ECL User's Guide

Giuseppe Attardi
Juan Jose Garcia Ripoll (revised version)
Daniel Kochmański (revised revised version)

Copyright © 1990, Giuseppe Attardi
Copyright © 2000, Juan Jose Garcia Ripoll
Copyright © 2015, Daniel Kochmański

Preface

Embeddable Common Lisp is an implementation of Common-Lisp originally designed for being *embeddable* into C based applications. This document describes the Embeddable Common Lisp implementation and how it differs from [ANSI, see [Bibliography], page 203] and [Steele:84, see [Bibliography], page 203]. Chapter 4 [Developer's guide], page 159, and Chapter 1 [User's guide], page 11, for the details about the implementation and how to interface with other languages.

Table of Contents

Introduction **3**
About this book 3
User's guide 3
Developer's guide 3
Standards 3
Extensions 3
What is ECL 3
History 4
Credits 6
Copyrights 7
Copyright of ECL 7
Copyright of this manual 8

1 User's guide **11**
1.1 Building ECL 11
1.1.1 Autoconf based configuration 11
1.1.2 Platform specific instructions 12
1.1.2.1 MSVC based configuration 12
1.1.2.2 Android 12
1.1.2.3 iOS 13
1.2 Entering and leaving Embeddable Common Lisp 13
1.3 The break loop 16
1.4 Embedding ECL 16
1.4.1 Minimal Example 16
1.4.2 Embedding Reference 16
1.4.2.1 Starting and Stopping 16
1.4.2.2 Catching Errors and Managing Interrupts 19

2 Standards **23**
2.1 Overview 23
2.1.1 Reading this manual 23
2.1.2 C Reference 23
2.2 Evaluation and compilation 25
2.2.1 Compiler declaration `optimize` 25
2.2.2 `declaim` and `proclaim` 26
2.2.3 C Reference 27
2.2.3.1 ANSI Dictionary 27
2.3 Types and classes 27
2.3.1 C Reference 27
2.3.1.1 ANSI Dictionary 27
2.4 Data and control flow 27
2.4.1 Shadowed bindings 28

2.4.2 Minimal compilation . . . 28
2.4.3 Function types . . . 29
2.4.4 C Calling conventions . . . 29
2.4.5 C Reference . . . 30
2.4.5.1 ANSI Dictionary . . . 34
2.5 Objects . . . 35
2.5.1 C Reference . . . 35
2.5.1.1 ANSI Dictionary . . . 35
2.6 Structures . . . 36
2.6.1 Redefining a defstruct structure . . . 36
2.6.2 C Reference . . . 36
2.6.2.1 ANSI Dictionary . . . 36
2.7 Conditions . . . 36
2.7.1 C Reference . . . 36
2.7.1.1 ANSI dictionary . . . 38
2.8 Symbols . . . 38
2.8.1 C Reference . . . 38
2.8.1.1 ANSI Dictionary . . . 39
2.9 Packages . . . 40
2.9.1 C Reference . . . 40
2.9.1.1 ANSI Dictionary . . . 40
2.10 Numbers . . . 41
2.10.1 Numeric types . . . 41
2.10.2 Floating point exceptions . . . 42
2.10.3 Random-States . . . 42
2.10.4 Infinity and Not a Number . . . 43
2.10.5 Branch cuts and signed zero . . . 43
2.10.6 Dictionary . . . 44
2.10.7 C Reference . . . 44
2.10.7.1 Number C types . . . 44
2.10.7.2 Number constructors . . . 45
2.10.7.3 Number accessors . . . 46
2.10.7.4 Number coercion . . . 46
2.10.7.5 ANSI dictionary . . . 47
2.11 Characters . . . 50
2.11.1 Unicode vs. POSIX locale . . . 50
2.11.1.1 Character types . . . 50
2.11.1.2 Character names . . . 50
2.11.2 `#\Newline` characters . . . 51
2.11.3 C Reference . . . 51
2.11.3.1 C types . . . 51
2.11.3.2 Constructors . . . 51
2.11.3.3 Predicates . . . 52
2.11.3.4 Character case . . . 52
2.11.3.5 ANSI Dictionary . . . 52
2.12 Conses . . . 53
2.12.1 C Reference . . . 53
2.12.1.1 Accessors . . . 53

2.12.1.2 ANSI Dictionary . . . 54
2.13 Arrays . . . 57
2.13.1 Array limits . . . 57
2.13.2 Specializations . . . 57
2.13.3 C Reference . . . 58
2.13.3.1 Types and constants . . . 58
2.13.3.2 ecl_aet_to_symbol, ecl_symbol_to_aet . . . 58
2.13.3.3 Constructors . . . 59
2.13.3.4 Accessors . . . 60
2.13.3.5 Array properties . . . 60
2.13.3.6 ANSI Dictionary . . . 61
2.14 Strings . . . 62
2.14.1 String types & Unicode . . . 63
2.14.2 C reference . . . 63
2.14.2.1 Base string constructors . . . 63
2.14.2.2 String accessors . . . 64
2.14.2.3 Converting Unicode strings . . . 64
2.14.2.4 ANSI dictionary . . . 65
2.15 Sequences . . . 66
2.15.1 C Reference . . . 66
2.15.1.1 ANSI dictionary . . . 66
2.16 Hash tables . . . 68
2.16.1 Extensions . . . 68
2.16.1.1 Weakness in hash tables . . . 68
2.16.1.2 Thread-safe hash tables . . . 68
2.16.1.3 Hash tables serialization . . . 68
2.16.1.4 Custom equivalence predicate . . . 69
2.16.1.5 Example . . . 69
2.16.2 C Reference . . . 69
2.16.2.1 ANSI dictionary . . . 69
2.17 Filenames . . . 70
2.17.1 Syntax . . . 70
2.17.2 Wild pathnames and matching . . . 71
2.17.3 C Reference . . . 71
2.17.3.1 ANSI dictionary . . . 71
2.18 Files . . . 72
2.18.1 Dictionary . . . 72
2.18.2 C Reference . . . 73
2.18.2.1 ANSI Dictionary . . . 73
2.19 Streams . . . 73
2.19.1 ANSI Streams . . . 73
2.19.1.1 Supported types . . . 74
2.19.1.2 Element types . . . 74
2.19.1.3 External formats . . . 74
2.19.2 Dictionary . . . 76
2.19.2.1 File Stream Extensions . . . 76
2.19.2.2 Binary Stream Extensions . . . 76
2.19.2.3 External Format Extensions . . . 76

2.19.2.4 Sequence Streams . . . 78
2.19.3 C Reference . . . 79
2.19.3.1 ANSI dictionary . . . 79
2.20 Printer . . . 80
2.20.1 C Reference . . . 81
2.20.1.1 ANSI Dictionary . . . 81
2.21 Reader . . . 82
2.21.1 `*read-supress*` . . . 82
2.21.2 C Reference . . . 82
2.21.2.1 ANSI Dictionary . . . 82
2.22 System construction . . . 83
2.22.1 Dictionary . . . 83
2.22.1.1 Extensions . . . 83
2.22.2 C Reference . . . 84
2.22.2.1 ANSI Dictionary . . . 84
2.23 Environment . . . 84
2.23.1 Dictionary . . . 84
2.23.2 C Reference . . . 87
2.23.2.1 ANSI Dictionary . . . 87

3 Extensions . . . 89

3.1 System building . . . 89
3.1.1 Compiling with ECL . . . 89
3.1.1.1 Portable FASL . . . 90
3.1.1.2 Native FASL . . . 91
3.1.1.3 Object file . . . 92
3.1.1.4 Static library . . . 92
3.1.1.5 Shared library . . . 93
3.1.1.6 Executable . . . 93
3.1.1.7 Summary . . . 94
3.1.2 Compiling with ASDF . . . 94
3.1.2.1 Example code to build . . . 94
3.1.2.2 Build it as an single executable . . . 94
3.1.2.3 Build it as shared library and use in C . . . 95
3.1.2.4 Build it as static library and use in C . . . 97
3.1.3 Cross compilation . . . 97
3.1.3.1 Getting started with cross compilation . . . 97
3.1.3.2 Limitations and pitfalls for cross compiling Common Lisp . . . 98
3.1.4 C compiler configuration . . . 100
3.1.4.1 Compiler flags . . . 100
3.1.4.2 Compiler & Linker programs . . . 101
3.2 Operating System Interface . . . 101
3.2.1 Command line arguments . . . 101
3.2.2 External processes . . . 103
3.2.3 FIFO files (named pipes) . . . 105
3.2.4 Operating System Interface Reference . . . 105
3.3 Foreign Function Interface . . . 106

3.3.1 What is a FFI? ... 106
3.3.2 Two kinds of FFI ... 106
3.3.3 Foreign objects ... 107
3.3.3.1 C Reference ... 108
3.3.4 Higher level interfaces ... 109
3.3.5 SFFI Reference ... 111
3.3.6 DFFI Reference ... 115
3.3.7 UFFI Reference ... 115
3.3.7.1 Primitive Types ... 115
3.3.7.2 Aggregate Types ... 117
3.3.7.3 Foreign Objects ... 120
3.3.7.4 Foreign Strings ... 124
3.3.7.5 Functions and Libraries ... 128
3.4 Native threads ... 129
3.4.1 Tasks, threads or processes ... 129
3.4.2 Processes (native threads) ... 130
3.4.3 Processes dictionary ... 130
3.4.4 Locks (mutexes) ... 134
3.4.5 Locks dictionary ... 134
3.4.6 Readers-writer locks ... 135
3.4.7 Read-Write locks dictionary ... 136
3.4.8 Condition variables ... 136
3.4.9 Condition variables dictionary ... 137
3.4.10 Semaphores ... 137
3.4.11 Semaphores dictionary ... 138
3.4.12 Barriers ... 139
3.4.13 Barriers dictionary ... 139
3.4.14 Atomic operations ... 139
3.4.15 Atomic operations dictionary ... 139
3.5 Signals and Interrupts ... 143
3.5.1 Problems associated to signals ... 143
3.5.2 Kinds of signals ... 143
3.5.2.1 Synchronous signals ... 143
3.5.2.2 Asynchronous signals ... 144
3.5.3 Signals and interrupts in ECL ... 144
3.5.3.1 Handling of asynchronous signals ... 145
3.5.3.2 Handling of synchronous signals ... 145
3.5.4 Considerations when embedding ECL ... 146
3.5.5 Signals Reference ... 146
3.6 Memory Management ... 147
3.6.1 Introduction ... 147
3.6.2 Boehm-Weiser garbage collector ... 148
3.6.3 Memory limits ... 148
3.6.4 Memory conditions ... 149
3.6.5 Finalization ... 149
3.6.6 Memory Management Reference ... 150
3.7 Meta-Object Protocol (MOP) ... 152
3.7.1 Introduction ... 152

3.8 Gray Streams ... 152
3.8.1 Gray Stream Extensions ... 153
3.9 Tree walker ... 153
3.10 Local package nicknames ... 154
3.10.1 Overview ... 154
3.10.2 Package local nicknames dictionary ... 154
3.11 Package locks ... 155
3.11.1 Package Locking Overview ... 155
3.11.2 Operations Violating Package Locks ... 155
3.11.3 Package Lock Dictionary ... 156
3.12 CDR Extensions ... 157

4 Developer's guide ... 159
4.1 Sources structure ... 159
4.1.1 src/c ... 159
4.2 Contributing ... 162
4.3 Defun preprocessor ... 163
4.4 Manipulating Lisp objects ... 164
4.4.1 Objects representation ... 164
4.4.2 Constructing objects ... 167
4.5 Environment implementation ... 175
4.6 The interpreter ... 175
4.6.1 ECL stacks ... 175
4.6.2 Procedure Call Conventions ... 175
4.6.3 The lexical environment ... 177
4.6.4 The interpreter stack ... 177
4.7 The compiler ... 179
4.7.1 The compiler translates to C ... 179
4.7.2 The compiler mimics human C programmer ... 179
4.7.3 Implementation of Compiled Closures ... 181
4.7.4 Use of Declarations to Improve Efficiency ... 182
4.7.5 Inspecting generated C code ... 183
4.8 Porting ECL ... 184
4.9 Removed features ... 184

Indexes ... 187
Concept index ... 187
Configure option index ... 188
Feature index ... 188
Example index ... 188
Function index ... 189
Variable index ... 194
Type index ... 194
Common Lisp symbols ... 195
C/C++ index ... 197

Bibliography ... 203

Introduction

About this book

This manual is part of the ECL software system. It documents deviations of ECL from various standards ([ANSI, see [Bibliography], page 203], [AMOP, see [Bibliography], page 203],...), extensions, daily working process (compiling files, loading sources, creating programs, etc) and the internals of this implementation.

It is not intended as a source to learn Common Lisp. There are other tutorials and textbooks available in the Net which serve this purpose. The homepage of the Common-Lisp.net (`https://common-lisp.net`) contains a good list of links of such teaching and learning material.

This book is structured into four parts:

User's guide

We begin with [Chapter 1 [User's guide], page 11] which provides introductory material showing the user how to build and use ECL and some of its unique features. This part assumes some basic Common Lisp knowledge and is suggested as an entry point for new users who want to start using Embeddable Common Lisp.

Developer's guide

[Chapter 4 [Developer's guide], page 159] documents Embeddable Common Lisp implementation details. This part is not meant for normal users but rather for the ECL developers and other people who want to contribute to Embeddable Common Lisp. This section is prone to change due to the dynamic nature of the software. Covered topics include source code structure, contributing guide, internal implementation details and many other topics relevant to the development process.

Standards

[Chapter 2 [Standards], page 23] documents all parts of the standard which are left as implementation specific or to which ECL doesn't adhere. For instance, precision of floating point numbers, available character sets, actual input/output protocols, etc.

Section covers also *C Reference* as a description of ANSI Common-Lisp from the C/C++ programmer perspective and *ANSI Dictionary* for Common-Lisp constructs available from C/C++.

Extensions

[Chapter 3 [Extensions], page 89] introduces all features which are specific to ECL and which lay outside the standard. This includes configuring, building and installing ECL multiprocessing capabilities, graphics libraries, interfacing with the operating system, etc.

What is ECL

Common-Lisp is a general purpose programming language. It lays its roots in the LISP programming language [LISP1.5, see [Bibliography], page 203] developed by John McCarthy in

the 80s. Common-Lisp as we know it ANSI Common-Lisp is the result of an standardization process aimed at unifying the multiple lisp dialects that were born from that language.

Embeddable Common Lisp is an implementation of the Common-Lisp language. As such it derives from the implementation of the same name developed by Giuseppe Attardi, which itself was built using code from the Kyoto Common-Lisp [Yasa:85, see [Bibliography], page 203]. [History], page 4, for the history of the code you are about to use.

