry)();
} DCThunk;
void dcbInitThunk(DCThunk* t, void (*e)()) {
  t->code[0]=
    0xffffffff9058d48ULL;
  t->code[1]=
    0x9090900000000325ULL;
  t->entry  =e;
}
```

arm32

```
sub %r12, %r15, 8
ldr %r15, %r15,-4
```

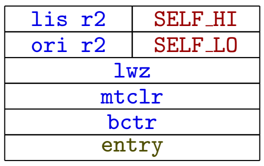


Thunk-Register: R12

```
typedef struct
{
  uint32_t code[2];
  void     (*entry)();
} DCThunk;
void dcbInitThunk(DCThunk* t, void (*e)() ) {
  t->code[0]=0xE24FC008;
  t->code[1]=0xE51FF004;
  t->entry  =e;
}
```

ppc32-osx

```
lis   %r2,HI
ori   %r2,%r2,LO
lwz   %r12,20(%r2)
mtclr %r12
bctr
```



Thunk-Register: R2

```
typedef struct {
  uint16_t lis, HI;
  uint16_t ori, LO;
  uint32_t code[3];
  void    (*entry)();
} T;
void dcbInitThunk(DCThunk *t, void (*e)() ) {
  t->lis=0x3C40;t->HI=hi16(t);
  t->ori=0x6042;t->LO=lo16(t);
  t->code[0]=0x81820014;
  t->code[1]=0x7D8903A6;
  t->code[2]=0x4E800420;
  t->entry=e;
}
```

Listing 40: Thunk Assembly Header, Code Block and C Implementation.

Architecture	Free register	Supported ABIs and Calling Conventions
`x86-32`	EAX	`cdecl`, `stdcall` and `thiscall`
`x86-64`	RAX	`sysv` and `x64`
`arm32`	R12	`apcs`, `linux-aapcs` and `armhf`
`ppc32`	R2	`osx`
`ppc32`	R12	`osx`, `sysv`
`mips32`	R1	`oabi`, `eabi`, `n32`, `n64`, `eabi`
`sparc`	G1	`v7`, `v9`

Table 5.15: Free registers suitable for passing a reference of a Thunk.

or move instruction with arithmetic adjustment (e.g. `sub thunk, pc, 8` for `arm32`[7]), or

– as an immediate constant, given by `p`, encoded in a `li` (*load immediate*) or `mov` (*move*) instruction (`x86-32` and `ppc32-osx`).

The *thunk entry code* address, given by `e`, is encoded as a data field in the code header below the `jump` instruction. The data field is addressed via an offset to the *thunk register* (`x86-32/64` and `ppc32-osx`) or the program counter (`arm32`) as the target of the `jump`.

- As a fallback solution: *Thunk object address* and *thunk entry code* are both encoded as immediate constants in `load` and `jump` instructions.

5.5.7.3 Anatomy of Callback objects

Callback objects consist of a thunk header followed by callback-specific data. The thunk entry routine "Callback Thunk Entry" gets access to the data part via the *thunk register* so that the handler function pointer and a user data pointer can be retrieved. The default structure is given below:

```
typedef struct {
  DCThunk             thunk;
  DCCallbackHandler*  handler;
  void*               userdata;
} DCCallback;
```

The API function `dcbNewCallback` is responsible for the creation of this object. A default implementation, which is used by most ports, is given below:

```
DCCallback* dcbNewCallback(const char* signature, DCCallbackHandler* handler, void* userdata) {
  int err;
  DCCallback* p = 0;
  err = dcAllocWX(sizeof(DCCallback), (void**)&p);
  if (!err) {
    dcbInitThunk(&p->thunk, &dcCallbackThunkEntry);
    p->handler  = handler;
    p->userdata = userdata;
  }
  return p;
}
```

[7] I.e. *thunk register* = *program counter* − 8 bytes; the offset is needed since the `sub` instruction is executed when the program counter is already incremented.

At first, memory is allocated by using `dcAllocWX`, and then the thunk header is initialized by a call to `dcbInitThunk`. Finally, callback-specific properties, such as the user-defined handler and data, are stored, and a pointer is returned.

5.5.7.4 Anatomy of Argument Iterators and Callback Thunk Entry

The definition of the `DCArgs` data structure and corresponding methods for iteration can be implemented in C while its allocation and initialization need to be done by the `dcCallbackThunk` routine written in assembly language.

The argument iterator is very similar to the Call VM of `dyncall`. It is a state machine that, via an iteration interface, updates variables and pointers in order to deliver type-specific values. However, the interface is plain functional and does not use a common object-oriented structure as in the case of Call VM; the front-end API functions (e.g. `dcbArg<TYPE>`) are implemented by a back-end port without an intermediate layer, such as a function table.

The callback thunk entry routine is written in assembly language and needs to perform the following tasks:

1. Allocation and initialization of a `DCArgs` object in the local stack frame. All parameter registers need to be saved to corresponding locations which need to match the structure of the C object.
2. Allocation of a `DCValue` object in the local stack frame.
3. Calling the handler `hf(p,a,r,ud)`, where `p` is the value of the *thunk register*, `a` and `r` are the addresses of `DCArgs` and `DCValue` on the local stack frame, and `h` and `ud` are read from `DCCallback` (accessible via the *thunk register*).
4. On return, the handler's return value indicates the return type. According to the type code the actual return value of the callback (stored in `DCValue`) is loaded to corresponding registers reserved for passing return values.
5. The local stack frame is removed. Depending on the calling convention the caller's stack frame also needs to be removed. Finally control flow is passed back to the callee.

5.5.7.5 Port `arm32-aapcs-softfloat`

The implementation of `DCArgs` and a suitable *callback thunk entry* represent the major parts of a port (aside from the thunk). As an example we discuss the `arm32-aapcs-softfloat` port.

On entry of a function call all arguments are placed in sequence from *left-to-right* in register memory of four registers for the first 16 bytes of argument data. Further data is passed via the stack in ascending order of memory addresses. We can implement a suitable argument iterator by means of a buffer for the register content and we use a counter for iteration which runs from zero to three. When the counter reaches index four the iteration is performed using a pointer, initially initialized to the top of the stack. The data structure of the argument iterator object is declared as follows:

```
typedef struct
{
  int  reg_data[4];
  int  reg_count;
  int* stack_ptr;
} DCArgs; /* for ARM */
```

The implementation of the argument iterator is given in Listing 41. Front-end API functions read a 4 or 8-byte quantity, perform type conversion and then pass back the result as return value. The three core functions for reading data update the counter and/or the pointer: `arm_word` is used for reading a 4-byte word, `arm_double` and `arm_longlong` are used for 8-byte quantities and return data as integer or floating-point values, respectively. In order to support the `eabi` ABI (e.g. `aapcs-linux`), `arm_double` and `arm_longlong` are slightly modified; the counter is adjusted to be the next even index or the stack pointer is adjusted to the next 8-byte boundary.

```
DClonglong  dcbArgLongLong  (DCArgs* p) { return arm_longlong(p);               }
DClong      dcbArgLong      (DCArgs* p) { return *(DClong*)arm_word(p);         }
DCint       dcbArgInt       (DCArgs* p) { return (DCint)   dcbArgLong(p);       }
DCchar      dcbArgChar      (DCArgs* p) { return (DCchar)  dcbArgLong(p);       }
DCshort     dcbArgShort     (DCArgs* p) { return (DCshort) dcbArgLong(p);       }
DCbool      dcbArgBool      (DCArgs* p) { return (dcbArgLong(p) == 0) ? 0 : 1;  }
DCuint      dcbArgUInt      (DCArgs* p) { return (DCuint)     dcbArgInt(p);     }
DCuchar     dcbArgUChar     (DCArgs* p) { return (DCuchar)    dcbArgChar(p);    }
DCushort    dcbArgUShort    (DCArgs* p) { return (DCushort)   dcbArgShort(p);   }
DCulong     dcbArgULong     (DCArgs* p) { return (DCulong)    dcbArgLong(p);    }
DCulonglong dcbArgULongLong (DCArgs* p) { return (DCulonglong)dcbArgLongLong(p); }
DCpointer   dcbArgPointer   (DCArgs* p) { return (DCpointer)  dcbArgLong(p);    }
DCdouble    dcbArgDouble    (DCArgs* p) { return arm_double(p);                 }
DCfloat     dcbArgFloat     (DCArgs* p) { return *(DCfloat*)  arm_word(p);      }
```

```
void* arm_word(DCArgs* args) {
  if(args->reg_count < 4)
    return &args->reg_data[args->reg_count++];
  else
    return (void*)args->stack_ptr++;
}
DCdouble arm_double(DCArgs* args) {
  union {
    DCdouble d;
    DClong   l[2];
  } d;
#if defined(DC__ABI_ARM_EABI)
  arm_align8(args);
#endif
  d.l[0] = *(DClong*)arm_word(args);
  d.l[1] = *(DClong*)arm_word(args);
  return d.d;
}
```

```
DClonglong arm_longlong(DCArgs* args) {
  union {
    DClonglong ll;
    DClong     l[2];
  } ll;
#if defined(DC__ABI_ARM_EABI)
  arm_align8(args);
#endif
  ll.l[0] = *(DClong*)arm_word(args);
  ll.l[1] = *(DClong*)arm_word(args);
  return ll.ll;
}
void  arm_align8(DCArgs* args) {
  if(args->reg_count < 4)
    args->reg_count = (args->reg_count+1)&~1;
  else if(args->stack_ptr & 4)
    ++args->stack_ptr;
}
```

Listing 41: Implementation of Argument Iterator methods for `arm`.

The callback thunk entry routine for `arm` is given in Listing 42.

Preserved registers, stack pointer (Register R13) and link register (Register R14) are saved on the stack (line 3). However, the stack pointer is not updated and still points to the parameter region of