Embeddable Common Lisp (ECL for short) uses standard C calling conventions for Lisp compiled functions, which allows C programs to easily call Lisp functions and vice versa. No foreign function interface is required: data can be exchanged between C and Lisp with no need for conversion.

ECL is based on a Common Runtime Support (CRS) which provides basic facilities for memory management, dynamic loading and dumping of binary images, support for multiple threads of execution. The CRS is built into a library that can be linked with the code of the application. ECL is modular: main modules are the program development tools (top level, debugger, trace, stepper), the compiler, and CLOS. A native implementation of CLOS is available in ECL. A runtime version of ECL can be built with just the modules which are required by the application.

The ECL compiler compiles from Lisp to C, and then invokes the C compiler to produce binaries. Additionally portable bytecode compiler is provided for machines which doesn't have C compiler. While former releases of ECL adhere to the the reference of the language given in Common-Lisp*: The Language*2 [Steele90, see [Bibliography], page 203], the ECL is now compliant with X3J13 ANSI Common Lisp [ANSI, see [Bibliography], page 203].

History

The ECL project is an implementation of the Common Lisp language inherits from many other previous projects, as shown in Figure 1.

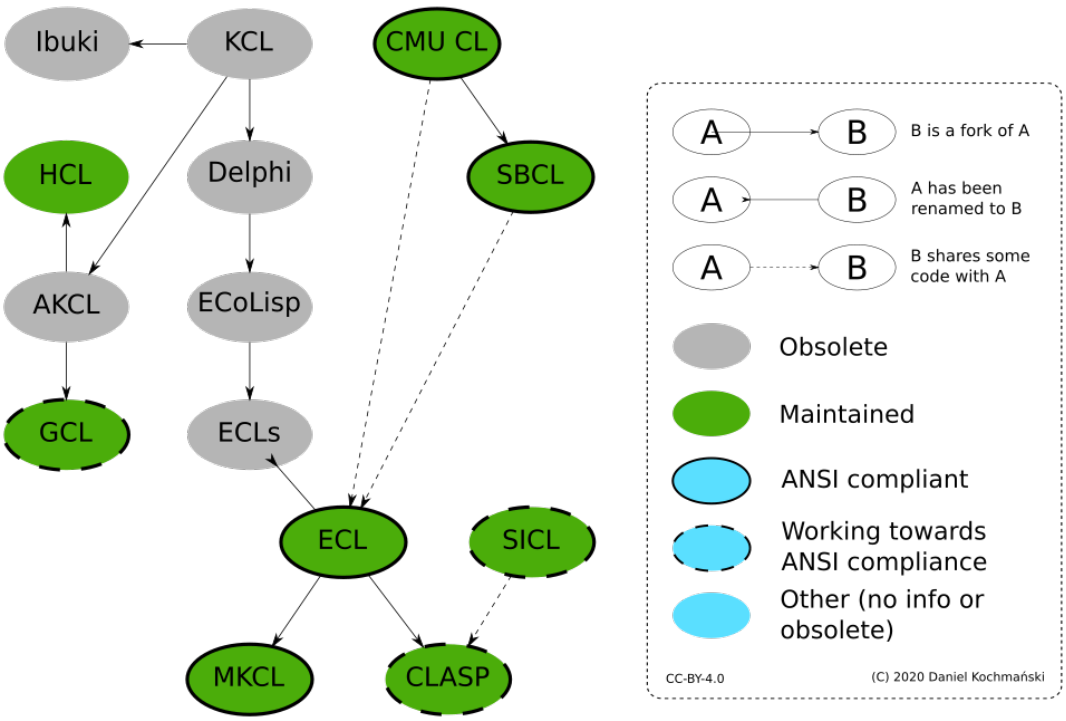


Figure 1: ECL's family tree

The oldest ancestor is the Kyoto Common Lisp (KCL), an implementation developed in 1984 at the the Research Institute for Mathematical Sciences, Kyoto University [Yasa:85, see [Bibliography], page 203] by Taiichi Yuasa and Masami Hagiya. This implementation was developed partially in C and partially in Common Lisp itself and featured a lisp to C translator.

Richard W. Weyhrauch in 1985 founded a company Ibuki to sell its version of KCL called Ibuki Common Lisp (IBCL) in the USA.

Giuseppe Attardi learned about KCL from Richard Weyhrauch in 1986. At the time he was working with the company DELPHI. He extended KCL by adding a CLOS implementation and other improvements. The whole combination was released in 1988 as DELPHI Common Lisp (DCL).

William F. Schelter in 1987 started improving KCL in several areas and developed Austin Kyoto Common-Lisp (AKCL). However, those changes had to be distributed as patches over the proprietary KCL implementation.

Much later in 1994, when KCL became open source (and by extension also AKCL), William F. Schelter re-released AKCL as GNU Common Lisp (GCL), implementation which primary

purpose at the time was to support the Maxima compute algebra system, which was also maintained by William F. Schelter.

In 1995 another AKCL descendant called HCL is released. This commercial implementation is an extension language for the application HP PE/SolidDesigner. A large part of the application's user interface is written in Common Lisp.

In 1994 Giuseppe Attardi left DELPHI and went back to the University of Pisa, where he decided to build a version of DCL that would allow integrating Lisp code with other languages[Attardi:95, see [Bibliography], page 203]. Since Lisp had become less popular and Lisp programmers were diminishing in number, he used the acronym ECL, which could be read as either the Embeddable Common Lisp as well as ECoLisp, as a joke for the ecological goal of preserving the disappearing species, the Lisp programmer. The implementation achieved rather good compliance to the informal specification of the language in CLTL2 [Steele:90, see [Bibliography], page 203], and which run on a rather big number of platforms.

The ECL project stagnated a little bit in the coming years. In particular, certain dependencies such as object binary formats, word sizes and some C quirks made it difficult to port it to new platforms. Furthermore, ECL was not compliant with the ANSI specification, a goal that other Common Lisps were struggling to achieve.

This is where the ECLS or ECL-Spain project began. Juanjo García-Ripoll took the ECoLisp sources and worked on them, with some immediate goals in mind: increase portability, make the code 64-bit clean, make it able to build itself from scratch, without other implementation of Common Lisp and restore the ability to link ECL with other C programs.

Those goals were rather quickly achieved. ECL was then ported to a number of platforms and with time the compatibility with the ANSI specification became a more important goal. At some point the fork ECLS, with agreement of Prof. Attardi, took over the original ECL implementation and it became what it is nowadays, a community project.

In 2013 once again project got unmaintained. In 2015 Daniel Kochmański took the position of a maintainer with consent of Juanjo García-Ripoll. In 2020 Marius Gerbershagen became the project co-maintainer.

The ECL project owes a lot to different people who have contributed in many different aspects, from pointing out bugs and incompatibilities of ECL with other programs and specifications, to actually solving these bugs and porting ECL to new platforms.

Current development of ECL is still driven by Daniel Kochmański and Marius Gerbershagen with main focus on improving ANSI compliance and compatibility with the Common Lisp libraries ecosystem, fixing bugs, improving speed and the portability. The project homepage is located at `https://common-lisp.net/project/ecl/`.

Credits

The Embeddable Common Lisp project is an implementation of the Common-Lisp language that aims to comply with the ANSI Common-Lisp standard. The first ECL implementations were developed by Giuseppe Attardi's who produced an interpreter and compiler fully conformant with the Common-Lisp as reported in *Steele:84*. ECL derives itself mostly from Kyoto Common-Lisp, an implementation developed at the Research Institute for Mathematical Sciences (RIMS), Kyoto University, with the cooperation of Nippon Data General

Corporation. The main developers of Kyoto Common-Lisp were Taiichi Yuasa and Masami Hagiya, of the Research Institute for Mathematical Sciences, at Kyoto University.

We must thank Giuseppe Attardi, Yuasa and Hagiya and Juan Jose Garcia Ripoll for their wonderful work with preceding implementations and for putting them in the Public Domain under the GNU General Public License as published by the Free Software Foundation. Without them this product would have never been possible.

This document is an update of the original ECL documentation, which was based in part on the material in [Yuasa:85, see [Bibliography], page 203]

The following people or organizations must be credited for support in the development of Kyoto Common-Lisp: Prof. Reiji Nakajima at RIMS, Kyoto University; Nippon Data General Corporation; Teruo Yabe; Toshiyasu Harada; Takashi Suzuki; Kibo Kurokawa; Data General Corporation; Richard Gabriel; Daniel Weinreb; Skef Wholey; Carl Hoffman; Naruhiko Kawamura; Takashi Sakuragawa; Akinori Yonezawa; Etsuya Shibayama; Hagiwara Laboratory; Shuji Doshita; Takashi Hattori.

William F. Schelter improved KCL in several areas and developed Austin Kyoto Common-Lisp (AKCL). Many ideas and code from AKCL have been incorporated in Embeddable Common Lisp.

The following is the partial list of contributors to ECL: Taiichi Yuasa and Masami Hagiya (KCL), William F. Schelter (Dynamic loader, conservative GC), Giuseppe Attardi (Top-level, trace, stepper, compiler, CLOS, multithread), Marcus Daniels (Linux port) Cornelis van der Laan (FreeBSD port) David Rudloff (NeXT port) Dan Stanger, Don Cohen, and Brian Spilsbury.

We have to thank for the following pieces of software that have helped in the development of Embeddable Common Lisp

Bruno Haible
: For the Cltl2-compliance test

Peter Van Eynde
: For the ANSI-compliance test

Symbolic's Inc.
: For the ANSI-compliant LOOP macro.

The Embeddable Common Lisp project also owes a lot to the people who have tested this program and contributed with suggestions, error messages and documentation: Eric Marsden, Hannu Koivisto, Jeff Bowden and Yuto Hayamizu, Bo Yao and others whose name we may have omitted.

Copyrights

Copyright of ECL

ECL is distributed under the GNU LGPL, which allows for commercial uses of the software. A more precise description is given in the Copyright notice which is shipped with ECL.

```
---- BEGINNING OF COPYRIGHT FOR THE ECL CORE ENVIRONMENT ------------
```

```
Copyright (c) 2019-2024 Daniel Kochmański and Marius Gerbershagen
Copyright (c) 2015-2018, Daniel Kochmański
Copyright (c) 2000-2013, Juan Jose Garcia Ripoll
Copyright (c) 1990, 1991, 1993 Giuseppe Attardi
Copyright (c) 1984 Taiichi Yuasa and Masami Hagiya
            All Rights Reserved

ECL is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA

PLEASE NOTE THAT:

This license covers all of the ECL program except for the files
      src/lsp/loop.lsp        ; Symbolic's LOOP macro
      src/lsp/pprint.lsp      ; CMUCL's pretty printer
      src/lsp/format.lsp      ; CMUCL's format
and the directories
      contrib/                ; User contributed extensions
      examples/               ; Examples for the ECL usage
Look the precise copyright of these extensions in the corresponding
files.

Examples are licensed under: (SPDX-License-Identifier) BSD-2-Clause

Report bugs, comments, suggestions to the ecl mailing list:
ecl-devel@common-lisp.net.

---- END OF COPYRIGHT FOR THE ECL CORE ENVIRONMENT ------------------
```

Copyright of this manual

Copyright Daniel Kochmański and Marius Gerbershagen, 2020

Copyright Daniel Kochmański, 2016

Copyright Juan José García-Ripoll, 2006

Copyright Kevin M. Rosenberg, 2002-2003 (UFFI Reference)

Trademark AllegroCL is a registered trademark of Franz Inc.

Trademark Lispworks is a registered trademark of Xanalys Inc.

Trademark Microsoft Windows is a registered trademark of Microsoft Inc.

Trademark Other brand or product names are the registered trademarks or trademarks of their respective holders.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with the no Front-Cover Texts, and with no Back-Cover Texts. Exact text of the license is available at `https://www.gnu.org/copyleft/fdl.html`.

1 User's guide

1.1 Building ECL

Due to its portable nature, ECL works on every (at least) 32-bit architecture which provides a proper C99 compliant compiler.

Operating systems on which ECL is reported to work: Linux, Darwin (Mac OS X), Solaris, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, Windows and Android. On each of them ECL supports native threads.

In the past Juanjo José García-Ripoll maintained a test farm which performed ECL tests for each release on number of platforms and architectures. Due to lack of the resources we can't afford such doing, however each release is tested by volunteers with an excellent package cl-test-grid (`https://common-lisp.net/project/cl-test-grid`) created and maintained by Anton Vodonosov.

1.1.1 Autoconf based configuration

ECL, like many other FOSS programs, can be built and installed with a GNU tool called Autoconf. This is a set of automatically generated scripts that detect the features of your machine, such as the compiler type, existing libraries, desired installation path, and configures ECL accordingly. The following procedure describes how to build ECL using this procedure and it applies to all platforms except for the Windows ports using Microsoft Visual Studio compilers (however you may build ECL with cygwin or mingw using the autoconf as described here).

To build Embeddable Common Lisp you need to

1. Extract the source code and enter it's directory

```
$ tar -xf ecl-xx.x.x.tgz
$ cd ecl-xx.x.x
```

2. Run the configuration file, build the program and install it

```
$ ./configure --prefix=/usr/local
$ make                          # -jX if you have X cores
$ make install
```

3. Make sure the program is installed and ready to run:

```
$ /usr/local/bin/ecl

ECL (Embeddable Common-Lisp) 16.0.0
Copyright (C) 1984 Taiichi Yuasa and Masami Hagiya
Copyright (C) 1993 Giuseppe Attardi
Copyright (C) 2000 Juan J. Garcia-Ripoll
Copyright (C) 2015 Daniel Kochmanski
ECL is free software, and you are welcome to redistribute it
under certain conditions; see file 'Copyright' for details.
Type :h for Help.
Top level in: #<process TOP-LEVEL>.
>
```

1.1.2 Platform specific instructions

1.1.2.1 MSVC based configuration

You need Microsoft Visual Studio 2015 or better to compile ECL, which flavor(Professional, Community, etc) does not matter.

You also need yasm (`http://yasm.tortall.net`) optionally to build gmp, fetch yasm-1.3.0-win64.exe and yasm-1.3.0-win32.exe, and put them in your system PATH directory.

In the Visual Studio's startup menu, click Developer Command Prompt for Visual Studio (`https://docs.microsoft.com/en-us/dotnet/framework/tools/developer-command-prompt-for-vs`) to open the console window. Alternatively, open the developer console from the start menu through "Visual Studio 20xx" -> "Visual Studio Tools" -> "VC" and select "x64 Native Tools Command Prompt for VS 20xx" or "x86 Native Tools Command Prompt for VS 20xx", depending on whether you want to build 32 or 64bit versions of ECL.

1. Change to the msvc directory.
2. Run nmake to build ECL.
3. Run nmake install prefix=d:\Software\ECL where the prefix is the directory where you want to install ECL.
4. If you want to build debug version, add ECL_DEBUG=1 to nmake command line.
5. If you want to build 64bit version, add ECL_WIN64=1 to nmake command line, you can also set GMP_TYPE=AMD64 to use specific assembly codes.
6. Optionally, if you want to build a self-installing executable, you can install NSIS and run nmake windows-nsi.

1.1.2.2 Android

Cross compiling ECL for Android requires first configuring the cross compilation toolchain. This requires the Android NDK version 22 or higher.

```
export NDK_PATH=/opt/android-ndk
export ANDROID_API=23
export TOOLCHAIN_PATH=${NDK_PATH}/toolchains/llvm/prebuilt/linux-x86_64 # use darwin i
export TARGET=armv7a-linux-androideabi
```

Here, `ANDROID_API` is the minimum Android API version ECL will run on. Moreover, an existing installation ECL is required. This installation has to be the same version as the one you are attempting to build. Assuming you have installed in "/usr/local":

```
export ECL_TO_RUN=/usr/local/bin/ecl
```

Finally, we can build and install the target ECL:

```
export CC="${TOOLCHAIN_PATH}/bin/clang --target=${TARGET}${ANDROID_API}"
export LD=${TOOLCHAIN_PATH}/bin/ld
export AR=${TOOLCHAIN_PATH}/bin/llvm-ar
export RANLIB=${TOOLCHAIN_PATH}/bin/llvm-ranlib
export ECL_TO_RUN=/usr/local/bin/ecl
./configure --host=${TARGET} \
            --prefix=‘pwd‘/ecl-android \
            --disable-c99complex \
```

```
            --with-cross-config=`pwd`/src/util/android-arm.cross_config
make -j9
make install
```

Library and assets are installed in the "ecl-android" directory and are ready to run on the Android system.

1.1.2.3 iOS

The cross-compilation steps for iOS are similar to those for Android.

```
export ECL_TO_RUN=/usr/local/bin/ecl
```

Configure the toolchain:

```
export IOS_VERSION_MIN="8.0"
export IOS_SDK_DIR="`xcode-select --print-path`/Platforms/iPhoneOS.platform/Developer/

export CC="clang"
export CXX="clang++"

export CFLAGS="-arch arm64 -miphoneos-version-min=${IOS_VERSION_MIN} -isysroot ${IOS_S
export CFLAGS="$CFLAGS -pipe -Wno-trigraphs -Wreturn-type -Wunused-variable"
export CFLAGS="$CFLAGS -fpascal-strings -fasm-blocks -fmessage-length=0 -fvisibility=h
export CFLAGS="$CFLAGS -O2 -DNO_ASM"

export LD="ld"
export LDFLAGS="-arch arm64 -pipe -std=c99 -gdwarf-2 -isysroot ${IOS_SDK_DIR}"
export LIBS="-framework Foundation"
```

Build and install the target library:

```
export CFLAGS="$CFLAGS -DGC_DISABLE_INCREMENTAL -DECL_RWLOCK"
export CXXFLAGS="$CFLAGS"
./configure --host=aarch64-apple-darwin \
            --prefix=`pwd`/ecl-iOS \
            --disable-c99complex \
            --disable-shared \
            --with-cross-config=`pwd`/src/util/iOS-arm64.cross_config
make -j9
make install
```

Library and assets in the "ecl-iOS" directory are ready to run on the iOS system.

1.2 Entering and leaving Embeddable Common Lisp

Embeddable Common Lisp is invoked by the command `ecl`.

```
% ecl
ECL (Embeddable Common-Lisp) 0.0e
Copyright (C) 1984 Taiichi Yuasa and Masami Hagiya
Copyright (C) 1993 Giuseppe Attardi
Copyright (C) 2000 Juan J. Garcia-Ripoll
Copyright (C) 2015 Daniel Kochmanski
```

```
ECL is free software, and you are welcome to redistribute it
under certain conditions; see file 'Copyright' for details.
Type :h for Help.  Top level.
Top level in: #<process TOP-LEVEL>.
>
```

When invoked, Embeddable Common Lisp will print the banner and initialize the system. The number in the Embeddable Common Lisp banner identifies the revision of Embeddable Common Lisp. `0.0e` is the value of the function `lisp-implementation-version`.

Unless user specifies `--norc` flag when invoking the Embeddable Common Lisp, it will look for the initialization files `~/.ecl` and `~/.eclrc`. If he wants to load his own file from the current directory, then he should pass the file path to the `--load` parameter:

```
% ecl --norc --load init.lisp
```

After the initialization, Embeddable Common Lisp enters the *top-level loop* and prints the prompt '`>`'.

```
Type :h for Help.  Top level.
>
```

The prompt indicates that Embeddable Common Lisp is now ready to receive a form from the terminal and to evaluate it.

Usually, the current package (i.e., the value of `*package*`) is the user package, and the prompt appears as above. If, however, the current package is other than the user package, then the prompt will be prefixed with the package name.

```
> (in-package "CL")
#<"COMMON-LISP" package>
COMMON-LISP> (in-package "SYSTEM")
#<"SI" package>
SI>
```

To exit from Embeddable Common Lisp, call the function `ext:quit`.

```
> (quit)
%
```

Alternatively, you may type `^D` on UNIX-like operating systems, i.e. press the key `D` while pressing down the control key (`Ctrl`), on Windows, you should use `^Z` followed by a return.

```
> ^D

%
```

The top-level loop of Embeddable Common Lisp is almost the same as that defined in Section 20.2 of [Steele:84, see [Bibliography], page 203]. Since the input from the terminal is in line mode, each top-level form should be followed by a newline. If more than one value is returned by the evaluation of the top-level form, the values will be printed successively. If no value is returned, then nothing will be printed.

```
> (values 1 2)
1
2
> (values)
```

```
>
```

When an error is signaled, control will enter the break loop.

```
> (defun foo (x) (bar x))
foo

> (defun bar (y) (bee y y))

bar
> (foo 'lish)
Condition of type: UNDEFINED-FUNCTION
The function BEE is undefined.

Available restarts:

1. (RESTART-TOPLEVEL) Go back to Top-Level REPL.

Broken at FOO. In: #<process TOP-LEVEL>.
>>
```

'`>>`' in the last line is the prompt of the break loop. Like in the top-level loop, the prompt will be prefixed by the current package name, if the current package is other than the `cl-user` package.

To go back to the top-level loop, type `:q`

```
>>:q

Top level in: #<process TOP-LEVEL>.
>
```

If more restarts are present, user may invoke them with by typing `:rN`, where `N` is the restart number. For instance to pick the restart number two, type `:r2`.

See [Section 1.3 [The break loop], page 16] for the details of the break loop.

The terminal interrupt (usually caused by typing `^C` (Control-`C`)) is a kind of error. It breaks the running program and calls the break level loop.

Example:

```
> (defun foo () (do () (nil)))
foo

> (foo)
^C

Condition of type: INTERACTIVE-INTERRUPT
Console interrupt.

Available restarts:

1. (CONTINUE) CONTINUE
2. (RESTART-TOPLEVEL) Go back to Top-Level REPL.
```

```
Broken at FOO. In: #<process TOP-LEVEL>.
>>
```

1.3 The break loop

1.4 Embedding ECL

1.4.1 Minimal Example

An example project is included in the ECL source distribution in the `examples/embed/` directory.

This example consists of a Common Lisp library (`hello-lisp.lisp`) and a system definition (`hello-lisp.asd`, See Section 3.1.2 [Compiling with ASDF], page 94) that is called from a C program (`hello.c`). The example `Makefile` shows how to build a static library from the Lisp library and link it with the C program.

1.4.2 Embedding Reference

1.4.2.1 Starting and Stopping

`int cl_boot` (*int argc, char **argv*); [Function]

Setup the lisp environment.

argc An integer with the number of arguments to this program.

argv A vector of strings with the arguments to this program.

Description This function must be called before any other function from the ECL library, including the creation of any lisp object or evaluating any lisp code. The only exception are `ecl_set_option` and `ecl_get_option`.

`int cl_shutdown` (*void*); [Function]

Close the lisp environment.

Description This function must be called before exiting a program that uses the ECL environment. It performs some cleaning, including the execution of any finalizers, unloading shared libraries and deleting temporary files that were created by the compiler.

`void ecl_set_option` (*int option, cl_fixnum value*); [Function]

Set a boot option.

option An integer from Table 1.1.

value A `cl_index` value for this option

Description This functions sets the value of different options that have to be customized *before* ECL boots. The table of options and default values [Table 1.1] shows that some of them are boolean, and some of them are unsigned integers.

We distinguish three sets of values. The first set determines whether ECL handles certain exceptions, such as access to forbidden regions of memory, interrupts via , floating point exceptions, etc.

The second set is related to the sizes of different stacks. Currently ECL uses four stacks: a bind stack for keeping assignments to special variables; a frame stack for implementing blocks, tagbodys and catch points; an interpreter stack for evaluating bytecodes, and finally the machine or C stack, of the computer we run in. We can set the expected size of these stacks, together with the size of a safety area which, if penetrated, will lead to the generation of a correctable error.

Name (ECL_OPT_*)	**Type**	**Default**	**Description**
INCREMENTAL_GC	boolean	TRUE	Activate generational garbage collector.
TRAP_SIGSEGV	boolean	TRUE	Capture SIGSEGV signals.
TRAP_SIGFPE	boolean	TRUE	Capture floating point exceptions.
TRAP_SIGINT	boolean	TRUE	Capture user interrupts.
TRAP_SIGILL	boolean	TRUE	Capture SIGILL exception.
TRAP_INTERRUPT_SIGNAL	boolean	TRUE	Capture the signal that implements mp:interrupt-process.
SIGNAL_HANDLING_THREAD	boolean	TRUE	Create a signal to capture and process asynchronous threads (See Section 3.5.2.2 [Signals and Interrupts - Asynchronous signals], page 144).
BOOTED	boolean	TRUE/FALSE	Has ECL booted (read only).
BIND_STACK_SIZE	cl_index	8192	Size of stack for binding special variables.
BIND_STACK_SAFETY_AREA	cl_index	128	
FRAME_STACK_SIZE	cl_index	2048	Size of stack for nonlocal jumps.
FRAME_STACK_SAFETY_AREA	cl_index	128	
LISP_STACK_SIZE	cl_index	32768	Size of interpreter stack.
LISP_STACK_SAFETY_AREA	cl_index	128	
C_STACK_SIZE	cl_index	0 or 1048576	Size of C stack in bytes. The effect and default value of this option depends on the operating system. On Unix, the default is 0 which means that ECL will use the stack size provided by the OS. If set to a non-default value, ECL will set the stack size to the given value unless the stack size provided by the OS is already large enough. On Windows, the stack size is set at build time and cannot be changed at runtime. Here, we use a default of 1 MiB. For other operating systems, it is up to the user to set this value to the available stack size so that ECL can reliably detect stack overflows.
C_STACK_SAFETY_AREA	cl_index	4192	
THREAD_INTERRUPT_SIGNAL	unsigned int	0	If nonzero, specify the unix signal which is used to communicate between different Lisp threads.

Table 1.1: Boot options for embedded ECL

`cl_fixnum ecl_get_option` (*int option*); [Function]

Read the value of a boot option.

option An integer from Table 1.1.

Description This functions reads the value of different options that have to be customized *before* ECL boots. The table of options and default values is Table 1.1.

`bool ecl_import_current_thread` (*cl_object name, cl_object bindings*); [Function]

Import an external thread in the Lisp environment.

name Thread name.

bindings Unused (specifying initial bindings for external threads is not supported currently)

returns True if the thread was successfully imported, false otherwise.

Description External threads, i.e. threads which are not created in the Lisp world using the routines described in Section 3.4.2 [Processes (native threads)], page 130, need to be imported with `ecl_import_current_thread` before Lisp code can be executed.

See also ecl_release_current_thread

`void ecl_release_current_thread` (*void*); [Function]

Release an external thread imported with `ecl_import_current_thread`. Must be called before thread exit to prevent memory leaks.

`ECLDIR` [Environment variable]

Specify a non-standard installation directory.

Description ECL includes various files for external modules (e.g. asdf, sockets), character encodings or documentation strings. The installation directory for these files is chosen during build time by the configure script. If the directory is moved to a different place, the `ECLDIR` environment variable should be updated accordingly. Note that the contents of the variable are parsed as a Common Lisp pathname, thus it must end with a slash.

1.4.2.2 Catching Errors and Managing Interrupts

`ECL_CATCH_ALL` [Macro]

Create a protected region.

C Macro

```
cl_env_ptr env = ecl_process_env();
ECL_CATCH_ALL_BEGIN(env) {
  /*
   * Code that is protected. Uncaught lisp conditions, THROW,
   * signals such as SIGSEGV and SIGBUS may cause jump to
   * this region.
   */
} ECL_CATCH_ALL_IF_CAUGHT {
```

```
  /*
   * If the exception, lisp condition or other control transfer
   * is caught, this code is executed.
   */
} ECL_CATCH_ALL_END
/*
 * In all cases we exit here.
 */
```

Description This is a set of three macros that create an `unwind-protect` region that prevents any nonlocal transfer of control to outer loops. In the Lisp speak, the previous code is equivalent to

```
(block nil
  (unwind-protect
     (progn
        ;; Code that is protected
)
      (return nil)))
```

As explained in `ECL_UNWIND_PROTECT`, it is normally advisable to set up an unwind-protect frame to avoid the embedded lisp code to perform arbitrary transfers of control.

See also ECL_UNWIND_PROTECT

`ECL_UNWIND_PROTECT` [Macro]

Create a protected region.

C Macro

```
cl_env_ptr env = ecl_process_env();
ECL_UNWIND_PROTECT_BEGIN(env) {
  /*
   * Code that is protected. Uncaught lisp conditions, THROW,
   * signals such as SIGSEGV and SIGBUS may cause jump to
   * this region.
   */
} ECL_UNWIND_PROTECT_EXIT {
  /*
   * If the exception, lisp condition or other control transfer
   * is caught, this code is executed. After this code, the
   * process will jump to the original destination of the
   * THROW, GOTO or other control statement that was interrupted.
   */
} ECL_UNWIND_PROTECT_END
/*
 * We only exit here if NO nonlocal jump was interrupted.
 */
```

Description When embedding ECL it is normally advisable to set up an `unwind-protect` frame to avoid the embedded lisp code to perform arbitrary

transfers of control. Furthermore, the unwind protect form will be used in at least in the following occasions:

- In a normal program exit, caused by `ext:quit`, ECL unwinds up to the outermost frame, which may be an `ECL_CATCH_ALL` or `ECL_UNWIND_PROTECT` macro.