```
 1 .code 32
 2 arm_callback_thunk_entry:
 3   stmdb %r13 ,{%r4-%r11,%r13,%r14} # save
 4   mov   %r11 ,%r13                 # r11 = sp
 5   sub   %r13 ,%r13,#40             # Skip save region
 6   mov   %r4  ,#0                   # r4  = 0 (counter)
 7   stmdb %r13!,{%r0-%r4,%r11}       # push DCArgs(r0-r3,0,sp)
 8   mov   %r0  ,%r12                 # r0  = Thunk/Callback Object
 9   mov   %r1  ,%r13                 # r1  = DCArgs*
10   sub   %r13 ,%r13,#8              # Alloc sizeof(DCValue) storage
11   mov   %r2  ,%r13                 # r2  = DCValue*
12   ldr   %r3  ,[%r12, #16]          # r3  = void*
13   ldr   %r4  ,[%r12, #12]          # r4  = handler*
14   mov   %r14 ,%r15                 # r14 = r15
15   bx    %r4                        # Branch with link register
16   ldmia %r13 ,{%r0,%r1}            # Load results into r0/r1
17   ldmdb %r11 ,{%r4-%r11,%r13,%r15} # Restore and return
```

Listing 42: Callback thunk entry routine for `arm`.

the callback call so that this value is saved to R11 (line 4) and then is finally adjusted by 40 bytes (line 5). The allocation of `DCArgs` object is then prepared. Register R4 is loaded with constant zero. Now a set of registers, including parameter registers of the callback R0 to R3 for `reg_data[4]`, R4 for `reg_count` and R11 for the `stack_ptr`, is saved to the stack in accordance to the data structure of `DCArgs` (line 7). As all parameter registers are saved, the function call to the handler is prepared. The thunk register (R12) is loaded in R0 as the first argument `DCCallback*` (line 8). The current stack pointer gives the reference of `DCArgs` and is loaded as second argument in R1 (line 9). `DCValue` is allocated on the stack by subtraction of `sizeof(DCValue)` and the result is loaded as third argument `DCValue*` in R2 (line 10-11). The `mUserdata` field of `DCCallback` is retrieved via thunk register (at offset 16) and loaded as fourth argument in R3 (line 12). The handler function (also retrieved via thunk register) is loaded in R4 (line 13) followed by a `arm-thumb` *interworking-compatible* function call (line 14-15). On function return the handler has written return values in `DCValue` object (referenced by R13). Since, by convention, return values are always passed via R0 and R1, regardless of the data type, the type code returned by the handler is ignored. Both registers are loaded by data of `DCValue` (line 16). The local stack frame gets cleaned up. But the saved link register value is restored in the program counter in order to return to the caller (line 17).

5.6 Dynamic Loading of Code

The `dynload` library offers a software abstraction layer for services typically provided by the dynamic linker and loader of an operating system.

The C API represents a thin wrapper to platform-specific APIs for loading of shared libraries and resolving of symbols. In addition the library was extended with advanced functions to enumerate symbols tables. It can be used as a portable foundation layer for handling shared libraries in dynamic FFIs as was illustrated in Section 4.2 using the example of R and `rdyncall`.

5.6.1 Interface

The API is accessible via inclusion of the C header file `dynload.h`. The interface consists of three functions, namely the loading of code, resolving of symbols to addresses and the unloading of code:

```
DLLib*  dlLoadLibrary (const char* libpath);
void*   dlFindSymbol  (DLLib* pLib, const char* name);
void    dlFreeLibrary (DLLib* pLib);
```

`dlLoadLibrary` opens a shared library object specified by the path `libpath`. On success the shared library is loaded to the running process and a non-`NULL` pointer is returned. The latter represents an abstract reference object (or *handle)* to the loaded code resource; it is passed as first parameter for the other two functions.

`dlFindSymbol` looks up a `name` in the symbol table for exported code and data objects and returns the loaded memory address or `NULL` if the symbol was not found.

`dlFreeLibrary` destroys the *handle*; if there are no other references the shared library is unloaded[8] from the running process.

In order to resolve symbols of the executable program a `NULL` pointer can be given as `libpath` and `dlLoadLibrary` returns a *handle* that gives a reference to the running process.

Note that `libpath` is passes to the dynamic loader as-is without any modifications for a portable naming scheme. We discussed different naming conventions for the path across operating-system platforms in Section 4.2.3. And we also presented a portable naming scheme for loading shared libraries based on `dynload`. See Section 4.2.5 for a details.

5.6.2 Enumeration of Symbols

Shared object and executable files usually contain a symbol table which the dynamic loader uses for resolving of symbolic names to addresses. An extension to the core interface for providing this service was contributed; the interface is given below:

[8]Usually the operating system uses a reference count for loaded code modules to determine when to unload the object but details of this behaviour are platform-specific.

```
DLSyms*     dlSymsInit    (const char* libPath);
const char* dlSymsName    (DLSyms* pSyms, int index);
int         dlSymsCount   (DLSyms* pSyms);
void        dlSymsCleanup (DLSyms* pSyms);
```

`dlSymInit` opens the symbol table of a shared library file specified by the path `libPath` and returns an abstract *handle* object which is passed as first parameter for the other functions. A list of *exported symbols* can be queried by a series of `dlSymsName` calls with an `index` run from 0 to *number of symbols* (exclusive). The latter is returned by `dlSymsCount`. `dlSymsCleanup` closes the symbol table again.

5.6.3 Implementation

`dynload` was ported to major operating-system platforms that use ELF, PE or Mach-O as binary image format. Since the core interface is a thin wrapper to C API functions we give the two major native APIs, namely for Microsoft Windows and POSIX; the latter is available on most Unix-based platforms.

```
void* dlopen  (const char* path, int mode);
void* dlsym   (void* handle, const char* symbol);
void  dlclose (void* handle);
```

POSIX C API

```
HMODULE LoadLibrary    (LPCTSTR lpFilename);
FARPROC GetProcAddress (HMODULE hModule, LPCSTR lpName);
BOOL    FreeLibrary    (HMODULE hModule);
```

Microsoft Windows C API

The load function `dlopen` of the POSIX API offers a second parameter `mode` which can be used to pass generic and operating-system specific details for the dynamic loading/linking process. An adoption of this parameter in `dynload`'s C interface for portable fine-tuning the loading process is planned.

In contrast to the three core functions, the enumeration of symbol tables is not commonly available via dynamic linker C APIs. However, we discovered that the handle `HMODULE`, returned by `LoadLibrary` on Windows platforms, is actually a pointer to the PE DLL file in memory so that it was feasible to implement a Windows port for the `dlSyms*` API by processing the binary data structure of PE; a port for Linux ELF platforms was also feasible using the returned pointer of `dlopen`. However, this does apply for all ELF platforms. The ELF port was modified in order to open ELF shared objects as ordinary data files for processing. The new ELF port, a Mach-O port and additions to the interface were contributed by the author of `bridj` (Chafik, 2011) - a middleware package for Java to connect with C/C++ components which uses `dynload` to *discover* API elements and `dyncall/dyncallback` for interoperability.

5.7 Software Tools for Building and Testing

In this section we give an overview of methods and tools that are used for building, testing and release-management of the `dyncall` sources. The source package is organized as a tree of sub-level projects as follows (miscellaneous files, such as `README.*` files, are omitted here):

`autovar/`	Common abstraction to predefined macros.
`buildsys/`	Build tool files.
`doc/`	Documentation sources.
`dyncall/`	Library `dyncall` sources.
`dyncallback/`	Library `dyncallback` sources.
`dynload/`	Library `dynload` sources.
`portasm/`	Common abstraction to assembly language dialects.
`test/`	Test units.

The source package contains three library projects (`dyn*` folders), common C macro utilities (`autovar` and `portasm`), unit tests (`test`), documentation (`doc`) and build tool files (`buildsys`).

5.7.1 Build systems

The source code of `dyncall` comprises platform-specific modules; the appropriate subset of code must be selected for building depending on the target platform. Numerous source packages of software that require compilation incorporate a configuration phase, which detects the target platform, compiler tools, size of data types and so on.., before the actual build phase. Typically, on Unix-based system, a "`configure`" shell script is offered in the top-level folder of a source package that needs to be executed as a first step, followed by executing the `make` build tool to start the build phase.

A collection of popular tools for management of configure/make build systems are available from the GNU project. The GNU `autoconf` tool is a shell code generator which outputs a `configure` script based on specifications written in the M4 macro language. GNU `automake` is used to specify the build process. A project tree in a source package is usually specified by `Makefile.am` files, which are translated to `GNU make`-compatible `Makefile`s when running the auto-generated `configure` script.

Essentially, some kind of "configuration" is required in the `dyncall` package for the identification of the target platform and, based on that, the selection of platform-specific code for building. However, the GNU build system is rather complex and we found it hard to control it in detail; the generated shell code is large (even with default `autoconf` configurations) and when it runs a long list of tests is executed per default. Besides the popular Linux/GNU platform, `dyncall` also supports further platforms, such as Windows and BSD systems, which offer different *default* tools; those are not supported by the GNU build tools so that further methods for configuration and building software were considered:

	configure	`make` tool					out-of-source
Approach	phase	GNU	BSD	Sun	nmake	mk	builds
`make`-specific `Makefiles`	yes	X	X	-	X	X	(GNU)
common `Makefile.embedded`	no	X	X	X	-	-	-
common `Makefile.generic`	yes	X	X	X	-	-	yes
`CMake`	yes	X	X	X	X	-	yes
Visual Studio project files							

We incorporated three different methods for building the software (depicted in the first three rows of the table above) in order to support various `make`-derivates, such as `GNU make` (Stallman and McGrath, 2002), `BSD make`, `Sun make`, Microsoft's `nmake` and Plan9's `mk`. Support for building `dyncall` with `CMake` (Martin et al., 2003) was also added; this is a meta build system and supports a large number of other build systems, including a number of `make` tools.

5.7.1.1 Make-specific files with a common configuration

One goal was to incorporate methods for building the sources with default tools available on supported platforms. Over time three configuration scripts were added, namely a Unix `configure` shell script, a Windows `configure.bat` batch file and a Plan 9 `configure.rc` script. The scripts identify the target platform, architecture and default compiler/assembler tools by platform-specific means. Identified facts about the target environment are saved as properties in a file, named `ConfigVars`, using a *common* syntax format as depicted in Listing 43.