Besides this, normal mechanisms for exit, such as `ext:quit`, and uncaught exceptions, such as serious signals (See Section 3.5.2.1 [Signals and Interrupts - Synchronous signals], page 143), are best handled using `unwind-protect` blocks.

See also ECL_CATCH_ALL

`ecl_clear_interrupts` () [Macro]

Clear all pending signals and exceptions.

Description This macro clears all pending interrupts.

See also `ecl_disable_interrupts` and `ecl_enable_interrupts`.

`ecl_disable_interrupts` () [Macro]

Postpone handling of signals and exceptions.

Description This macro sets a thread-local flag indicating that all received signals should be queued for later processing. Note that it is not possible to execute lisp code while interrupts are disabled in this way. For this purpose, use the `mp:without-interrupts` macro. Every call to `ecl_disable_interrupts` must be followed by a corresponding call to `ecl_enable_interrupts`, otherwise race conditions will appear.

See also `ecl_enable_interrupts` and `ecl_clear_interrupts`.

`ecl_enable_interrupts` (); [Macro]

Activate handling of signals and exceptions.

Description This macro sets a thread-local flag indicating that all received signals can be handled. If there are any pending signals, they will be immediately processed.

See also `ecl_disable_interrupts` and `ecl_clear_interrupts`.

`ECL_WITH_LISP_FPE` [Macro]

Execute Lisp code with correct floating point environment

Description Unless floating point exceptions are disabled (via the `--without-fpe` configure option or `ECL_OPT_TRAP_SIGFPE` runtime option), ECL will change the floating point environment when booting. This macro allows for execution of Lisp code while saving and later restoring the floating point environment of surrounding C code so that changes in the floating point environment don't leak outside.

`ECL_WITH_LISP_FPE` can be also used before ECL has booted.

Example

```
#include <ecl/ecl.h>
#include <stdio.h>

int main(int argc, char **argv) {
  ECL_WITH_LISP_FPE_BEGIN {
    cl_boot(argc, argv);
```

```
  } ECL_WITH_LISP_FPE_END;

  double a = 1.0 / 0.0;
  double b;

  ECL_WITH_LISP_FPE_BEGIN {
    cl_object form = ecl_read_from_cstring("(handler-case"
                                           "(/ 1d0 0d0)"
                                          "(division-by-zero () 0d0))");
    b = ecl_to_double(si_safe_eval(3, form, ECL_NIL, ECL_NIL));
  } ECL_WITH_LISP_FPE_END;

  printf("%g %g\n", a, b);

  cl_shutdown();
  return 0;
}
```

will output

```
inf 0
```

See also ext:trap-fpe

2 Standards

2.1 Overview

2.1.1 Reading this manual

Common Lisp users

Embeddable Common Lisp supports all Common-Lisp data types exactly as defined in the [ANSI, see [Bibliography], page 203]. All functions and macros are expected to behave as described in that document and in the HyperSpec [HyperSpec, see [Bibliography], page 203] which is the online version of [ANSI, see [Bibliography], page 203]. In other words, the Standard is the basic reference for Common Lisp and also for Embeddable Common Lisp, and this part of the manual just complements it, describing implementation-specific features such as:

- Platform dependent limits.
- Behavior which is marked as *implementation specific* in the standard.
- Some corner cases which are not described in [ANSI, see [Bibliography], page 203].
- The philosophy behind certain implementation choices, etc.

In order to aid in locating these differences, this first part of the manual copies the structure of the ANSI Common-Lisp standard, having the same number of chapters, each one with a set of sections documenting the implementation-specific details.

C/C++ programmers

The second goal of this document is to provide a reference for C programmers that want to create, manipulate and operate with Common Lisp programs at a lower level, or simply embedding Embeddable Common Lisp as a library.

The C/C++ reference evolves in parallel with the Common Lisp one, in the form of one section with the name "C Reference" for each chapter of the ANSI Common-Lisp standard. Much of what is presented in those sections is redundant with the Common Lisp specification. In particular, there is a one-to-one mapping between types and functions which should be obvious given the rules explained in the next section *C Reference*.

We must remark that the reference in this part of the manual is not enough to know how to embed Embeddable Common Lisp in a program. In practice the user or developer will also have to learn how to build programs (Section 3.1 [System building], page 89), interface with foreign libraries (Section 3.3 [Foreign Function Interface], page 106), manage memory (Section 3.6 [Memory Management], page 147), etc. These concepts are explained in a different (Section 1.4 [Embedding ECL], page 16) part of the book.

2.1.2 C Reference

One type for everything: `cl_object`

ECL is designed around the basic principle that Common Lisp already provides everything that a programmer could need, orienting itself around the creation and manipulation of

Common Lisp objects: conses, arrays, strings, characters, ... When embedding ECL there should be no need to use other C/C++ types, except when interfacing data to and from those other languages.

All Common Lisp objects are represented internally through the same C type, `cl_object`, which is either a pointer to a union type or an integer, depending on the situation. While the inner guts of this type are exposed through various headers, the user should never rely on these details but rather use the macros and functions that are listed in this manual.

There are two types of Common Lisp objects: immediate and memory allocated ones. Immediate types fit in the bits of the `cl_object` word, and do not require the garbage collector to be created. The list of such types may depend on the platform, but it includes at least the `fixnum` and `character` types.

Memory allocated types on the other hand require the use of the garbage collector to be created. ECL abstracts this from the user providing enough constructors, either in the form of Common Lisp functions (`cl_make_array`, `cl_complex`,...), or in the form of C/C++ constructors (`ecl_make_symbol`, etc).

Memory allocated types must always be kept alive so that the garbage collector does not reclaim them. This involves referencing the object from one of the places that the collector scans:

- The fields of an object (array, structure, etc) which is itself alive.
- A special variable or a constant.
- The C stack (i.e. automatic variables in a function).
- Global variables or pointers that have been registered with the garbage collector.

For memory allocation details See Section 3.6 [Memory Management], page 147. For object implementation details See Section 4.4 [Manipulating Lisp objects], page 164.

Naming conventions

As explained in the introduction, each of the chapters in the Common Lisp standard can also be implemented using C functions and types. The mapping between both languages is done using a small set of rules described below.

- Functions in the Common Lisp (`cl`) package are prefixed with the characters `cl_`, functions in the System (`si`) and Extensions (`ext`) package are prefix with `si_`, etc, etc.
- If a function takes only a fixed number of arguments, it is mapped to a C function with also a fixed number of arguments. For instance, `cos` maps to `cl_object cl_cos(cl_object)`, which takes a single Lisp object and returns a Lisp object of type `float`.
- If the function takes a variable number of arguments, its signature consists on an integer with the number of arguments and zero or more of required arguments and then a C vararg. This is the case of `cl_object cl_list(cl_narg narg, ...)`, which can be invoked without arguments, as in `cl_list(0)`, with one, `cl_list(1, a)`, etc.
- Functions return at least one value, which is either the first value output by the function, or `nil`. The extra values may be retrieved immediately after the function call using the function `ecl_nth_value`.

In addition to the Common Lisp core functions (`cl_*`), there exist functions which are devoted only to C/C++ programming, with tasks such as coercion of objects to and from C types, optimized functions, inlined macroexpansions, etc. These functions and macros typically carry the prefix `ecl_` or `ECL_` and only return one value, if any.

Most (if not all) Common Lisp functions and constructs available from C/C++ are available in "ANSI Dictionary" sections which are part of the [Chapter 2 [Standards], page 23] entries.

Only in Common Lisp

Some parts of the language are not available as C functions, even though they can be used in Common Lisp programs. These parts are either marked in the "ANSI Dictionary" sections using the tag *Only in Common Lisp*, or they are simply not mentioned (macros and special constructs). This typically happens with non-translatable constructs such as

- Common Lisp macros such as `with-open-files`
- Common Lisp special forms, such as `cond`
- Common Lisp generic functions, which cannot be written in C because of their dynamical dispatch and automatic redefinition properties.

In most of those cases there exist straightforward alternatives using the constructs and functions in ECL. For example, `unwind-protect` can be implemented using a C macro which is provided by ECL

```
cl_env_ptr env = ecl_process_env();
ECL_UNWIND_PROTECT_BEGIN(env) {
    /* protected code goes here */
} ECL_UNWIND_PROTECT_EXIT {
    /* exit code goes here */
} ECL_UNWIND_PROTECT_END;
```

Common Lisp generic functions can be directly accessed using `funcall` or `apply` and the function name, as shown in the code below

```
cl_object name = ecl_make_symbol("MY-GENERIC-FUNCTION","CL-USER");
cl_object output = cl_funcall(2, name, argument);
```

Identifying these alternatives requires some knowledge of Common Lisp, which is why it is recommended to approach the embeddable components in ECL only when there is some familiarity with the language.

2.2 Evaluation and compilation

2.2.1 Compiler declaration `optimize`

The `optimize` declaration includes three concepts: `debug`, `speed`, `safety` and `space`. Each of these declarations can take one of the integer values 0, 1, 2 and 3. According to these values, the implementation may decide how to compile or interpret a given lisp form.

ECL currently does not use all these declarations, but some of them definitely affect the speed and behavior of compiled functions. For instance, the `debug` declaration, as shown in Table 2.1, the value of debugging is zero, the function will not appear in the debugger and, if redefined, some functions might not see the redefinition.

Behavior	**0**	**1**	**2**	**3**
Compiled functions in the same source file are called directly	Y	Y	N	N
Compiled function appears in debugger backtrace	N	N	Y	Y
All functions get a global entry (SI:C-LOCAL is ignored)	N	N	Y	Y

Table 2.1: Behavior for different levels of `debug`

A bit more critical is the value of `safety` because as shown in Table 2.2, it may affect the safety checks generated by the compiler. In particular, in some circumstances the compiler may assume that the arguments to a function are properly typed. For instance, if you compile with a low value of `safety`, and invoke `rplaca` with an object which is not a list, the consequences are unspecified.

Behavior	**0**	**1**	**2**	**3**
The compiler generates type checks for the arguments of a lambda form, thus enforcing any type declaration written by the user.	N	Y	Y	Y
The value of an expression or a variable declared by the user is assumed to be right.	Y	Y	N	N
We believe type declarations and type inference and, if the type of a form is inferred to be right for a function, slot accessor, etc, this may be inlined. Affects functions like `car`, `cdr`, etc	Y	Y	N	N
We believe types defined before compiling a file do not change before the compiled code is loaded.	Y	Y	N	N
Arguments in a lisp form are assumed to have the appropriate types so that the form will not fail.	Y	N	N	N
The slots or fields in a lisp object are accessed directly without type checks even if the type of the object could not be inferred (see line above). Affects functions like `pathname-type`, `car`, `rest`, etc.	Y	N	N	N

Table 2.2: Behavior for different levels of `safety`

2.2.2 `declaim` and `proclaim`

Declarations established with `proclaim` stay in force indefinitely. Declarations established with `declaim` in a file do not persist after the file has been compiled. However, they are established with `proclaim` at load time when the compiled file is loaded. This means that when compiling two files, `declaim` declarations in the first file will not be in force when compiling the second file unless the first file was loaded before the second one was compiled.

2.2.3 C Reference

`cl_env_ptr ecl_process_env ()` [C/C++ identifier]

ECL stores information about each thread on a dedicated structure, which is the process environment. A pointer to this structure can be retrieved using the function or macro above. This pointer can be used for a variety of tasks, such as defining special variable bindings, controlling interrupts, retrieving function output values, etc.

2.2.3.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
compile	[Only in Common Lisp]
macroexpand	cl_object cl_macroexpand(cl_narg narg, cl_object form, ...)
macroexpand-1	cl_object cl_macroexpand_1(cl_narg narg, cl_object form, ...)
proclaim	[Only in Common Lisp]
special-operator-p	cl_object cl_special_operator_p(cl_object form)
constantp	cl_object cl_constantp (cl_narg narg, cl_object arg, ...)

2.3 Types and classes

ECL defines the following additional built-in classes in the `cl` package:

- `single-float`
- `double-float`
- `long-float`

2.3.1 C Reference

2.3.1.1 ANSI Dictionary

Common Lisp and C equivalence

Synopsis

Lisp symbol	**C function**
coerce	cl_object cl_coerce(cl_object object, cl_object result_type)
subtypep	cl_object cl_subtypep(cl_narg narg, cl_object type1, cl_object type2, ...)
type-of	cl_object cl_type_of(cl_object object)
typep	cl_object cl_typep(cl_narg narg, cl_object object, cl_object type_specifier, ...)
type-error-datum	[Only in Common Lisp]
type-error-expected-type	[Only in Common Lisp]

2.4 Data and control flow

2.4.1 Shadowed bindings

ANSI doesn't specify what should happen if any of the `let`, `flet` and `labels` special operators contain many bindings sharing the same name. Because the behavior varies between the implementations and the programmer can't rely on the spec ECL signals an error if such situation occur.

Moreover, while ANSI defines lambda list parameters in the terms of `let*`, when used in function context programmer can't provide an initialization forms for required parameters. If required parameters share the same name an error is signaled.

Described behavior is present in ECL since version 16.0.0. Previously the `let` operator were using first binding. Both `flet` and `labels` were signaling an error if C compiler was used and used the last binding as a visible one when the byte compiler was used.

2.4.2 Minimal compilation

Former versions of ECL, as well as many other lisps, used linked lists to represent code. Executing code thus meant traversing these lists and performing code transformations, such as macro expansion, every time that a statement was to be executed. The result was a slow and memory hungry interpreter.

Beginning with version 0.3, ECL was shipped with a bytecodes compiler and interpreter which circumvent the limitations of linked lists. When you enter code at the lisp prompt, or when you load a source file, ECL begins a process known as minimal compilation. Barely this process consists on parsing each form, macroexpanding it and translating it into an intermediate language made of bytecodes.

The bytecodes compiler is implemented in src/c/compiler.d. The main entry point is the lisp function si::make-lambda, which takes a name for the function and the body of the lambda lists, and produces a lisp object that can be invoked. For instance,

```
> (defvar fun (si::make-lambda 'f '((x) (1+ x))))
*FUN*
> (funcall fun 2)
3
```

ECL can only execute bytecodes. When a list is passed to `eval` it must be first compiled to bytecodes and, if the process succeeds, the resulting bytecodes are passed to the interpreter. Similarly, every time a function object is created, such as in `defun` or `defmacro`, the compiler processes the lambda form to produce a suitable bytecodes object.

The fact that ECL performs this eager compilation means that changes on a macro are not immediately seen in code which was already compiled. This has subtle implications. Take the following code:

```
> (defmacro f (a b) `(+ ,a ,b))
F
> (defun g (x y) (f x y))
G
> (g 1 2)
3
> (defmacro f (a b) `(- ,a ,b))
F
> (g 1 2)
```

```
3
```

The last statement always outputs 3 while in former implementations based on simple list traversal it would produce -1.

2.4.3 Function types

Functions in ECL can be of two types: they are either compiled to bytecodes or they have been compiled to machine code using a lisp to C translator and a C compiler. To the first category belong function loaded from lisp source files or entered at the toplevel. To the second category belong all functions in the ECL core environment and functions in files processed by `compile` or `compile-file`.

The output of `(symbol-function fun)` is one of the following:

- a function object denoting the definition of the function `fun`,
- a list of the form `(macro . function-object)` when `fun` denotes a macro,
- or simply `'special`, when `fun` denotes a special form, such as `block`, `if`, etc.

ECL usually keeps the source code of a function unless the global variable `si:*keep-definitions*` was false when the function was translated into bytecodes. Therefore, if you don't need to use compile and disassemble on defined functions, you should issue `(setq si:*keep-definitions* nil)` at the beginning of your session.

`si:*keep-definitions*` [Variable]

If set to `t` ECL will preserve the compiled function source code for disassembly and recompilation.

In Table 2.3 we list all Common Lisp values related to the limits of functions.

call-arguments-limit	`65536`
lambda-parameters-limit	`call-arguments-limit`
multiple-values-limit	`64`
lambda-list-keywords	`(&optional &rest &key &allow-other-keys &aux &whole &environment &body)`

Table 2.3: Function related constants

2.4.4 C Calling conventions

ECL is implemented using either a C or a C++ compiler. This is not a limiting factor, but imposes some constraints on how these languages are used to implement functions, multiple values, closures, etc. In particular, while C functions can be called with a variable number of arguments, there is no facility to check how many values were actually passed. This forces us to have two types of functions in ECL

- Functions that take a fixed number of arguments have a simple C signature, with all arguments being properly declared, as in `cl_object cl_not(cl_object arg1)`.
- Functions with a variable number of arguments, such as those accepting `&optional`, `&rest` or `&key` arguments, must take as first argument the number of remaining ones,

as in `cl_object cl_list(cl_narg narg, ...)`. Here *narg* is the number of supplied arguments.

The previous conventions set some burden on the C programmer that calls ECL, for she must know the type of function that is being called and supply the right number of arguments. This burden disappears for Common Lisp programmers, though.

As an example let us assume that the user wants to invoke two functions which are part of the ANSI [ANSI, see [Bibliography], page 203] standard and thus are exported with a C name. The first example is `cl_cos`, which takes just one argument and has a signature `cl_object cl_cos(cl_object)`.

```
#include <math.h>
...
cl_object angle = ecl_make_double_float(M_PI);
cl_object c = cl_cos(angle);
printf("\nThe cosine of PI is %g\n", ecl_double_float(c));
```

The second example also involves some Mathematics, but now we are going to use the C function corresponding to `+`. As described in Section 2.10.7.5 [Numbers - ANSI dictionary], page 47, the C name for the plus operator is `cl_P` and has a signature `cl_object cl_P(cl_narg narg,...)`. Our example now reads as follows

```
cl_object one = ecl_make_fixnum(1);
cl_object two = cl_P(2, one, one);
cl_object three = cl_P(3, one, one, one);
printf("\n1 + 1 is %d\n", ecl_fixnum(two));
printf("\n1 + 1 + 1 is %d\n", ecl_fixnum(three));
```

Note that most Common Lisp functions will not have a C name. In this case one must use the symbol that names them to actually call the functions, using `cl_funcall` or `cl_apply`. The previous examples may thus be rewritten as follows

```
/* Symbol + in package CL */
cl_object plus = ecl_make_symbol("+","CL");
cl_object one = ecl_make_fixnum(1);
cl_object two = cl_funcall(3, plus, one, one);
cl_object three = cl_funcall(4, plus, one, one, one);
printf("\n1 + 1 is %d\n", ecl_fixnum(two));
printf("\n1 + 1 + 1 is %d\n", ecl_fixnum(three));
```

Another restriction of C and C++ is that functions can only take a limited number of arguments. In order to cope with this problem, ECL uses an internal stack to pass any argument above a hardcoded limit, `ECL_C_CALL_ARGUMENTS_LIMIT`, which is as of this writing 63. The use of this stack is transparently handled by the Common Lisp functions, such as `apply`, `funcall` and their C equivalents, and also by a set of macros, `cl_va_arg`, which can be used for coding functions that take an arbitrary name of arguments.

2.4.5 C Reference

`void ecl_bds_bind` (*cl_env_ptr cl_env, cl_object var, cl_object value*); [Function]

`void ecl_bds_push` (*cl_env_ptr cl_env, cl_object var*); [Function]

Bind a special variable

Description Establishes a variable binding for the symbol *var* in the Common Lisp environment *env*, assigning it *value*.

This macro or function is the equivalent of let* and let.

`ecl_bds_push` does a similar thing, but reuses the old value of the same variable. It is thus the equivalent of `(let ((var var)) ...)`

Every variable binding must undone when no longer needed. It is best practice to match each call to `ecl_bds_bind` by another call to `ecl_bds_unwind1` in the same function.

`void ecl_bds_unwind1` (*cl_env_ptr cl_env*); [Function]
`void ecl_bds_unwind_n` (*cl_env_ptr cl_env, int n*); [Function]

Undo one variable binding

Description `ecl_bds_unwind1` undoes the outermost variable binding, restoring the original value of the symbol in the process.

`ecl_bds_unwind_n` does the same, but for the *n* last variables.

Every variable binding must undone when no longer needed. It is best practice to match each call to `ecl_bds_bind` by another call to `ecl_bds_unwind1` in the same function.

`cl_object ecl_setq` (*cl_env_ptr cl_env, cl_object var, cl_object value*); [Function]

C equivalent of setq

Description Assigns *value* to the special variable denoted by the symbol *var*, in the Common Lisp environment *cl_env*.

This function implements a variable assignment, not a variable binding. It is thus the equivalent of setq.

`cl_object ecl_symbol_value` (*cl_object var*); [Function]

Description Retrieves the value of the special variable or constant denoted by the symbol *var*, in the Common Lisp environment *cl_env*.

This function implements the equivalent of symbol-value and works both on special variables and constants.

If the symbol is not bound, an error is signaled.

`typedef struct { ... } ecl_va_list[1];` [Macro]

`ecl_va_start` (*ecl_va_list arglist, last_argument, narg, n_ordinary*); [Macro]

`cl_object ecl_va_arg` (*ecl_va_list arglist*); [Macro]

`cl_object ecl_va_end` (*ecl_va_list arglist*); [Macro]

Accepting a variable number of arguments

Description The macros above are used to code a function that accepts an arbitrary number of arguments. We will describe them in a practical example

```
cl_object my_plus(cl_narg narg, cl_object required1, ...)
{
    cl_env_ptr env = ecl_process_env();
```

```
        cl_object other_value;
        ecl_va_list varargs;
        ecl_va_start(varargs, required1, narg, 1);
        while (narg > 1) {
            cl_object other_value = ecl_va_arg(varargs);
            required1 = ecl_plus(required1, other_value);
        }
        ecl_va_end(varargs);
        ecl_return1(env, required1);
    }
```

The first thing to do is to declare the variable that will hold the arguments. This is *varargs* in our example and it has the type `ecl_va_list`.

This arguments list is initialized with the `ecl_va_start` macro, based on the supplied number of arguments, *narg*, the number of required arguments which are passed as ordinary C arguments (1 in this case), the last such ordinary arguments, *required*, and the buffer for the argument list, *varargs*.

Once *varargs* has been initialized, we can retrieve these values one by one using `ecl_va_arg`. Note that the returned value always has the type `cl_object`, for it is always a Common Lisp object.

The last statement before returning the output of the function is `ecl_va_end`. This macro performs any required cleanup and should never be omitted.

`cl_object ecl_nvalues` (*cl_env_ptr env*); [Function]
`cl_object ecl_nth_value` (*cl_env_ptr env, int n*); [Function]

Accessing output values

Description Common Lisp functions may return zero, one or more values. In ECL, the first two cases do not require any special manipulation, as the C function returns either `nil` or the first (zeroth) value directly. However, if one wishes to access additional values from a function, one needs to use these two macros or functions

- `ecl_nvalues(env)` returns the number of values that the function actually outputs. The single argument is the lisp environment. This value is larger or equal to 0 and smaller than `ECL_MULTIPLE_VALUES_LIMIT`.
- Once we know the number of return values, they can be directly accessed using the function `ecl_nth_value(env,n)`, where *n* is a number larger than or equal to 1, and smaller than `ECL_MULTIPLE_VALUES_LIMIT`, which must correspond to a valid output value. No checking is done.

Note that in both cases these macros and functions have to be used right after the Lisp function was called. This is so because other Lisp functions might destroy the content of the return stack.

Example A C/C++ excerpt:

```
cl_env_ptr env = ecl_process_env();
cl_object a = ecl_make_fixnum(13);
cl_object b = ecl_make_fixnum(6);
cl_object modulus = cl_floor(2, a, b);
cl_object remainder = ecl_nth_value(env, 1);
```

The somewhat equivalent Common Lisp code:

```
(multiple-value-bind (modulus equivalent)
    (floor 13 6))
```

`ecl_return0` (*cl_env_ptr cl_env*); [Macro]
`ecl_return1` (*cl_env_ptr cl_env*, *cl_object value1*); [Macro]
`ecl_return2` (*cl_env_ptr cl_env*, *cl_object value1*, *cl_object value2*); [Macro]
`ecl_return3` (*cl_env_ptr cl_env*, *cl_object value1*, *cl_object value2*, *cl_object value3*); [Macro]

Returning multiple values

Description Returns *N* values from a C/C++ function in a way that a Common Lisp function can recognize and use them. The 0-th value is returned directly, while values 1 to N are stored in the Common Lisp environment *cl_env*. This macro has to be used from a function which returns an object of type `cl_object`.

`ECL_BLOCK_BEGIN` [Macro]

```
ECL_BLOCK_BEGIN(env,code) {

} ECL_BLOCK_END;
```

Description `ECL_BLOCK_BEGIN` establishes a block named *code* that becomes visible for the Common Lisp code. This block can be used then as a target for `cl_return`.

env must be the value of the current Common Lisp environment, obtained with `ecl_process_env`.

The C/C++ program has to ensure that the code in `ECL_BLOCK_END` gets executed, avoiding a direct exit of the block via `goto` or a C/C++ return.

`ECL_CATCH_BEGIN` [Macro]

```
ECL_CATCH_BEGIN(env,tag) {

} ECL_CATCH_END;
```

Description `ECL_CATCH_BEGIN` establishes a destination for `throw` with the code given by *tag*.

env must be the value of the current Common Lisp environment, obtained with `ecl_process_env`.

The C/C++ program has to ensure that the code in `ECL_CATCH_END` gets executed, avoiding a direct exit of the catch block via goto or a C/C++ return.

`ECL_UNWIND_PROTECT_BEGIN` [Macro]

C macro for unwind-protect

Synopsis

```
ECL_UNWIND_PROTECT_BEGIN(env) {

} ECL_UNWIND_PROTECT_EXIT {

} ECL_UNWIND_PROTECT_END;
```

Description `ECL_UNWIND_PROTECT_BEGIN` establishes two blocks of C code that work like the equivalent ones in Common Lisp: a protected block, contained between the "BEGIN" and the "EXIT" statement, and the exit block, appearing immediately afterwards. The form guarantees that the exit block is always executed, even if the protected block attempts to exit via some nonlocal jump construct (`throw`, `return`, etc).

env must be the value of the current Common Lisp environment, obtained with `ecl_process_env`.

The utility of this construct is limited, for it only protects against nonlocal exits caused by Common Lisp constructs: it does not interfere with C `goto`, `return` or with C++ exceptions.

2.4.5.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function or constant**
apply	cl_object cl_apply(cl_narg narg, cl_object function, ...)
call-arguments-limit	ECL_CALL_ARGUMENTS_LIMIT
compiled-function-p	cl_object cl_compiled_function_p(cl_object object)
complement	cl_object cl_complement(cl_object function)
constantly	cl_object cl_constantly(cl_object value)
every	cl_object cl_every(cl_narg narg, cl_object predicate, ...)
eq	cl_object cl_eq(cl_object x, cl_object y)
eql	cl_object cl_eql(cl_object x, cl_object y)
equal	cl_object cl_equal(cl_object x, cl_object y)
equalp	cl_object cl_equalp(cl_object x, cl_object y)
fboundp	cl_object cl_fboundp(cl_object function_name)
fdefinition	cl_object cl_fdefinition(cl_object function_name)
(setf fdefinition)	cl_object si_fset(cl_narg narg, cl_object function_name, cl_object definition, ...)
fmakunbound	cl_object cl_fmakunbound(cl_object function_name)
funcall	cl_object cl_funcall(cl_narg narg, cl_object function, ...)
function-lambda-expression	cl_object cl_function_lambda_expression(cl_object function)
functionp	cl_object cl_functionp(cl_object object)
get-setf-expansion	cl_object cl_get_setf_expansion(cl_narg narg, cl_object place, ...)
identity	cl_object cl_identity(cl_object x)
let, let*	cl_object ecl_bds_bind(cl_env_ptr env, cl_object symbol, cl_object value)
lambda-parameters-limit	ECL_LAMBDA_PARAMETERS_LIMIT
multiple-values-limit	ECL_MULTIPLE_VALUES_LIMIT
not	cl_object cl_not(cl_object object)
notevery	cl_object cl_notevery(cl_narg narg, cl_object predicate, ...)
notany	cl_object cl_notany(cl_narg narg, cl_object predicate, ...)
set	cl_object cl_set(cl_object symbol, cl_object value)

setq	cl_object ecl_setq(cl_env_ptr env, cl_object symbol, cl_object value)
symbol-value	cl_object ecl_symbol_value(cl_env_ptr env, cl_object symbol)
some	cl_object cl_some(cl_narg narg, cl_object predicate, ...)
values-list	cl_object cl_values_list(cl_object list)

2.5 Objects

2.5.1 C Reference

2.5.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
function-keywords	[Only in Common Lisp]
ensure-generic-function	cl_object cl_ensure_generic_function(cl_narg narg, cl_object function_name, ...)
allocate-instance	[Only in Common Lisp]
reinitialize-instance	[Only in Common Lisp]
shared-initialize	[Only in Common Lisp]
update-instance-for-different-class	[Only in Common Lisp]
update-instance-for-redefined-class	[Only in Common Lisp]
change-class	[Only in Common Lisp]
slot-boundp	cl_object cl_slot_boundp(cl_object instance, cl_object slot_name)
slot-exists-p	cl_object cl_slot_exists_p(cl_object instance, cl_object slot_name)
slot-makunbound	cl_object cl_slot_makunbound(cl_object instance, cl_object slot_name)
slot-missing	[Only in Common Lisp]
slot-unbound	[Only in Common Lisp]
slot-value	cl_object cl_slot_value(cl_object instance, cl_object slot_name)
method-qualifiers	[Only in Common Lisp]
no-applicable-method	[Only in Common Lisp]
no-next-method	[Only in Common Lisp]
remove-method	[Only in Common Lisp]
make-instance	[Only in Common Lisp]
make-instances-obsolete	[Only in Common Lisp]
make-load-form	[Only in Common Lisp]
make-load-form-saving-slots	cl_object cl_make_load_form_saving_slots(cl_narg narg, cl_object object, ...)
find-class	cl_object cl_find_class(cl_narg narg, cl_object symbol, ...)
compute-applicable-methods	[Only in Common Lisp]
find-method	[Only in Common Lisp]
add-method	[Only in Common Lisp]

initialize-instance	[Only in Common Lisp]
class-name	[Only in Common Lisp]
(setf class-name)	[Only in Common Lisp]
class-of	cl_object cl_class_of(cl_object object)
unbound-slot-instance	[Only in Common Lisp]

2.6 Structures

2.6.1 Redefining a defstruct structure

ANSI Common-Lisp says that consequences of redefining a `defstruct` are undefined. ECL defines this behavior to signal an error if the new structure is not compatible. Structures are incompatible when:

They have a different number of slots
: This is particularly important for other structures which could have included the current one and for already defined instances.