```
#auto-generated by dyncall/configure
CONFIG_PACKAGE=dyncall
CONFIG_HOST=darwin
CONFIG_OS=darwin
CONFIG_ARCH=x64
CONFIG_TOOL=gcc
CONFIG_ASM=as
CONFIG_CONFIG=release
CONFIG_PREFIX=/usr/local
```

```
ifdef BUILD_ARCH_ppc32
MODS += dyncall_callvm_ppc32
MODS += dyncall_ppc32
endif
ifdef BUILD_ARCH_mips32
MODS += dyncall_callvm_mips32
MODS += dyncall_mips32
endif
```

Listing 43: Common configuration file `ConfigVars` (*left*) and an excerpt of file `dyncall/GNUmakefile` with conditional blocks to select platform-specific sources (*deprecated* as of `dyncall` Version 0.4).

Configuration properties for building the software are specified as a list of key/value pairs. Note that the syntax is simple so that this file can be included by various `make` tools and the properties become `make` variables. The selection of platform-specific code modules was specified by means of conditional blocks offered by `make` tools. However, the syntax of conditional blocks is `make`-tool specific. As a consequence several make systems were integrated by using separate trees of `make` files (e.g. `GNUmakefile`, `BSDmakefile` and `Nmakefile`) according to the projects sub-folder structure of the source package. A sample of conditional blocks for the selection of the Call VM and call kernel of the project file `dyncall/GNUmakefile` is given in the right panel of Listing 43.

5.7.1.2 Predefined macros

Properties of the target platform are also provided by C compilers via *predefined macros*, which can be evaluated during preprocessing in C via standard `#if`, `#elif` and `#else` conditional blocks in order to enable/disable code sections, or to include platform-specific files.

However, names of predefined macros that indicate a specific target platform are specific to the C compiler. In order to use a common naming scheme in sources a thin abstraction layer to compiler-specific names for similar facts and properties is implemented in the file `dyncall/dyncall_macros.h`:

```
#if defined(_M_X64_) || defined(_M_AMD64) || defined(__amd64__) || defined(__amd64) || defined(__x86_64__)
# define DC__Arch_AMD64
#elif defined(__arm__)
# define DC__Arch_ARM
#elif /* [...] */
```

A common front-end C module file was added which includes a particular platform-specific C module file based on the definition of a macro. For example, the selection of the platform-specific Call VM code module is managed by a front-end C module `dyncall/dyncall_callvm.c`:

```
#if defined(DC__Arch_Intel_x86)
#  include "dyncall_callvm_x86.c"
#elif defined(DC__Arch_AMD64)
#  include "dyncall_callvm_x64.c"
#elif defined(DC__Arch_PowerPC)
#  include "dyncall_callvm_ppc32.c"
#elif defined(DC__Arch_PPC64)
#  include "dyncall_callvm_ppc64.c"
#elfi /* [...] */
```

Major Unix-based compiler tools, such as `gcc`, support *preprocessing of assembly language files* with the filename extension "`.S`" (in contrast to the default "`.s`" file extension) so that the same method was feasible for the assembly language sources in `dyncall`. By using this method we simplified the specification of `make` files in `dyncall` significantly.

5.7.1.3 Plain Makefiles

A set of files, named `Makefile.embedded`, was added as a new alternative for building the sources with `make` tools. However, the syntax used in these files was carefully chosen to be compatible with `GNU make`, `BSD make` and `Sun make`. See Listing 44 for an example.

This build method omits the `configure` script due to the use of front-end modules and predefined macros for C and assembly languages. Note that `configure` scripts are also used as command-line interface by users to specify build and installation settings, for example:

```
./configure --prefix=/my/install/dir CFLAGS=-m64
```

However, `make` tools also support such an interface using shell arguments or environment variables as shown by various examples in Listing 45. While users need to be aware of common variables for controlling a build, such as the `CFLAGS` variable to specify C compilation settings, the settings can be

```
LIBNAME = dyncall
OBJS = dyncall_vector.o dyncall_api.o dyncall_callvm.o dyncall_callvm_base.o dyncall_call.o
HEADERS = dyncall_macros.h dyncall_config.h dyncall_types.h dyncall.h
LIB = lib${LIBNAME}_s.a
.PHONY: all clean install
all: ${LIB}
${LIB}: ${OBJS}
        ${AR} ${ARFLAGS} ${LIB} ${OBJS}
clean:
        rm -f ${OBJS} ${LIB}
install: all
        mkdir -p ${PREFIX}/lib
        mkdir -p ${PREFIX}/include
        cp ${LIB} ${PREFIX}/lib
        cp ${HEADERS} ${PREFIX}/include
```

Listing 44: Example of a portable (GNU/BSD/SUN) plain makefile in `dyncall`.

```
$ make PREFIX=/tmp install                          # Install in /tmp directory
$ CFLAGS=-m64 make                                  # Build 64-bit version e.g. on sparc
$ PREFIX=/my/install/dir make CFLAGS=-m64 all install  # Build 64-bit version, install in /my/in[..]
$ make test                                         # Build all tests
$ CFLAGS=-g make                                    # Build 'debug' version
$ make CFLAGS=-O2                                   # Build 'release' version
$ make CC=clang                                     # Use clang compiler
$ make CFLAGS="-arch i386 -arch ppc"                # Build Mac OS X universal binary
```

Listing 45: Using `make` with user settings.

propagated from top-level (or from where the user executes `make`) to all sub-level directories; `dyncall` contains three library projects and 17 test projects. Note that the above commands can be executed from the top-level folder; all user settings (e.g. `CFLAGS=-m64`) are passed to sub-level directions.

The `Makefile.embedded` build system was successfully used for *embedding* the `dyncall` sources in the R `rdyncall` source package at `src/dyncall`. It was feasible to link the build process of `rdyncall` with that of `dyncall` by using standard R package make files, namely `src/Makevars` (depicted in Listing 46).

```
PKG_CPPFLAGS=-Idyncall/dyncall -Idyncall/dynload -Idyncall/dyncallback
PKG_LIBS=dyncall/dyncall/libdyncall_s.a \
         dyncall/dynload/libdynload_s.a \
         dyncall/dyncallback/libdyncallback_s.a

.PHONY: all mylibs

all: $(SHLIB)
$(SHLIB): mylibs

mylibs:
        (cd dyncall ; CC="$(CC)" AR="$(AR)" CFLAGS="-fPIC $(CFLAGS)" \
                      CPPFLAGS="-DNDEBUG ${CPPFLAGS}" make -f Makefile.embedded)
```

Listing 46: `Makevars` file of `rdyncall` to build the embedded `dyncall` sources.

While the R package build system offers support for running `configure` scripts, the delegation of a `make` shell call to another directory was less complex and our solution turns out to be portable as well.

Furthermore, R packages which use the default `Makevar` files (instead of `Makefile` and `configure` scripts) have various advantages, such as building R packages on Mac OS X per default as universal binaries.

5.7.1.4 Portable Assembler Sources

The syntax of assembly languages is naturally *specific* to the processor architecture. The code for call kernels of the x86-32 architecture can be *shared* across various operating systems. However, the syntax of assembly languages for the same processor architecture is often also differs for assembler tools on different operating systems.

Two syntax types are common on the `x86` architecture family, namely AT&T and Intel; these differ significantly (e.g. the order of source and destination operands of instructions is swapped). `MASM` (Windows) uses Intel syntax, others on Unix (GNU assembler `gas` and Sun's `fbe`) use AT&T syntax per default. Thus, the same code needed to be reimplemented repeatedly.