Slot name, type or offset is different
: Binary compatibility between old and new instances.

2.6.2 C Reference

2.6.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
copy-structure	cl_object cl_copy_structure(cl_object structure)

2.7 Conditions

2.7.1 C Reference

`ECL_HANDLER_CASE` [Macro]

C macro for handler-case

Synopsis

```
ECL_HANDLER_CASE_BEGIN(env,names) {

} ECL_HANDLER_CASE(n,condition) { {

} ECL_HANDLER_CASE_END;
```

Description `ECL_HANDLER_CASE_BEGIN` runs a block of C code with a set of error handlers bound to the names given by the list *names*. The subsequent `ECL_HANDLER_CASE` statements specify what to do when the *n*-th type of conditions is found, where *n* is an integer denoting the position of the name in the list *names*.

When a condition is signaled, ECL scans the list of signal handlers, looking for matches based on `typep`. If the match with the highest precedence belongs to the list *names*,

ECL will perform a non-local transfer of control to the appropriate `ECL_HANDLER_CASE`, passing it a *condition* object as unique argument.

The following example shows how to establish a handler for `error` conditions. Note how the first value to `ECL_HANDLER_CASE` matches the position of the condition name in the list:

```
cl_object error = ecl_make_symbol("ERROR","CL");
ECL_HANDLER_CASE_BEGIN(the_env, ecl_list1(error)) {
   /* This form is evaluated with bound handlers */
   output = cl_eval(1, form);
} ECL_HANDLER_CASE(1, condition) {
   /* This code is executed when an error happens */
   /* We just return the error that took place */
   output = condition;
} ECL_HANDLER_CASE_END;
```

`ECL_RESTART_CASE` [Macro]

C macro for restart-case

Synopsis

```
ECL_RESTART_CASE_BEGIN(env,names) {

} ECL_RESTART_CASE(n,args) { {

} ECL_RESTART_CASE_END;
```

Description `ECL_RESTART_CASE_BEGIN` runs a block of C code with a set of restarts bound to the names given by the list *names*. The subsequent `ECL_RESTART_CASE` statements specify what to do when the *n*-th restart is invoked, where *n* is an integer denoting the position of the name in the list *names*.

When the restart is invoked, it can receive any number of arguments, which are grouped in a list and stored in a new variable created with the name *args*.

The following example shows how to establish an *abort* and a *use-value* restart. Note how the first value to `ECL_RESTART_CASE` matches the position of the restart name in the list:

```
cl_object abort = ecl_make_symbol("ABORT","CL");
cl_object use_value = ecl_make_symbol("USE-VALUE","CL");
ECL_RESTART_CASE_BEGIN(the_env, cl_list(2, abort, use_value)) {
   /* This form is evaluated with bound restarts */
   output = cl_eval(1, form);
} ECL_RESTART_CASE(1, args) {
   /* This code is executed when the 1st restart (ABORT) is invoked */
   output = ECL_NIL;
} ECL_RESTART_CASE(2, args) {
   /* This code is executed when the 2nd restart (USE-VALUE) is invoked */
   output = ECL_CAR(args);
} ECL_RESTART_CASE_END;
```

2.7.1.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
abort	cl_object cl_abort(cl_narg narg, ...)
break	[Only in Common Lisp]
cell-error-name	[Only in Common Lisp]
cerror	cl_object cl_cerror(cl_narg narg, cl_object continue_format_control, cl_object datum, ...)
compute-restarts	cl_object cl_compute_restarts(cl_narg narg, ...)
continue	cl_object cl_continue(cl_narg narg, ...)
error	cl_object cl_error(cl_narg narg, cl_object datum, ...)
find-restart	cl_object cl_find_restart(cl_narg narg, cl_object identifier, ...)
handler-case	ECL_HANDLER_CASE macro
invalid-method-error	cl_object cl_invalid_method_error (cl_narg narg, cl_object method, cl_object format, ...)
invoke-debugger	[Only in Common Lisp]
invoke-restart	cl_object cl_invoke_restart(cl_narg narg, cl_object restart, ...)
invoke-restart-interactively	cl_object cl_invoke_restart_interactively(cl_object restart)
make-condition	cl_make_condition(cl_narg narg, cl_object type, ...)
method-combination-error	cl_object cl_method_combination_error(cl_narg narg, cl_object format, ...)
muffle-warning	cl_object cl_muffle_warning(cl_narg narg, ...)
restart-name	[Only in Common Lisp]
restart-case	ECL_RESTART_CASE macro
signal	[Only in Common Lisp]
simple-condition-format-control	[Only in Common Lisp]
simple-condition-format-arguments	[Only in Common Lisp]
store-value	cl_object cl_store_value(cl_narg narg, ...)
use-value	cl_object cl_use_value(cl_narg narg, ...)
warn	[Only in Common Lisp]

2.8 Symbols

There are no implementation-specific limits on the size or content of symbol names. It is however not allowed to write on the strings which have been passed to `#'make-symbol` or returned from `#'symbol-name`.

2.8.1 C Reference

`cl_object ecl_make_keyword` (*char *name*); [Function]

Find a lisp keyword

Description

Many Lisp functions take keyword arguments. When invoking a function with keyword arguments we need keywords, which are a kind of symbols that live in the `keyword` package. This function does the task of finding or creating those keywords from C strings.

- It is usually safe to store the resulting pointer, because keywords are always referenced by their package and will not be garbage collected (unless of course, you decide to delete it).
- Remember that the case of the string is significant. `ecl_make_keyword("TO")` with return `:TO`, while `ecl_make_keyword("to")` returns a completely different keyword, `:|to|`. In short, you usually want to use uppercase.

Example The following example converts a section of a string to uppercase characters:

```
cl_object start = ecl_make_keyword("START");
cl_object end = ecl_make_keyword("END");
...
sup = cl_string_upcase(4, s, start, ecl_make_fixnum(2),
                       end, ecl_make_fixnum(6));
```

`cl_object ecl_make_symbol` (*const char *name, const char *package_name*); [Function]

Find a lisp symbol

Description This function finds or create a symbol in the given package. First of all, it tries to find the package named by *package_name*. If it does not exist, an error is signaled. Then, a symbol with the supplied *name* is created and interned in the given package.

2.8.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
boundp	cl_object cl_boundp(cl_object symbolp)
copy-symbol	cl_object cl_copy_symbol(cl_narg narg, cl_object symbol, ...)
get	cl_object cl_get(cl_narg narg, cl_object sym, cl_object indicator, ...)
gensym	cl_object cl_gensym(cl_narg narg, ...)
gentemp	cl_object cl_gentemp(cl_narg narg, ...)
keywordp	cl_object cl_keywordp(cl_object object)
make-symbol	cl_object cl_make_symbol(cl_object name)
makunbound	cl_object cl_makunbound(cl_object makunbound)
remprop	cl_object cl_remprop(cl_object symbol, cl_object indicator)
set	cl_object cl_set(cl_object symbol, cl_object value)
symbolp	cl_object cl_symbolp(cl_object object)
symbol-function	cl_object cl_symbol_function(cl_object symbol)
(setf symbol-function)	cl_object si_fset(cl_narg narg, cl_object function_name, cl_object definition, ...)
symbol-name	cl_object cl_symbol_name(cl_object symbol)
symbol-package	cl_object cl_symbol_package(cl_object symbol)
symbol-plist	cl_object cl_symbol_plist(cl_object symbol)
(setf symbol-plist)	cl_object si_set_symbol_plist(cl_object symbol, cl_object plist)
symbol-value	cl_object cl_symbol_value(cl_object symbol)

2.9 Packages

In Table 2.4 we list all packages available in ECL. The nicknames are aliases for a package. Thus, `system:symbol` may be written as `sys:symbol` or `si:symbol`. The module field explains which library provides what package. For instance, the ASDF package is obtained when loading the ASDF library with `(require 'asdf)`.

Name	Nickname	In module	Description
COMMON-LISP	CL	ECL core	Main Common Lisp package.
COMMON-LISP-USER	CL-USER	ECL core	User package.
CLOS	MOP	ECL core	Symbols from the AMOP.
EXT		ECL core	ECL extensions to the language & library.
SYSTEM	SI, SYS	ECL core	Functions and variables internal to the implementation. Never to be used.
FFI		ECL core	Foreign function interface
CMP	C	CMP	The compiler
SB-BSD-SOCKETS		SOCKETS	Sockets library compatible with SBCL's
SB-RT	RT, REGRESSION-TEST	RT	Test units (customized for ECL)
ASDF		ASDF	System definition file with ECL customizations.

Table 2.4: ECL packages

2.9.1 C Reference

2.9.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
export	cl_object cl_export(cl_narg narg, cl_object symbols, ...)
find-symbol	cl_object cl_find_symbol(cl_narg narg, cl_object string, ...)
find-package	cl_object cl_find_package(cl_object name)
find-all-symbols	cl_object cl_find_all_symbols(cl_object string)
import	cl_object cl_import(cl_narg narg, cl_object symbols, ...)
list-all-packages	cl_object cl_list_all_packages(void)
rename-package	cl_object cl_rename_package(cl_narg narg, cl_object package, cl_object new_name, ...)

shadow	cl_object cl_shadow(cl_narg narg, cl_object symbols, ...)
shadowing-import	cl_object cl_shadowing_import(cl_narg narg, cl_object symbols, ...)
delete-package	cl_object cl_delete_package(cl_object package)
make-package	cl_object cl_make_package(cl_narg narg, cl_object package_name, ...)
unexport	cl_object cl_unexport(cl_narg narg, cl_object symbols, ...)
unintern	cl_object cl_unintern(cl_narg narg, cl_object symbol, ...)
unuse-package	cl_object cl_unuse_package(cl_narg narg, cl_object package, ...)
use-package	cl_object cl_use_package(cl_narg narg, cl_object package, ...)
intern	cl_object cl_intern(cl_narg narg, cl_object string, ...)
package-name	cl_object cl_package_name(cl_object package)
package-nicknames	cl_object cl_package_nicknames(cl_object package)
package-shadowing-symbols	cl_object cl_package_shadowing_symbols(cl_object package)
package-use-list	cl_object cl_package_use_list(cl_object package)
package-used-by-list	cl_object cl_package_used_by_list(cl_object package)
packagep	cl_object cl_packagep(cl_object object)
package-error-package	[Only in Common Lisp]

2.10 Numbers

2.10.1 Numeric types

ECL supports all of the Common Lisp numeric tower, which is shown in Table 2.5. The details, however, depend both on the platform on which ECL runs and on the configuration which was chosen when building ECL.

Type	**Description**
fixnum	Signed integer with a number of bits given by ext:fixnum-bits, fit in a machine word.
bignum	Arbitrary size integers, only limited by amount of memory.
ratio	Arbitrary size rational number, made up of two integers.
short-float	Equivalent to single-float.
single-float	32-bits IEEE floating point number.
double-float	64-bits IEEE floating point number.
long-float	Either equivalent to double-float, or a 96/128 bits IEEE floating point number (long double in C/C++).
rational	An alias for (or integer ratio)
float	An alias for (or single-float double-float short-float long-float)
real	An alias for (or rational float)
complex	Complex number made of two real numbers of the above mentioned types or a <float> _Complex type in C99.

Table 2.5: Numeric types in ECL

In general, the size of a `fixnum` is determined by the word size of a machine, which ranges from 32 to 64 bits. Integers larger than this are implemented using the GNU Multiprecision library (`https://gmplib.org/`). Rationals are implemented using two integers, without caring whether they are fixnum or not. Floating point numbers include at least the two IEEE types of 32 and 64 bits respectively.

In machines where it is supported, it is possible to associate the lisp `long-float` with the machine type `long double` whose size ranges from 96 to 128 bits, and which are a bit slower.

In machines where a type `<float> _Complex` is supported numbers of type `(complex float)` are implemented with it, otherwise like ratios all complex numbers are pairs of numbers.

2.10.2 Floating point exceptions

ECL supports two ways of dealing with special floating point values, such as Not a Number (NaN), infinity or denormalized floats, which can occur in floating point computations. Either a condition is signaled or the value is silently used as it is. There are multiple options controlling which behaviour is selected: If ECL is built with the `--without-ieee-fp` configure option, then a condition is signaled for every infinity or NaN encountered. If not, floating point exceptions can be disabled at build time using the `--without-fpe` configure option. Otherwise, if both `--with-ieee-fp` and `--with-fpe` options are on, by default, a condition is signaled for invalid operation, division by zero and floating point overflows. This can be changed at runtime by using `ext:trap-fpe`. If the `ECL_OPT_TRAP_SIGFPE` boot option is false (see Table 1.1), no conditions are signaled by default (Note that in this case, if you enable trapping of floating point exceptions with `ext:trap-fpe`, then you have to install your own signal handler).

`ext:trap-fpe` *condition flag* [Function]

Control the signaling of the floating point exceptions

Synopsis

condition a symbol - one of `cl:last`, `cl:t`, `cl:division-by-zero`, `cl:floating-point-overflow`, `cl:floating-point-underflow`, `cl:floating-point-invalid-operation`, `cl:floating-point-inexact` or an integer.

flag a generalized boolean

Description If *condition* is `last`, *flag* is ignored and the currently enabled floating point exceptions are returned in an implementation depended format (currently an integer). Otherwise, *flag* determines whether the current thread will signal a floating point exception for the conditions passed in *condition*. *condition* can be either a symbol denoting a single condition, `t` for all conditions that are enabled by default or a value obtained from an earlier call to `ext:trap-fpe` with `last`.

See also ECL_WITH_LISP_FPE

2.10.3 Random-States

ECL relies internally on a 32-bit Mersenne-Twister random number generator, using a relatively large buffer to precompute about 5000 pseudo-random bytes. This implies also that random states can be printed readably and also read, using the `#$` macro. There is

no provision to create random states from user arrays, though. Random state is printed unreadably by default.

The `#$` macro can be used to initialize the generator with a random seed (an integer), an array of random bytes (mainly used for reading back printed random-state) and another random-state (syntactic sugar for copying the random-state).

2.10.4 Infinity and Not a Number

The ANSI Common-Lisp standard does not specify the behaviour of numeric functions for infinite or not number valued floating point numbers. If ECL is configured to support these special values (see the `--with-ieee-fp` configure option) and floating point exceptions are disabled, numeric functions generally return the same value as the corresponding C function. This means, that the output will be a NaN for a NaN input, and the "mathematically correct" value (which may be NaN, e.g. for ∞-∞) for an infinite real input. For complex floats, however, the return value of a numeric function called with a complex number for which the real or imaginary part is infinite, is undefined[1].

For other functions dealing with numbers, we adopt the following behaviour:

Comparison functions
: All numeric comparisons with `=,<,<=,>,>=` involving NaN return false. Comparing two NaNs of the same type with `eql` returns true.

`min/max`
: NaN values are ignored, i.e. the maximum/minimum is taken only over the number valued parameters.

Rounding functions
: All rounding functions signal an `arithmetic-error` if any of the given parameters are not number valued or infinite.

2.10.5 Branch cuts and signed zero

For multi-valued complex functions like `asin`, `acos`, etc. the ANSI Common-Lisp standard specifies in detail the location of the branch cuts and in particular the value of these functions on the branch cuts. ECL diverges from the standard in that the sign of zero distinguishes two sides of a branch cut. For example, the `asin` function includes a branch cut running from 1 to ∞ on the real axis. The ANSI standard specifies a negative imaginary part for `asin` on this branch cut consistent with approaching the cut from below. Evaluating `(asin z)` in ECL, on the other hand, returns a result with positive imaginary part if the imaginary part of `z` is `+0.0` (consistent with approaching the cut from above) and a result with negative imaginary part if the imaginary part of `z` is `-0.0` (consistent with approaching the cut from below)[2]. This applies to `sqrt`, `asin`, `acos`, `atan`, `asinh`, `acosh` and `atanh`.

---

[1] The main reason for this is that some numeric functions for C complex numbers return mathematically incorrect values, for example sinh(i*∞) returns i*NaN instead of the mathematically correct i*∞. Keeping this consistent with our own implementation of complex arithmetic that is used when C complex numbers are not available would require to much work. Furthermore, complex arithmetic with infinities is unreliable anyway, since it quickly leads to NaN values (consider i*∞ = (0+i*1)*(∞+i*0) = NaN+i*∞; even this simple example is already mathematically incorrect).

[2] The reason for this behaviour is twofold: first, the approach taken by ECL is mathematically more sensible in a number system with signed zero and second, it is consistent with the specification of multi-valued complex functions in the C programming language.

2.10.6 Dictionary

`ext:{short,single,double,long}-float-{positive,negative}-infinity` [Constant]

Constant positive/negative infinity for the different floating point types.

`ext:nan` [Function]

Returns a double float NaN value. Coerce to other floating point types to get NaN values e.g. for single floats.

`ext:float-infinity-p` *x* [Function]

`ext:float-nan-p` *x* [Function]

Predicates to test if *x* is infinite or NaN.

2.10.7 C Reference

2.10.7.1 Number C types

Numeric C types understood by ECL

Type names

cl_fixnum	fixnum		
cl_index	(integer 0 most-positive-fixnum)		
float	short-float, single-float		
double	double-float		
long double (*)	long-float	ECL_LONG_FLOAT	:long-float
float _Complex (**)	(complex single-float)	ECL_COMPLEX_FLOAT	:complex-float
double _Complex (**)	(complex double-float)	ECL_COMPLEX_FLOAT	:complex-float
long-double _Complex (**)	(complex long-float)	ECL_COMPLEX_FLOAT	:complex-float
uint8_t	(unsigned-byte 8)	ecl_uint8_t	
int8_t	(signed-byte 8)	ecl_int8_t	
uint16_t	(unsigned-byte 16)	ecl_uint16_t	:uint16-t
int16_t	(signed-byte 16)	ecl_int16_t	:int16-t
uint32_t	(unsigned-byte 32)	ecl_uint32_t	:uint32-t
int32_t	(signed-byte 32)	ecl_int32_t	:int32-t
uint64_t	(unsigned-byte 64)	ecl_uint64_t	:uint64-t
int64_t	(signed-byte 64)	ecl_int64_t	:int64-t
short	(integer ffi:c-short-min ffi:c-short-max)		
unsigned short	(integer 0 ffi:c-ushort-max)		
int	(integer ffi:c-int-min ffi:c-int-max)		
unsigned int	(integer 0 ffi:c-uint-max)		
long	(integer ffi:c-long-min ffi:c-long-max)		
unsigned long	(integer 0 ffi:c-long-max)		

long long	(integer ffi:c-long-long-min ffi:c-long-long-max)	ecl_long_long_t	:long-long
unsigned long long	(integer 0 ffi:c-ulong-long-max)	ecl_ulong_long_t	:ulong-long

Description The table above shows the relation between C types and the equivalent Common Lisp types. All types are standard C99 types, except for two. First, `cl_fixnum` is the smallest signed integer that can fit a fixnum. Second, `cl_index` is the smallest unsigned integer that fits a fixnum and is typically the unsigned counterpart of `cl_fixnum`.

(*) **DEPRECATED** Previous versions of ECL supported compilers that did not define the long double type. The `ECL_LONG_DOUBLE` macro and `long-double` features indicating whether support for long double was available are removed now.

(**) The <float> _Complex types do not exist on all platforms. When they exist the macro `ECL_COMPLEX_FLOAT` will be defined.

Many other types might also not exist on all platforms. This includes not only `long long` and `unsigned long long`, but also some of the C99 integer types. There are two ways to detect which integer types are available in your system:

- Check for the definition of C macros with a similar name, shown in the fifth column above.
- In Lisp code, check for the presence of the associated features, shown in the fourth column above.

2.10.7.2 Number constructors

Creating Lisp types from C numbers

Functions

`cl_object ecl_make_fixnum` (*cl_fixnum n*) [Function]
`cl_object ecl_make_integer` (*cl_fixnum n*) [Function]
`cl_object ecl_make_unsigned_integer` (*cl_index n*) [Function]
`cl_object ecl_make_single_float` (*float n*) [Function]
`cl_object ecl_make_double_float` (*double n*) [Function]
`cl_object ecl_make_long_float` (*long double n*) [Function]
`cl_object ecl_make_csfloat` (*float _Complex n*) [Function]
`cl_object ecl_make_cdfloat` (*double _Complex n*) [Function]
`cl_object ecl_make_clfloat` (*long double _Complex n*) [Function]
`cl_object ecl_make_uint8_t` (*uint8_t n*) [Function]
`cl_object ecl_make_int8_t` (*int8_t n*) [Function]
`cl_object ecl_make_uint16_t` (*uint16_t n*) [Function]
`cl_object ecl_make_int16_t` (*int16_t n*) [Function]
`cl_object ecl_make_uint32_t` (*uint32_t n*) [Function]
`cl_object ecl_make_int32_t` (*int32_t n*) [Function]
`cl_object ecl_make_uint64_t` (*uint64_t n*) [Function]
`cl_object ecl_make_int64_t` (*int64_t n*) [Function]
`cl_object ecl_make_short_t` (*short n*) [Function]
`cl_object ecl_make_ushort_t` (*unsigned short n*) [Function]

`cl_object ecl_make_int` (*int n*) [Function]
`cl_object ecl_make_uint` (*unsigned int n*) [Function]
`cl_object ecl_make_long` (*long n*) [Function]
`cl_object ecl_make_ulong` (*unsigned long n*) [Function]
`cl_object ecl_make_long_long` (*long long n*) [Function]
`cl_object ecl_make_ulong_long` (*unsigned long long n*) [Function]
`cl_object ecl_make_ratio` (*cl_object numerator, cl_object denominator*) [Function]
`cl_object ecl_make_complex` (*cl_object real, cl_object imag*) [Function]

Description These functions create a Lisp object from the corresponding C number. If the number is an integer type, the result will always be an integer, which may be a bignum. If on the other hand the C number is a float, double or long double, the result will be a float.

There is some redundancy in the list of functions that convert from cl_fixnum and cl_index to lisp. On the one hand, `ecl_make_fixnum` always creates a fixnum, dropping bits if necessary. On the other hand, `ecl_make_integer` and `ecl_make_unsigned_integer` faithfully convert to a Lisp integer, which may be a bignum.

Note also that some of the constructors do not use C numbers. This is the case of `ecl_make_ratio` and `ecl_make_complex`, because they are composite Lisp types. When c99 complex float support is built in `ecl_make_complex` will use C number for float types.

These functions or macros signal no errors.

2.10.7.3 Number accessors

Unchecked conversion from Lisp types to C numbers

Functions

`cl_fixnum ecl_fixnum` (*cl_object n*) [Function]
`float ecl_single_float` (*cl_object n*) [Function]
`double ecl_double_float` (*cl_object n*) [Function]
`long double ecl_long_float` (*cl_object n*) [Function]
`float _Complex ecl_csfloat` (*cl_object n*) [Function]
`double _Complex ecl_cdfloat` (*cl_object n*) [Function]
`long double _Complex ecl_clfloat` (*cl_object n*) [Function]

Description These functions and macros extract a C number from a Lisp object. They do not check the type of the Lisp object as they typically just access directly the value from a C structure.

2.10.7.4 Number coercion

Checked conversion from Lisp types to C numbers

Functions

`cl_fixnum ecl_to_fixnum` (*cl_object n*); [Function]
`cl_index ecl_to_unsigned_integer` (*cl_object n*); [Function]

`float ecl_to_float` (*cl_object n*); [Function]
`double ecl_to_double` (*cl_object n*); [Function]
`long double ecl_to_long_double` (*cl_object n*); [Function]
`float _Complex ecl_to_csfloat` (*cl_object n*); [Function]
`double _Complex ecl_to_cdfloat` (*cl_object n*); [Function]
`long double _Complex ecl_to_clfloat` (*cl_object n*); [Function]
`uint8_t ecl_to_uint8_t` (*cl_object n*); [Function]
`int8_t ecl_to_int8_t` (*cl_object n*); [Function]
`uint16_t ecl_to_uint16_t` (*cl_object n*); [Function]
`int16_t ecl_to_int16_t` (*cl_object n*); [Function]
`uint32_t ecl_to_uint32_t` (*cl_object n*); [Function]
`int32_t ecl_to_int32_t` (*cl_object n*); [Function]
`uint64_t ecl_to_uint64_t` (*cl_object n*); [Function]
`int64_t ecl_to_int64_t` (*cl_object n*); [Function]
`short ecl_to_short` (*cl_object n*); [Function]
`unsigned short ecl_to_ushort` (*cl_object n*); [Function]
`int ecl_to_int` (*cl_object n*); [Function]
`unsigned int ecl_to_uint` (*cl_object n*); [Function]
`long ecl_to_long` (*cl_object n*); [Function]
`unsigned long ecl_to_ulong` (*cl_object n*); [Function]
`long long ecl_to_long_long` (*cl_object n*); [Function]
`unsigned long long ecl_to_ulong_long` (*cl_object n*); [Function]

Description These functions and macros convert a Lisp object to the corresponding C number type. The conversion is done through a coercion process which may signal an error if the argument does not fit the expected type.