```
#include "../portasm/portasm-x86.S"
BEGIN_ASM


GLOBAL(dcCall_x86_cdecl)
BEGIN_PROC(dcCall_x86_cdecl)
        PUSH(EBP)
        MOVL(ESP,EBP)
        PUSH(ESI)
        PUSH(EDI)
        MOVL(DWORD(EBP,12),ESI)
        MOVL(DWORD(EBP,16),ECX)
        ADDL(LIT(15),ECX)
        ANDL(LIT(-16),ECX)
        MOVL(ECX,DWORD(EBP,16))
        SUBL(ECX,ESP)
        MOVL(ESP,EDI)
        REP(MOVSB)
        CALL_DWORD(EBP,8)
        ADDL(DWORD(EBP,16),ESP)
        POP(EDI)
        POP(ESI)
        MOVL(EBP,ESP)
        POP(EBP)
        RET()
END_PROC(dcCall_x86_cdecl)
END_ASM
```

```
.386
.MODEL FLAT
.CODE

_dcCall_x86_cdecl PROC
OPTION PROLOGUE:NONE, EPILOGUE:NONE
 push EBP
 mov EBP,ESP
 push ESI
 push EDI
 mov ESI,dword ptr [EBP+12]
 mov ECX,dword ptr [EBP+16]
 add ECX,15
 and ECX,-16
 mov dword ptr [EBP+16],ECX
 sub ESP,ECX
 mov EDI,ESP
 rep movsb
 call dword ptr [EBP+8]
 add ESP,dword ptr [EBP+16]
 pop EDI
 pop ESI
 mov ESP,EBP
 pop EBP
 ret
_dcCall_x86_cdecl ENDP
END
```

```
 .text



.globl _dcCall_x86_cdecl
_dcCall_x86_cdecl:
 pushl %ebp
 movl %esp,%ebp
 pushl %esi
 pushl %edi
 movl 12(%ebp),%esi
 movl 16(%ebp),%ecx
 addl $15,%ecx
 andl $-16,%ecx
 movl %ecx,16(%ebp)
 subl %ecx,%esp
 movl %esp,%edi
 rep movsb
 call *8(%ebp)
 addl 16(%ebp),%esp
 popl %edi
 popl %esi
 movl %ebp,%esp
 popl %ebp
 ret
```

Listing 47: Sources written in `portasm` (*left*) and translated to `MASM` for Microsoft platforms (*center*) and `gas` for others (*right*).

A small experimental software framework, named `portasm`, was initiated for writing processor-specific but tool-portable assembly sources. Our aim is to consign the maintence of the assembly language sources exclusively to a single *shared* source code, which is translated to various assembler tool syntaxes via the C preprocessor. Effectively, `portasm` code is written by means of preprocessor macros. This framework requires only a C preprocessor and has no further dependencies.

Consider, for example, the `x86-32-cdecl` call kernel given in Listing 47, implemented in `portasm` (left) and translated to `MASM` (middle) and `gas` (right). `portasm` source files are written with the extension "`*.S`"; at first, an architecture-specific header file from the `portasm` framework is included. GNU and Sun compilers can process these files directly as preprocessor assembly language files. In order to support `MASM` the `portasm` source is processed by a standard C preprocessor, where the macro `GEN_MASM` is predefined (via command-line parameter e.g. `cc -E -DGEN_MASK`), which activates an appropriate driver of macro definitions, and the output of the sources is produced in `MASM` format. Currently `portasm` is used in `dyncall` for `x86-32`, `x86-64`, `ppc32` and `arm32` assembly sources.

5.7.2 Software testing

A collection of projects for software testing is made available below the `test` folder of the source distribution; an overview is given in Table 5.16.

Name	Test purpose
	Unit tests for `dyncall`
`call_suite`	Lua-based code generator for random/user-defined test cases.
`plain`	Plain C function calls.
`plain_c++`	Plain C++ member function calls.
`suite`	Python-based code generator for ordered permutation of function types.
`suite_floats`	Based on `suite` testing only floating-point function types.
`suite_x86win32fast`	Based on `suite` for `fastcall` calling convention.
`suite_x86win32std`	Based on `suite` for `stdcall` calling convention.
`suite2`	Python-based code generator for random/user-defined function types.
`suite2_x86win32fast`	Based on `suite2` for `fastcall` calling convention.
`suite2_x86win32std`	Based on `suite2` for `stdcall` calling convention.
`suite3`	Based on `suite2` for a subset of argument types.
`ellipsis`	Based on `suite2` for variadic function call tests.
`callf`	Plain test for formatted call interface.
`syscall`	Plain test for ystem calls.
	Unit tests for `dyncallback`
`callback_suite`	Lua-based code generator for random/user-defined test cases.
`callback_plain`	Plain C callback tests.
`malloc_wx`	Unit test for writable/executable memory allocator.
`thunk`	Unit test for thunks.
	Unit tests for `dynload`
`nm`	Enumerating symbol tables in shared libraries.
`resolve_self`	Unit test for resolving symbols of an executable program's symbol table..

Table 5.16: Overview of *DynCall* unit tests.

Some projects consist of simple unit tests for a specific feature, such as `malloc_wx` for `dcAllocWX`, while others represent test suites with advanced features to support testing and debugging of implementation code.

5.7.2.1 Code generators for test cases

It frequently occurs during port development or debugging that some very specific test cases have to be checked. For example, according to the `ppc32-sysv` calling convention, the first eight 32/64-bit floating-point values and the first eight 32-bit integer are passed via registers before using the stack. Suppose we want to check the correct passing of `double` and `float` values via registers and on stack. A number of test cases would be required, for example, eight `double` values (to fill registers) and then a `float` (make the stack 4-byte aligned) followed by a `double` to check the alignment of `double` values on stack.

Several test suites were developed using code generators for automation of test case generation and execution based on simple specification files. The test suite `call_suite` was designed for validating ports of the `dyncall` library. A test case validates the correct passing of argument values and the correct reception of return values for a specific function type. A set of test cases is specified in a text file named `cases.txt`. Each line specifies one test case encoded as character-based type signature; the first character specifies the return type followed by a sequence of characters that specify the argument types, for the example the sequence `vddddddddfd`. As an extension to the example, we also check all shortened subset sequences, beginning with an empty sequence so that we *design* a set of test cases in `cases.txt` as depicted in Listing 48. The command line and output of the generator are given on the right of the listing. At first the generator `mk-cases.lua` processes input the `cases.txt` file as input and the output is saved its as a C header file `cases.h`; this step can also be executed via `make config`. Then the test program is build via `make clean all`.

File: `cases.txt`
```
v
vd
vdd
vddd
vdddd
vddddd
vdddddd
vddddddd
vdddddddd
vddddddddf
vddddddddfd
```

```
$ lua ./mk-cases.lua <cases.txt >cases.h
$ ./make clean all
$ ./call_suite
 0:v:1
 1:vd:1
 2:vdd:1
 3:vddd:1
 4:vdddd:1
 5:vddddd:1
 6:vdddddd:1
 7:vddddddd:1
 8:vdddddddd:1
 9:vddddddddf:1
10:vddddddddfd:1
result: call_suite: 1
```

Listing 48: Example of user-defined test cases in `call_suite`.

Finally the test cases can be executed and a test report is generated. Each line of output gives the number of the test case (which starts counting at 0), followed by its signature and terminated by the test result. A result code of “1” indicates a success, otherwise “0” is given (as well as extended error diagnostics such as the index of the argument which failed). The last line of output gives a global result code: “1” if *all* test cases were successful, and otherwise “0”.

The test suite framework supports two basic operating modes: *design* and *random*. The above example was executed in *design* mode where users specify test cases manually for debugging sessions. The *random* mode was designed for validation of implementations by using random test case functions; an illustration is given in Listing 49.

config.lua

```
minargs   = 1
maxargs   = 8
ncases    = 4
types     = "csijlpfd"
```

→

rand-sig.lua

```
#!/usr/bin/env lua
-- rand-sig.lua
require"config"
rtypes   = "v"..types
math.randomseed(2342)
local sigs = { }
local id
for i = 1, ncases do
  id = math.random(#rtypes)
  local nargs = math.random(minargs,maxargs)
  local sig   = { rtypes:sub(id,id)}
  for j = 1, nargs do
    id = math.random(#types)
    sig[#sig+1] = types:sub(id,id)
  end
  io.write(table.concat(sig))
  io.write("\n")
end
```

→

cases.txt

```
vspfsdd
vjc
dlf
vlfjjcpcj
```

↓

mk-cases.lua

(See Listing 50)

↓

cases.h

```
/* 1:vspfsdd */ v f1(s a1,p a2,f a3,s a4,d a5,d a6){fid=1;V_s[1]=a1;V_p[2]=a2;V_f[3]=a3;V_s[4]=a4;V_d[5]=a5;V_...
/* 2:vjc */ v f2(j a1,c a2){fid=2;V_j[1]=a1;V_c[2]=a2;ret_v(2)}
/* 3:dlf */ d f3(l a1,f a2){fid=3;V_l[1]=a1;V_f[2]=a2;ret_d(2)}
/* 4:vlfjjcpcj */ v f4(l a1,f a2,j a3,j a4,c a5,p a6,c a7,j a8){fid=4;V_l[1]=a1;V_f[2]=a2;V_j[3]=a3;V_j[4]=a4;...
funptr G_funtab[] = {
        (funptr)&f1,
        (funptr)&f2,
        (funptr)&f3,
        (funptr)&f4,
};
char const * G_sigtab[] = {
        "vspfsdd",
        "vjc",
        "dlf",
        "vlfjjcpcj",
};
int G_maxargs = 8;
```

Listing 49: Random test case generator using Lua.

Base parameters are specified in the file config.lua. Then the script `rand-sig.lua` is executed which reads configuration properties and generates a number of random signatures, given by `ncases`. Other properties give details for preparing random sequences such as the upper and lower limits for the number of arguments, given by `minargs` and `maxargs`, and the base set of argument types, given by `types`. A set of random function type signatures is created in file `cases.txt` by executing `make config-random`. Subsequent steps are identical to those of *design* mode (e.g. `make clean all && ./call_suite`).

We discuss now the generic design of the test case in more detail. In preparation for running a single test case, a global memory buffer is cleared to zero. The test is executed by calling the test case function via `dyncall` and passing a sequence of *reference values* as arguments. The callee stores all received argument values in the global memory buffer and returns a reference value based on its function type. Then it is checked that output values stored in global memory and the return value are

```lua
require"math"
local max = math.max
local maxargs = 0

function trim(l)
  return l:gsub("^%s+",""):gsub("%s+$","")
end

function mkcase(id,sig)
  local sig = trim(sig)
  local h = {
    "/* ",id,":",sig," */ ",sig:sub(1,1),
    " f", id,"(",""
  }
  local t = { "fid=",id,";" }
  local pos = 0
  maxargs = max(maxargs, #sig-1)
  for i = 2, #sig do
    pos = tostring(i-1)
    local name = "a"..pos
    local ch   = sig:sub(i,i)

    h[#h+1] = ch;h[#h+1] = " ";h[#h+1] = name
    h[#h+1] = ","

    t[#t+1] = "V_";t[#t+1] = ch;
    t[#t+1] = "[";t[#t+1] = pos;t[#t+1] = "]"
    t[#t+1] = "=";t[#t+1] = name;t[#t+1] = ";"
  end
  h[#h] = "){"
  t[#t+1] = "ret_";t[#t+1] = sig:sub(1,1)
  t[#t+1] = "(";t[#t+1] = pos;t[#t+1] = ")"
  t[#t+1] = "}\n"
  return table.concat(h,"")..table.concat(t,"")
end
```

```lua
function mkfuntab(n)
  local s = { "funptr G_funtab[] = {\n"}
  for i = 1, n do
    s[#s+1] = "\t(funptr)&f"..i..",\n"
  end
  s[#s+1] = "};\n"
  return table.concat(s,"")
end

function mksigtab(sigs)
  local s = { "char const * G_sigtab[] = {\n"}
  for k,v in pairs(sigs) do
    s[#s+1] = '\t"'
    s[#s+1] = v
    s[#s+1] = '",\n'
  end
  s[#s+1] = "};\n"
  return table.concat(s,"")
end

function mkall()
  local lineno = 1
  local sigtab = { }
  for line in io.lines() do
    local sig = trim(line)
    io.write(mkcase(lineno,sig))
    sigtab[#sigtab+1] = sig
    lineno = lineno + 1
  end
  io.write(mkfuntab(lineno-1))
  io.write(mksigtab(sigtab))
  io.write("int G_maxargs = "..maxargs..";\n")
end

mkall()
```

Listing 50: Test case generator in Lua.

equal to corresponding reference values. Reference values are retrieved from arrays of constant values (one array for each supported data type). The arrays (one for each data type) are initialized with *distinct* pseudo-random numbers. The reference value for a specific argument type in the sequence is determined by the type and the position in the sequence. The argument type selects the array and the positional index specifies the index of the actual value to use. The reference value of a return value uses the number of arguments as index for selecting the value.

The first test suites in `dyncall` (e.g. `suite`) were written in C++ and Python. In more recent frameworks, such as `call_suite`, we use now C and Lua. First, Lua is very convenient for shell scripting and text processing. Furthermore, Lua is small, runs fast and has very low dependencies; it requires an ANSI C compiler. If Lua is not available for the target platform, a script is available for bootstrapping Lua from source; this downloads sources and builds the Lua interpreter in very little time (ten seconds to a minute). This tool and others (e.g. `callback_suite` for testing `dyncallback`) were very *valuable* during porting. We use them to run large numbers of test cases with tens to hundreds of arguments to validate all ports before software releases.

5.7.2.2 Testing Environment

Stability tests across a large range of operating system/processor architecture combinations are tedious to set up and run. `dynos` is a testing environment that was developed for automation of building, testing and packaging of releases.

"The dynos test project(Philipp, 2011) is a FreeBSD based collection of shell scripts that run dyncall builds and regression tests on different OS and toolchain deployments. This is done natively for the host OS, in OS images run by various emulators (such as QEMU and GXemul), on more or less directly attached devices (e.g. Nintendo DS), or over the network. The platforms are set up in a way that allows fully automated startups, tests and shutdowns." (Philipp, 2011)

It follows a list of supported platforms as of May 2012:

```
Arch Linux 2011.08.19 / x64
Arch Linux 2011.08.19 / x86
Debian GNU/Linux 4.1.1-21 / PPC
Debian GNU/kFreeBSD 6.0 / x64
Debian GNU/kFreeBSD 6.0 / x86
DragonFlyBSD 2.10.1 / x64
DragonFlyBSD 2.0.0 / x86
Haiku r1 alpha2 r38811 / x86
NetBSD 4.0.1 / cats (arm)
NetBSD 5.0.2 / pmax (mips)
OpenBSD 4.0 / x64
OpenBSD 4.0 / x86
Plan9 4th edition / x86
Windows XP / x64
Windows XP / x86
Darwin 8.0.1 / x86
Desmume 0.8 / Nintedo DS (arm) - experimental
NetBSD 4.0.1 / macppc
Nexenta (OpenSolaris) 1.0.1-b85 / x64
Nexenta (OpenSolaris) 1.0.1-b85 / x86
OpenBSD 4.4 / mvme88k - experimental
ReactOS 0.3.11 / x86 - experimental
```

5.8 Summary

This chapter covers the design of a *Generic Dynamic FFI* software package, named `dyncall`, and its implementation on the basis of five processor architecture families, namely `x86`, `arm`, `powerpc`, `mips` and `sparc`. We introduced the topic with an overview of the anatomy of machine-level function calls. We outlined the environment of 'machine-code' execution before we discussed the ISA and ABI, and the roles of registers and the call stack in a calling sequence of calling conventions.

An overview on the processor architecture families was given. Each architecture was introduced with a brief account of its history and current developments, followed by an outline of available binary interfaces and their particularities. We illustrated differences of stack frame structures and utilization of CPU, FPU and SIMD register in calling conventions. We closed this overview with an empirical analysis of machine-level representations of C data types at machine level. We motivated the use of low-level techniques for generic solutions due to limitations of C using the static FFI of R as an example.

The design of the `dyncall` library was discussed in detail. We presented a C API for dynamic function calls founded on a portable implementation architecture which consists of a state machine (Call VM) in C and a generic caller code (call kernel) in assembly language. We emphasized the portability of the architecture design by numerous examples of available ports and detailed descriptions of their implementation in C and assembly, thereby giving a detailed picture of the variety of binary interfaces and calling convention standards, and indicated similarities and differences.

We discussed dynamic handling of callbacks on the callee side. Our objective was to enable implementation of C callback functions by means of a scripting language. We suggested a generic solution for versatile use within various C APIs based on wrapping of callable objects of the scripting language as plain C function pointers and presented the `dyncallback` software framework for implementing such *wrapping* services in scripting languages.

As a foundation we designed a hybrid code and data object type, which works as a proxy for calls made from compiled code. Based on this object we developed a method for delegating callbacks to a common handler function which handles the call by means of a different language. The software framework makes use of a small machine-code generator. We suggested a method for developing such generators, based on prototyping (2-4 lines of assembly code) and subsequent code analysis for an implementation as a generator in C. Included here was a discussion on the implementation of such codes in accordance to the software framework, and we gave implementations for some processor architectures.

We closed this chapter with a brief overview on the interface of `dynload` and a discussion on software tools for building and testing.

We discussed examples of applications with an emphasis on R; in detail we described the implementations of the core components of `rdyncall` in C, founded on the C interface of `dyncall`. However, the methods for implementation of dynamic FFIs are transferable to other interpreter software that offer a C extension and embedded interface i.e. virtually all interpreters.

Chapter 6

Conclusion

We close with a summary of contributions and related work, followed by a conclusion and an outline of ongoing developments.

6.1 Summary

Language bindings are typically implemented by means of adapter modules written in C on a per language-and-library basis which needs to be compiled for each platform. We presented a middleware architecture for scripting languages for an extension to *connect* with compiled C libraries dynamically to give users scripting access to C APIs.

We gave a proof-of-concept by example of an implementation for R, contributed as the package `rdyncall` on CRAN, which offers dynamic access to the C API of standard library families such as `OpenGL` and `SDL`. The architecture fosters a rapid application development model. This was illustrated by examples of *portable* R applications code for interactive real-time graphics applications that run across platforms, including Microsoft Windows, Mac OS X, Linux, BSD derivates and Solaris.

We discussed the design and implementation of the architecture in terms of three layers:

- The top-level "abstraction" layer provides an intuitive *text-based* language- and platform-neutral type library format for C API specifications, designed to *automate wrapper creation.*
- The middle-level layer is "implemented" for a specific scripting language to provide *automation* of bindings to C APIs via a *dynamic* FFI using information of the top-level layer. The dynamic FFI is founded on the bottom-level layer.
- The bottom-level "abstraction" layer offers *portable* C interfaces for *dynamic* control of function calls and callback handling using *generic* implementation at machine level ported on various processor architectures.

We give a brief overview of contributions and findings of this thesis:

- The design and implementation of a generic dynamic FFI, contributed as `dyncall`, with ports to `x86-32/64`, `arm32`, `ppc32`, `mips32/64` and `sparc32/64` (Chapter 5).
- The design and implementation of a dynamic FFI for R, contributed as `rdyncall`, (Chapter 4).
- A method for a common naming scheme to load shared libraries via the dynamic linkers of various operating systems (Section 4.2.5).
- Specifications for a text-based type library format for C APIs (Section 3.7).
- An application of `GCC_XML` and XML tools for the development of a parser for C/C++ header files (Section 3.8).
- Various R bindings to C APIs, such as `OpenGL` and `SDL` (Section 4.7 and Table 4.10).
- A software framework for tool-portable abstraction to assembly language (Section 5.7.2).
- An example for portable `make` files for BSD, GNU and Sun `make` (Section 5.7.1.4).
- The design of a test-case code generator for FFIs using the example of the `call_suite` of `dyncall` (Section 5.7.2).

Our contributions exhibit a relatively low binary memory footprint due to the generic implementation at machine level. The code is small in comparison to other packages. Table 6.1 compares binary size and the number of lines of source code of `rdyncall` and `dyncall` libraries to `rgl`. Note that, in the 64 kb binary file of `rdyncall`, all three C library binary files are statically linked. However, whereas `rgl` contains 20 times more code and comprises an abstraction layer to windowing systems and a C++ scene graph with a small interface of 122 functions, `rdyncall` gives virtual access to a dozent C APIs i.e. hundreds of functions.

	Binary	Sources (Lines of code)			
	size	R	C		Assembly
Package			Sources	Headers	
R package `rgl`	1.3 mb	7020	21484	-	-
R package `rdyncall`	64 kb	682	1499	-	-
C library `dyncall`	25 kb	-	5805	737	3140
C library `dyncallback`	7.5 kb	-	2336	391	1854
C library `dynload`	4.4 kb	-	807	65	-

Table 6.1: Some metrics of software packages (platform `x86-64-osx`).

6.2 Related work

The origins of automated bindings come from the `SWIG` software development tool (The SWIG Developers, 2010) supports a large number of languages but uses compiled code. Kloss (2008) describes the `ctypeslib` python package, an extension to `ctypes` (Heller, 2011) that brings automation of dynamic bindings to C APIs - based on XML output of `GCC_XML` and transformation to python code via `pyXML`. Our approach extends the idea but uses a language-neutral encoding for type information. Our toolchain for parsing C header files is also based on `GCC_XML` but applies XML/XSLT processing tools, such as `libxslt`, for transformation of XML output to type signatures encoded in plain text. The FFI library of LuaJIT (Pall, 2012), a JIT (Just-In-Time) compiler-based version of Lua, also offers automation of bindings creation; it uses a built-in C parser.

Several interpreter-based languages offer a dynamic FFI, such as `ctypes` (Heller, 2011) for Python, `alien` (Mascarenhas, 2009) for Lua, `RFFI` (Lang, 2011) for R, `CFFI` (Bielman, 2010) for Common LISP and the `FFI` module for Perl (Paul Moore, 2008) and Ruby (Meissner, 2011). Most FFIs use a language-specific interface, except `Perl FFI` that suggested a text-based interface policy; we followed this approach in this work. These all facilitate similar services, such as foreign function calls and handling of foreign data. With the exception of `Rffi`, these also support wrapping of scripting functions as C callbacks. In most cases the type information is specified as a sequence of statements in the grammar of the dynamic language.

Specific alternatives to `dyncall` include `libffi` (Green and et al, 2011) and `ffcall` (Haible, 2004). They offer a *data-driven* C interface and, in particular, `libffi` has support for a large number of current and older platforms. The `dyncall` interface offers a more *functional* C interface inspired by the OpenGL API. The `C/Invoke` library (Weisser, 2007) also offers a similar FFI facility with bindings to Lua, Java and Kite.

The `csound` (Brown, 2012) R package on CRAN contributes R bindings to the C API of `Csound`, a software sound synthesizer, and uses the binding facilities of `rdyncall`. Package `cart` (Gastner and Newman., 2013) uses diffusion-based cartograms with R based on bindings to the FFT (Fast Fourier Transformation) library `libfftw3` via `rdyncall`.

The following software projects have already incorporated `dyncall`: A Java to C++ middleware, named `BridJ` (Chafik, 2011), a lightweight Perl 6-like environment for virtual machines, named `nqp` (Foundation, 2013), an embedded scripting language, named `pawn` (CompuPhase, 2013), an extension for native calls to Rakudo (a variant of Perl 6), named `Zavolaj!` (Worthington, 2013), a game server engine plugin, named `SPE` (Developers, 2013), an application server, named `GTK Server` (van Eerten, 2012), a C/C++ type and value introspection library with Ruby bindings, named `Typelib` (Joyeux, 2009) and a JavaScript shell, named `jsh` (Taylor, 2013).

6.3 Conclusion and Future Direction

In this thesis we have presented in this thesis the architecture of a *Dynamic Bindings Model*, with a focus on the technical feasibility on a variety of platforms. Our initial assumptions of the potential for Generic Dynamic FFIs were confirmed by the feedback from `dyncall` users and its apllication in a variety of other projects. Since little has been documented on the design and implementation of such FFIs we approached the task of developing our Generic Dynamic FFI, `dyncall` "de novo". We covered a wide range of architectures in order to verify the feasibility. In particular, the development of test case code generators and testing enviroments was a *major* improvement for porting and verification of code in `dyncall`. We have focused primarily on the mappings of C to machine interfaces and a subset of the data types; this simplified porting and tool development. This limitation restricts access to a few exotic functions but covers the vast majority of cases, including the C APIs supported by `rdyncall`.

We presented various examples of rapid application development using *dynamic* R bindings to C APIs of two families of library interfaces. However, C allows a large degree of freedom, and interface designs of C APIs can be diverse. We have focused on typical elements on the basis of some standard APIs.

A few pratical issues should be solved first before users are likely to make use of dynamic bindings on a larger scale. If the installation of C libraries were as convenient as language extensions through CRAN, `rdyncall`-based applications could be more effectively used. Thus we are considering development of a distributed installation framework for C packages, designed for dynamic FFIs.

Moreover, it is certainly quite conceivable to think about a *foreign documentation interface.* We assume it would greatly help R users to use a C API if a corresponding R manual page or convenient interface to C documentation were available. Since the documentation of C libraries is not standardized automation remains a difficult task.

By example of `rdyncall`, an implementation of the Dynamic Bindings Model was shown. Major tasks of the middleware, such as parsing of *DynPort* files and wrapper creation, were implemented in R. This was beneficial for exploring techniques for wrapping objects. However, similar work needs to be done repeatedly for any other languages. Further improvements to the *DynPort* format require synchronization between language extensions.

Some abstraction layer for *DynPort* tasks and wrapper management within the middle-level layer of the *Dynamic Bindings Model* is conceivable. Then a common code base for managing C components could be shared among scripting language extensions, for example, by using C/C++ code and the C extension interface of languages. If we consider to design such an abstraction layer, we then could also ask the *next* question for a general abstraction to languages, possibly encapsulated as plug-ins, in a new model for generic interoperability. The findings of this work, and the Dynamic Bindings Model, contribute a portable implementation of a plug-in for the language C and its libraries.

Bibliography

A-EON Technology Ltd. (2011). AmigaONE X1000. URL `http://a-eon.com/x1000.html`; accessed 2013-09-30.

Adler, D. (2012). Foreign Library Interface. *The R Journal*, 4(1):30–40.

Adler, D. and Nenadic, O. (2003). A Framework for an R to OpenGL Interface for Interactive 3D graphics. *Proceedings of Distributed Statistical Computing 2003* (working paper) URL `http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Drafts/AdlerNenadic.pdf`; accessed 2013-04-18.

Adler, D., Nenadic, O., and Zucchini, W. (2003). RGL: A R-library for 3d visualization with OpenGL. In *Proceedings of the 35th Symposium of the Interface: Computing Science and Statistics*, volume 35, pp 419–429, Salt Lake City, USA.

Apple Inc. (2010a). *iOS ABI Function Call Guide.* (version 2010-07-07) URL `https://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf`; accessed 2013-03-31.

Apple Inc. (2010b). *Mac OS X ABI Function Call Guide.* (version 2010-11-17) URL `https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/LowLevelABI/Mac_OS_X_ABI_Function_Calls.pdf`; accessed 2013-03-17.

ARM Limited (2009). Procedure Call Standard for the ARM Architecture. URL `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf`; accessed 2013-04-18.

AROS Development Team (1995-2013). AROS Research Operating System. URL `http://aros.sourceforge.net/`; accessed 2013-09-30.

AT&T (1990). *Unix System V: Application Binary Interface.* Prentice Hall.

AT&T (1991). *System V Application Binary Interface: Intel386 processor supplement: UNIX system.* Prentice Hall.

AT&T (1991). *System V Application Binary Interface: MIPS RISC Processor Supplement: UNIX System V.* Prentice Hall.

AT&T and SCO (1996). *System V Application Binary Interface: SPARC Processor Supplement.* Prentice Hall.

BBC News (2012). AMD in chip tie-up with UK's ARM. URL `http://www.bbc.co.uk/news/technology-20137041`; accessed 2013-09-15.

Becker, R. A. (1994). A brief history of S. AT&T Bell Laboratories, New Jersey, USA. URL `http://cm.bell-labs.com/stat/doc/94.11.ps`; accessed 2013-10-27.

Becker, R. A. and Chambers, J. M. (1984). *S: An Interactive Environment for Data Analysis and Graphics.* Chapman and Hall/CRC.

Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988). *The New S Language.* Chapman and Hall/CRC.

Bielman, J. (2010). CFFI - Common Foreign Function Interface. (CL package version 0.10.6) URL `http://common-lisp.net/project/cffi/`; accessed 2013-04-18.

Box, D. (1998). *Essential COM, 1st edition.* Addison-Wesley Professional, Boston, MA, USA.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (1997). Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66.

Britton, R. (2003). *MIPS Assembly Language Programming.* Prentice Hall.

Brown, E. (2012). *csound: Accessing Csound functionality through R.* (R package version 0.1-1) URL `http://CRAN.R-project.org/package=csound`; accessed 2013-10-13.

Chafik, O. (2011). BridJ. (Java package version 0.5) URL `http://code.google.com/p/bridj/`; accessed 2013-04-18.

Chambers, J. M. (1998). *Programming with data: A guide to the S language.* Springer.

Chambers, J. M. and Hastie, T. J. (1991). *Statistical Models in S.* Chapman and Hall/CRC.

Christopher, E. (2003). mips eabi documentation. URL `http://sourceware.org/ml/binutils/2003-06/msg00436.html`; accessed 2013-09-20.

Ciccone, J., Gifford, J., Lankhorst, M., and Oliver, R. (2012). Cross-Compiled Linux From Scratch - Embedded. URL `http://cross-lfs.org/view/clfs-embedded/arm/index.html`; accessed 2013-04-18.

Clark, J. (1998-2007). The Expat XML Parser. (C library version 2.0.1) URL `http://expat.sourceforge.net/`; accessed 2013-04-18.

Clark, J. (2001). XSL transformations (XSLT) version 1.1, working draft. URL `http://www.w3.org/TR/2001/WD-xslt11-20010824`; accessed 2013-03-04.

Clark, J., DeRose, S., et al. (1999). XML Path Language (XPath), version 1.0. URL `http://www.w3.org/TR/1999/REC-xpath-19991116/`; accessed 2013-05-05.

CompuPhase (1998-2013). pawnscript - An embedded scripting language. (Interpreter version 4.0.4733) URL `http://www.compuphase.com/pawn/pawn.htm`; accessed 2013-10-01.

da Silva, A. R. F. (2011). cudabayesreg: Parallel implementation of a bayesian multilevel model for fmri data analysis. *Journal of Statistical Software*, 44(4):1–24.

Dandamudi, S. P. (2005a). *Guide to RISC Processors - for Programmers and Engineers*. Springer.

Dandamudi, S. P. (2005b). *Introduction to Assembly Language Programming: For Pentium and RISC Processors*. Springer.

Debian (2012). ARM Hardfloat Port. URL `http://wiki.debian.org/ArmHardFloatPort`; accessed 2013-04-18.

Developers, S. (2013). Source Python Extensions. (Software version 2.8.12) URL `http://python.eventscripts.com/pages/Spe`; accessed 2013-10-01.

Dutky, S. and Maechler, M. (2012). *bitops: Functions for Bitwise operations*. (R package version 1.0) URL `http://CRAN.R-project.org/package=bitops`; accessed 2013-06-12.

Dvorak, J. C. (1996). Inside Track. *PC Magazine*. October 1996.

Eddelbuettel, D. and Francois, R. (2012). *RInside: C++ classes to embed R in C++ applications*. (R package version 0.2.10) URL `http://CRAN.R-project.org/package=RInside`; accessed 2013-06-12.

Fisher, K., Pucella, R., and Reppy, J. (2000). Data-level interoperability. In *Electronic Notes in Theoretical Computer Science*.

Flynt, C. (2003). *TCL/TK - A Developer's Guide*. Morgan Kaufmann, San Francisco, USA.

Fog, A. (2004-2012). Calling conventions for different c++ compilers and operating systems. (Version 2012-02-29) URL `http://www.agner.org/optimize/calling_conventions.pdf`; accessed 2012-04-29.

Foundation, T. P. (2009-2013). NQP - Not Quite Perl. (Interpreter version 2013.10) URL `https://github.com/perl6/nqp`; accessed 2013-10-26.

Fuegi, J. and Francis, J. (2003). Lovelace & Babbage and the Creation of the 1843 “Notes”. *IEEE Annals of the History of Computing*, 25(4):16–26.

Furman, S. and Bandhauer, J. (2012). XPCOM Type Library File Format. (Version 2012-08-06) URL `http://www-archive.mozilla.org/scriptable/typelib_file.html`; accessed 2013-05-02.

Garcia-Ripoll, J. J. and Rosenberg, K. M. (2006). The ECL Manual. URL `http://ecls.sourceforge.net/new-manual`; accessed 2013-04-18.

Gastner, M. T. and Newman., M. E. J. (2013). Diffusion-based Cartograms with R. (R package in development) URL `https://github.com/thomaszumbrunn/cart`; accessed 2013-10-27.

Geraci, A., Katki, F., McMonegal, L., Meyer, B., and Porteous, H. (1991). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* IEEE.

Golemon, S. (2006). *Extending and Embedding PHP.* Sams, Indianapolis, USA.

Green, A. and et al (2011). libffi - A Portable Foreign Function Interface Library. (C library version 3.0.10) URL `http://sourceware.org/libffi/`; accessed 2013-04-18.

Guy Wright and Glenn Suokko (1986). Andy Warhol: An Artist and His Amiga. Amiga World magazine, Issue 3, 1986, IDG.

Haible, B. (2004). ffcall - foreign function call libraries. (C library version 1.10) URL `http://www.gnu.org/s/libffcall/`; accessed 2013-04-18.

Haiku, Inc. (2001-2013). What is Haiku? URL `http://www.haiku-os.org/about`; accessed 2013-10-01.

HAL Computer Systems, I. (1998). *SPARC64-III User's Guide.* HAL Computer Systems, Inc., Campbell, California, USA.

Hearn, M. (2004). Writing shared libraries. URL `http://plan99.net/~mike/writing-shared-libraries.html`; accessed 2013-05-23.

Heller, T. (2011). ctypes - A foreign function library for Python. URL `http://starship.python.net/crew/theller/ctypes/`; accessed 2013-04-18.

Hornik, K. (2012). Are There Too Many R Packages? *Austrian Journal of Statistics*, 41(1):59–66.

Hoxe, S., Karim, F., Hay, B., and Warren, H. (1996). *The PowerPC Compiler Writer's Guide.* Warthmann Associates, Palo Alto, California, USA.

IBM (2000). PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors. URL `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2`; accessed 2013-10-08.

IBM (2010). PowerISA. URL `https://www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf`; accessed 2013-09-30.

IBM and Motorola (1997). PowerPC Microprocessor Family: The Programming Environments. URL `http://www.freescale.com/files/32bit/doc/user_guide/MPCFPE_AD_R1.pdf`; accessed 2013-10-08.

Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. (1996). Lua—An Extensible Extension Language. *Software—Practice and Experience*, 26(6):635–652.

Ihaka, R. and Gentleman, R. (1996). R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.

ISO (2011). *ISO/IEC 9899:2011 Information technology — Programming languages — C.* International Organization for Standardization.

Jenness, T. and Cozens, S. (2003). *Extending and Embedding Perl.* Manning, Greenwich, CT, USA.

Jermár, J. (2007). Porting SPARTAN kernel to SPARC V9 architecture. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics. URL `http://www.helenos.org/doc/theses/jj-thesis.pdf`; accessed 2013-10-13.

Joyeux, S. (2004-2009). Typelib: a C++ type and value introspection library. (C++/Ruby package) URL `http://typelib.sourceforge.net/html/index.html`; accessed 2013-10-26.

Kaiser, H. (2011). Wave V2.0 - Boost C++ Libraries. (C++ library version 2.1.0; in Boost C++ library version 1.45) URL `http://www.boost.org/doc/libs/release/libs/wave/index.html`; accessed 2013-04-18.

Kernighan, B. W. and Pike, R. (1983). *The Unix programming environment.* Prentice Hall.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language, ANSI C, 2nd edition.* Prentice Hall, Englewood Cliffs, New Jersey, USA.

King, B. (2004). GCC-XML. (Program version 0.6.0) URL `http://gccxml.github.io/HTML/Index.html`; accessed 2013-04-18.

Kloss, G. K. (2008). Automatic C Library Wrapping – Ctypes from the Trenches. *The Python Papers*, 3(3). ISSN 1834-3147.

Knight, G. (2006). Amiga history guide. URL `http://www.amigahistory.co.uk/phase5story.html`; accessed 2013-09-29.

Kolb, C. and Pharr, M. (2006). Options Pricing on the GPU. *GPU Gems*, 2:719–731. Addison-Wesley.

Lang, D. T. (2011). Rffi. (R package version 0.3-0) URL `http://www.omegahat.org/Rffi/`; accessed 2013-04-18.

Language Popularity Index Project (2012). The Transparent Language Popularity Index. URL `http://lang-index.sourceforge.net/`; accessed 2013-04-18.

Lantinga, S. and et al (2012). libSDL: Simple DirectMedia Layer. (C library version 1.2.15) URL `http://www.libsdl.org/`; accessed 2013-04-18.

Lengyel, E. (2003). *The OpenGL Extensions Guide*. Charles River Media, Inc., Massachusetts, USA.

Levine, J. R. (2000). *Linkers and Loaders*. Morgan Kaufmann, San Francisco, USA.

MacPorts Project (2002). MacPorts Project. URL `http://www.macports.org`; accessed 2013-05-05.

Martin, K., Hoffman, B., Cedilnik, A., King, B., and Nuendorf, A. (2003). *Mastering CMake: A cross-platform build system*. Kitware Incorporated, New York, USA.

Mascarenhas, F. (2009). Alien - Pure Lua extensions. (Lua package version 0.5.1) URL `http://alien.luaforge.net/`; accessed 2013-04-18.

Mashey, J. R. (2004). Languages, Levels, Libraries, and Longevity. *ACM Queue*, 2(9):32–38.

Matsumoto, Y. (2002). *Ruby in a nutshell - a desktop quick reference*. O'Reilly.

Matz, M., Hubička, J., Jaeger, A., and Mitchell, M. (2012). System V Application Binary Interface: AMD64 Architecture Processor Supplement. (version 0.99.6) URL `http://ensiwiki.ensimag.fr/images/d/de/System_V_Application_Binary_Interface_AMD64_Architecture_Processor_Supplement.pdf`.

McFarlane, N. (2003). *Rapid Application Development with Mozilla*. Prentice Hall, New Jersey, USA.

Meissner, W. (2011). Ruby-FFI. (Ruby package version 1.0.10) URL `https://github.com/ffi/ffi/wiki`; accessed 2013-04-18.

Microsoft (2012). COM, DCOM, and Type Libraries. (version 2012-10-26) URL `http://msdn.microsoft.com/en-us/library/windows/desktop/aa366757(v=vs.85).aspx`; accessed 2013-05-02.

Microsoft (2013). x64 Software Conventions. URL `http://msdn.microsoft.com/en-us/library/7kcdt6fy.aspx`; accessed 2013-04-18.

Miller, C. and Zovi, D. A. D. (2009). *The Mac Hacker's Handbook*. Wiley.

MorphOS Development Team (2000-2013). MorphOS. URL `http://www.morphos-team.net/`; accessed 2013-09-29.

Muhammad, H. and Ierusalimschy, R. (2007). C APIs in Extension and Extensible Languages. *Journal of Universal Computer Science*, 13(6):839–853.

Murdoch, D. (2001). RGL: An R interface to OpenGL. In *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, Vienna, Austria.

Muresan, R., editor (2005). *Embedded System Development and Labs for ARM, English Edition*. University of Guelph Canada. URL `http://www.eos.uoguelph.ca/webfiles/rmuresan/ENGG4420%20OLD%20pre%20F10/EmbeddedSystemsAndLabsForARM-V1.1.pdf`; accessed 2013-09-14.

NetBSD (2013). NetBSD ELF FAQ. URL `http://www.netbsd.org/docs/elf.html`; accessed 2013-06-01.

OpenBSD (2013). OpenBSD: How to handle shared libraries in the ports tree. URL `http://openbsd.org/porting/libraries.html`; accessed 2013-05-24.

OpenCSW project (2002). OpenCSW Solaris packages. URL `http://www.opencsw.org`; accessed 2013-05-05.

OpenGL Architecture Review Board and D. Shreiner and et al (2005). *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.* Addison Wesley.

Ousterhout, J. (1990). Tcl: An embeddable command language. In *Proceedings of the Winter 1990 USENIX Conference*, pp 133–146, Washington, D.C., USA.

Ousterhout, J. K. (1998). Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30.

Pall, M. (2012). LuaJIT FFI Library. URL `http://luajit.org/ext_ffi.html`; accessed 2013-04-13.

Paul, R. P. (2000). *SPARC architecture, assembly language programming, and C, second edition.* Prentice Hall, second edition.

Paul Moore, Gaal Yahas, A. V. (2008). FFI - Perl Foreign Function Interface. (Perl module version 1.04) URL `http://search.cpan.org/~gaal/FFI/FFI.pm`; accessed 2013-04-18.

Pendleton, B. (2003). Game programming with the Simple DirectMedia Layer. *Linux Journal*, 2003(110):1. Online version URL `http://www.linuxjournal.com/article/6410`; accessed 2013-10-29.

Philipp, T. (2011). DynOS Project. URL `http://dyncall.org/dynos`; accessed 2013-04-18.

Pike, R. (1989). Notes on Programming in C. URL `http://doc.cat-v.org/bell_labs/pikestyle`; accessed 2013-04-18.

Pike, R. and Kernighan, B. (1984). Program design in the UNIX environment. *AT&T Bell Laboratories Technical Journal*, 63(8):1595–1605.

R Development Core Team (2012a). *R: A Language and Environment for Statistical Computing (version 2.15.1).* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

R Development Core Team (2012b). *R Internals (version 2.15.1).* R Foundation for Statistical Computing, Vienna, Austria.

R Development Core Team (2012c). *Writing R Extensions (version 2.15.1).* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-000051-11-9.

Rodriguez, C. S. and Fischer, G. (2006). *The Linux Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures.* Prentice Hall.

Rost, R. J. (2004). *OpenGL Shading Language.* Addison-Wesley.

Schlather, M., Menck, P., Singleton, R., Pfaff, B., and team, R. C. (2013). *RandomFields: Simulation and Analysis of Random Fields.* (R package version 2.0.66) URL `http://CRAN.R-project.org/package=RandomFields`; accessed 2013-08-03.

Sloss, A. N., Symes, D., and Wright, C. (2004). *ARM System Developer's Guide.* Morgan Kaufmann.

SPARC International (1992). *The SPARC Architecture Manual – Version 8.* SPARC International Inc., CA, USA.

SPARC International (1997). System V Application Binary Interface: SPARC Version 9 Processor Supplement. (Delta Document 1.35x, draft) URL `http://www.sparc.com/standards/64.psabi.1.35.ps.Z`; accessed 2013-04-18.

Stallman, R. M. (2003). *Using GCC: the GNU compiler collection reference manual.* GNU Press.

Stallman, R. M. and McGrath, R. (2002). *GNU Make: A Program for Directing Recompilation.* GNU Press.

Steven Sobek and Kevin Burke (2004). PowerPC Embedded Application Binary Interface (EABI): 32-Implementation. URL `http://www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf`; accessed 2013-09-19.

Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., and Schulten, K. (2007). Accelerating molecular modeling applications with graphics processors. *Journal of computational chemistry*, 28(16):2618–2640.

Sweetman, D. (2007). *See MIPS Run: Linux.* Elsevier.

SWIG Developers (2009). SWIG 1.3 Documentation. URL `http://www.swig.org/Doc1.3/SWIGDocumentation.pdf`; accessed 2013-04-18.

Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component Software: Beyond Object-Oriented Programming, Second Edition.* Addison-Wesley.

Tanenbaum, A. S. (2009). *Modern Operating Systems, Third Edition.* Pearson Prentice Hall.

Tasajrvi, L. (2004). *DEMOSCENE: The Art of Real-Time.* Even Lake Studios, Helsinki.

Taylor, T. J. (2013). jsh - The Hypersoft JavaScript Shell. (Interpreter, in development) URL `https://github.com/hypersoft/jsh`; accessed 2013-10-26.

The Open Group (1997). 64-Bit Programming Models: Why LP64? URL `http://www.unix.org/version2/whatsnew/lp64_wp.html`; accessed 2013-09-25.

The SWIG Developers (1995–2010). SWIG: Simplified Wrapper and Interface Generator. URL `http://www.swig.org/`; accessed 2013-04-18.

TIOBE (2012). TIOBE Programming Community Index. URL `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`; accessed 2012-04-18.

Turber, S. (2000). *ARM System on Chip Architecture, Second Edition.* Addison-Wesley.

Urban, R. (2004). Design Issues for Foreign Function Interfaces. (version 2004-05-04) URL `http://autocad.xarch.at/lisp/ffis.html`; accessed 2013-04-18.

van Eerten, P. (2008-2012). GTK-server. (Program version 2.3.1) URL `http://www.gtk-server.org/`; accessed 2013-10-26.

van Rossum, G. (2012). *Extending and Embedding the Python Interpreter.* Python Software Foundation, 2.7 edition. URL `http://docs.python.org/2/extending/`; accessed 2013-10-29.

Van Rossum, G. and Drake Jr, F. L. (1995). *Python reference manual.* Centrum voor Wiskunde en Informatica. (Online Version) URL `http://docs.python.org/2/reference/`; accessed 2013-10-29.

Veillard, D., Reese, B., and et al (2009). XSLT C library for GNOME. (C library version 1.1.26) URL `http://xmlsoft.org/XSLT/`; accessed 2013-04-18.

Wall, L. (2000). *Programming Perl.* O'Reilly.

Weaver, D. L., editor (2008). *OpenSPARC Internals: OpenSPARC T1/T2 Chip Multithreaded Throughput Computing.* Lulu, Inc., NC, USA.

Weaver, D. L. and Germond, T. (1994). *The SPARC Architecture Manual - Version 9.* Prentice Hall.

Weisser, W. (2007). C/Invoke. (C library version 1.0) URL `http://cinvoke.teegra.net/index.html`; accessed 2013-04-18.

Wheeler, D. A. (2003). Program Library HOWTO. URL `http://tldp.org/HOWTO/Program-Library-HOWTO`; accessed 2013-06-01.

Worthington, J. (2010-2013). Native call interface for Rakudo. (Language extension, in development) URL `https://github.com/jnthn/zavolaj`; accessed 2013-10-26.

Yiu, J. (2010). *The Definitive Guide to the ARM Cortex-M3, Second Edition.* Elsevier.

Zucker, S. and Karhi, K. (1995). *System V Application Binary Interface: PowerPC Processor Supplement.* SunSoft, USA. URL `http://refspecs.linux-foundation.org/elf/elfspec_ppc.pdf`; accessed 2013-03-17.