2.10.7.5 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
=	cl_object cl_E(cl_narg narg, ...)
/=	cl_object cl_NE(cl_narg narg, ...)
<	cl_object cl_L(cl_narg narg, ...)
>	cl_object cl_G(cl_narg narg, ...)
<=	cl_object cl_LE(cl_narg narg, ...)
>=	cl_object cl_GE(cl_narg narg, ...)
max	cl_object cl_max(cl_narg narg, ...)
min	cl_object cl_min(cl_narg narg, ...)
minusp	cl_object cl_minusp(cl_object real)
plusp	cl_object cl_plusp(cl_object real)
zerop	cl_object cl_zerop(cl_object number)
floor	cl_object cl_floor(cl_narg narg, cl_object number, ...)
ffloor	cl_object cl_ffloor(cl_narg narg, cl_object number, ...)
ceiling	cl_object cl_ceiling(cl_narg narg, cl_object number, ...)
fceiling	cl_object cl_fceiling(cl_narg narg, cl_object number, ...)
truncate	cl_object cl_truncate(cl_narg narg, cl_object number, ...)

ftruncate	cl_object cl_ftruncate(cl_narg narg, cl_object number, ...)
round	cl_object cl_round(cl_narg narg, cl_object number, ...)
fround	cl_object cl_fround(cl_narg narg, cl_object number, ...)
sin	cl_object cl_sin(cl_object radians)
cos	cl_object cl_cos(cl_object radians)
tan	cl_object cl_tan(cl_object radians)
asin	cl_object cl_asin(cl_object number)
acos	cl_object cl_acos(cl_object number)
atan	cl_object cl_atan(cl_narg narg, cl_object number1, ...)
sinh	cl_object cl_sinh(cl_object number)
cosh	cl_object cl_cosh(cl_object number)
tanh	cl_object cl_tanh(cl_object number)
asinh	cl_object cl_asinh(cl_object number)
acosh	cl_object cl_acosh(cl_object number)
atanh	cl_object cl_atanh(cl_object number)
*	cl_object cl_X(cl_narg narg, ...)
+	cl_object cl_P(cl_narg narg, ...)
-	cl_object cl_M(cl_narg narg, cl_object number, ...)
/	cl_object cl_N(cl_narg narg, cl_object number, ...)
1+	cl_object cl_1P(cl_object number)
1-	cl_object cl_1M(cl_object number)
abs	cl_object cl_abs(cl_object number)
evenp	cl_object cl_evenp(cl_object integer)
oddp	cl_object cl_oddp(cl_object integer)
exp	cl_object cl_exp(cl_object number)
expt	cl_object cl_expt(cl_object base, cl_object power)
gcd	cl_object cl_gcd(cl_narg narg, ...)
lcm	cl_object cl_lcm(cl_narg narg, ...)
log	cl_object cl_log(cl_narg narg, cl_object number, ...)
mod	cl_object cl_mod(cl_object number, cl_object divisor)
rem	cl_object cl_rem(cl_object number, cl_object divisor)
signum	cl_object cl_signum(cl_object number)
sqrt	cl_object cl_sqrt(cl_object number)
isqrt	cl_object cl_isqrt(cl_object natural)
make-random-state	cl_object cl_make_random_state(cl_narg narg, ...)
random	cl_object cl_random(cl_narg narg, cl_object limit, ...)
random-state-p	cl_object cl_random_state_p(cl_object object)
numberp	cl_object cl_numberp(cl_object object)
cis	cl_object cl_cis(cl_object radians)
complex	cl_object cl_complex(cl_narg narg, cl_object realpart, ...)
complexp	cl_object cl_complexp(cl_object object)
conjugate	cl_object cl_conjugate(cl_object number)
phase	cl_object cl_phase(cl_object number)
realpart	cl_object cl_realpart(cl_object number)
imagpart	cl_object cl_imagpart(cl_object number)
upgraded-complex-part-type	cl_object cl_upgraded_complex_part_type(cl_narg narg, cl_object typespec, ...)

realp	cl_object cl_realp(cl_object object)
numerator	cl_object cl_numerator(cl_object rational)
denominator	cl_object cl_denominator(cl_object rational)
rational	cl_object cl_rational(cl_object number)
rationalize	cl_object cl_rationalize(cl_object number)
rationalp	cl_object cl_rationalp(cl_object object)
ash	cl_object cl_ash(cl_object integer, cl_object count)
integer-length	cl_object cl_integer_length(cl_object integer)
integerp	cl_object cl_integerp(cl_object object)
parse-integer	cl_object cl_parse_integer(cl_narg narg, cl_object string, ...)
boole	cl_object cl_boole(cl_object op, cl_object integer1, cl_object integer2)
logand	cl_object cl_logand(cl_narg narg, ...)
logandc1	cl_object cl_logandc1(cl_object integer1, cl_object integer2)
logandc2	cl_object cl_logandc2(cl_object integer1, cl_object integer2)
logeqv	cl_object cl_logeqv(cl_narg narg, ...)
logior	cl_object cl_logior(cl_narg narg, ...)
lognand	cl_object cl_lognand(cl_object integer1, cl_object integer2)
lognor	cl_object cl_lognor(cl_object integer1, cl_object integer2)
lognot	cl_object cl_lognot(cl_object integer)
logorc1	cl_object cl_logorc1(cl_object integer1, cl_object integer2)
logorc2	cl_object cl_logorc2(cl_object integer1, cl_object integer2)
logxor	cl_object cl_logxor(cl_narg narg, ...)
logbitp	cl_object cl_logbitp(cl_object index, cl_object integer)
logcount	cl_object cl_logcount(cl_object integer)
logtest	cl_object cl_logtest(cl_object integer1, cl_object integer2)
byte	cl_object cl_byte(cl_object size, cl_object position)
bytes-size	cl_object cl_byte_size(cl_object bytespec)
byte-position	cl_object cl_byte_position(cl_object bytespec)
deposit-field	cl_object cl_deposit_field(cl_object newbyte, cl_object bytespec, cl_object integer)
dpb	cl_object cl_dpb(cl_object newbyte, cl_object bytespec, cl_object integer)
ldb	cl_object cl_ldb(cl_object bytespec, cl_object integer)
ldb-test	cl_object cl_ldb_test(cl_object bytespec, cl_object integer)
mask-field	cl_object cl_mask_field(cl_object bytespec, cl_object integer)
decode-float	cl_object cl_decode_float(cl_object float)
scale-float	cl_object cl_scale_float(cl_object float, cl_object integer)
float-radix	cl_object cl_float_radix(cl_object float)
float-sign	cl_object cl_float_sign(cl_narg narg, cl_object float1, ...)
float-digits	cl_object cl_float_digits(cl_object float)
float-precision	cl_object cl_float_precision(cl_object float)
integer-decode-float	cl_object cl_integer_decode_float(cl_object float)
float	cl_object cl_float(cl_narg narg, cl_object number, ...)
floatp	cl_object cl_floatp(cl_object object)
arithmetic-error-operands	[Only in Common Lisp]

arithmetic-error-operation [Only in Common Lisp]

2.11 Characters

ECL is fully ANSI Common-Lisp compliant in all aspects of the character data type, with the following peculiarities.

2.11.1 Unicode vs. POSIX locale

There are two ways of building ECL: with C or with Unicode character codes. These build modes are accessed using the `--disable-unicode` and `--enable-unicode` configuration options, the last one being the default.

When using C characters we are actually relying on the char type of the C language, using the C library functions for tasks such as character conversions, comparison, etc. In this case characters are typically 8 bit wide and the character order and collation are determines by the current POSIX or C locale. This is not very accurate, leaves out many languages and character encodings but it is sufficient for small applications that do not need multilingual support.

When no option is specified ECL builds with support for a larger character set, the Unicode 6.0 standard. This uses 24 bit large character codes, also known as *codepoints*, with a large database of character properties which include their nature (alphanumeric, numeric, etc), their case, their collation properties, whether they are standalone or composing characters, etc.

2.11.1.1 Character types

If ECL is compiled without Unicode support, all characters are implemented using 8-bit codes and the type `extended-char` is empty. If compiled with Unicode support, characters are implemented using 24 bits and the `extended-char` type covers characters above code 255.

Type	**Without Unicode**	**With Unicode**
standard-char	#\Newline,32-126	#\Newline,32-126
base-char	0-255	0-255
extended-char	-	256-16777215

2.11.1.2 Character names

All characters have a name. For non-printing characters between 0 and 32, and for 127 we use the ordinary ASCII names. Characters above 127 are printed and read using hexadecimal Unicode notation, with a U followed by 24 bit hexadecimal number, as in `U0126`.

Character	Code
#\Null	0
#\Ack	1
#\Bell	7
#\Backspace	8
#\Tab	9
#\Newline	10
#\Linefeed	10
#\Page	12
#\Esc	27
#\Escape	27
#\Space	32
#\Rubout	127
#\U0080	128

Table 2.6

Note that `#\Linefeed` is synonymous with `#\Newline` and thus is a member of `standard-char`.

2.11.2 `#\Newline` characters

Internally, ECL represents the `#\Newline` character by a single code. However, when using external formats, ECL may parse character pairs as a single `#\Newline`, and vice versa, use multiple characters to represent a single `#\Newline`, see Section 2.19.1.3 [Streams - External formats], page 74.

2.11.3 C Reference

2.11.3.1 C types

C character types

Type names

ecl_character	character
ecl_base_char	base-char

Description ECL defines two C types to hold its characters: `ecl_base_char` and `ecl_character`.

- When ECL is built without Unicode, they both coincide and typically match `unsigned char`, to cover the 256 codes that are needed.
- When ECL is built with Unicode, the two types are no longer equivalent, with `ecl_character` being larger.

For your code to be portable and future proof, use both types to really express what you intend to do.

2.11.3.2 Constructors

Creating and extracting characters from Lisp objects

Functions

`cl_object ECL_CODE_CHAR` (*ecl_character code*); [Macro]
`ecl_character ECL_CHAR_CODE` (*cl_object o*); [Macro]

`ecl_character ecl_char_code` (*cl_object o*); [Function]
`ecl_base_char ecl_base_char_code` (*cl_object o*); [Function]

Description These functions and macros convert back and forth from C character types to Lisp. The macros `ECL_CHAR_CODE` and `ECL_CODE_CHAR` perform this coercion without checking the arguments. The functions `ecl_char_code` and `ecl_base_char_code`, on the other hand, verify that the argument has the right type and signal an error otherwise.

2.11.3.3 Predicates

C predicates for Lisp characters

Functions

`bool ecl_base_char_p` (*ecl_character c*); [Function]
`bool ecl_alpha_char_p` (*ecl_character c*); [Function]
`bool ecl_alphanumericp` (*ecl_character c*); [Function]
`bool ecl_graphic_char_p` (*ecl_character c*); [Function]
`bool ecl_digitp` (*ecl_character c*); [Function]
`bool ecl_standard_char_p` (*ecl_character c*); [Function]

Description These functions are equivalent to their Lisp equivalents but return C booleans.

2.11.3.4 Character case

C functions related to the character case

Functions

`bool ecl_upper_case_p` (*ecl_character c*); [Function]
`bool ecl_lower_case_p` (*ecl_character c*); [Function]
`bool ecl_both_case_p` (*ecl_character c*); [Function]
`ecl_character ecl_char_downcase` (*ecl_character c*); [Function]
`ecl_character ecl_char_upcase` (*ecl_character c*); [Function]

Description These functions check or change the case of a character. Note that in a Unicode context, the output of these functions might not be accurate (for instance when the uppercase character has two or more codepoints).

2.11.3.5 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
char=	cl_object cl_charE(cl_narg narg, ...)
char/=	cl_object cl_charNE(cl_narg narg, ...)
char<	cl_object cl_charL(cl_narg narg, ...)

char>	cl_object cl_charG(cl_narg narg, ...)
char<=	cl_object cl_charLE(cl_narg narg, ...)
char>=	cl_object cl_charGE(cl_narg narg, ...)
char-equal	cl_object cl_char_equal(cl_narg narg, ...)
char-not-equal	cl_object cl_char_not_equal(cl_narg narg, ...)
char-lessp	cl_object cl_char_lessp(cl_narg narg, ...)
char-greaterp	cl_object cl_char_greaterp(cl_narg narg, ...)
char-not-greaterp	cl_object cl_char_not_greaterp(cl_narg narg, ...)
char-not-lessp	cl_object cl_char_not_lessp(cl_narg narg, ...)
character	cl_object cl_character(cl_object char_designator)
characterp	cl_object cl_characterp(cl_object object)
alpha-char-p	cl_object cl_alpha_char_p(cl_object character)
alphanumericp	cl_object cl_alphanumericp(cl_object character)
digit-char	cl_object cl_digit_char(cl_narg narg, cl_object character, ...)
digit-char-p	cl_object cl_digit_char_p(cl_narg narg, cl_object character, ...)
graphic-char-p	cl_object cl_graphic_char_p(cl_object character)
standard-char-p	cl_object cl_standard_char_p(cl_object character)
char_upcase	cl_object cl_char_upcase(cl_object character)
char-downcase	cl_object cl_char_downcase(cl_object character)
upper-case-p	cl_object cl_upper_case_p(cl_object character)
lower-case-p	cl_object cl_lower_case_p(cl_object character)
both-case-p	cl_object cl_both_case_p(cl_object character)
char-code	cl_object cl_char_code(cl_object character)
char-int	cl_object cl_char_int(cl_object character)
code-char	cl_object cl_code_char(cl_object code)
char-name	cl_object cl_char_name(cl_object character)
name-char	cl_object cl_name_char(cl_object name)
char-code-limit	ECL_CHAR_CODE_LIMIT

2.12 Conses

2.12.1 C Reference

2.12.1.1 Accessors

Accessing the elements of conses

Functions

`cl_object ECL_CONS_CAR` (*cl_object o*) [Function]
`cl_object ECL_CONS_CDR` (*cl_object o*) [Function]
`cl_object ECL_RPLACA` (*cl_object o, cl_object v*) [Function]
`cl_object ECL_RPLACD` (*cl_object o, cl_object v*) [Function]
`cl_object _ecl_car` (*cl_object o*) [Function]
`cl_object _ecl_cdr` (*cl_object o*) [Function]
`cl_object _ecl_caar` (*cl_object o*) [Function]
`cl_object _ecl_cadr` (*cl_object o*) [Function]
...

Description These functions access the elements of objects of type cons (`ECL_CONS_CAR`, `ECL_CONS_CDR`, `ECL_RPLACA` and `ECL_RPLACD`) or type list (`_ecl_car`, `_ecl_cdr`, `_ecl_caar`, ...). They don't check the type of their arguments.

2.12.1.2 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
cons	cl_object cl_cons(cl_object car, cl_object cdr)
consp	cl_object cl_consp(cl_object object)
atom	cl_object cl_atom(cl_object object)
rplaca	cl_object cl_rplaca(cl_object cons, cl_object car)
rplacd	cl_object cl_rplacd(cl_object cons, cl_object cdr)
car	cl_object cl_car(cl_object cons)
cdr	cl_object cl_cdr(cl_object cons)
caar	cl_object cl_caar(cl_object cons)
cdar	cl_object cl_cdar(cl_object cons)
cadr	cl_object cl_cadr(cl_object cons)
cddr	cl_object cl_cddr(cl_object cons)
caaar	cl_object cl_caaar(cl_object cons)
cdaar	cl_object cl_cdaar(cl_object cons)
cadar	cl_object cl_cadar(cl_object cons)
cddar	cl_object cl_cddar(cl_object cons)
caadr	cl_object cl_caadr(cl_object cons)
cdadr	cl_object cl_cdadr(cl_object cons)
caddr	cl_object cl_caddr(cl_object cons)
cdddr	cl_object cl_cdddr(cl_object cons)
caaaar	cl_object cl_caaaar(cl_object cons)
cdaaar	cl_object cl_cdaaar(cl_object cons)
cadaar	cl_object cl_cadaar(cl_object cons)
cddaar	cl_object cl_cddaar(cl_object cons)
caadar	cl_object cl_caadar(cl_object cons)
cdadar	cl_object cl_cdadar(cl_object cons)
caddar	cl_object cl_caddar(cl_object cons)
cdddar	cl_object cl_cdddar(cl_object cons)
caaadr	cl_object cl_caaadr(cl_object cons)
cdaadr	cl_object cl_cdaadr(cl_object cons)
cadadr	cl_object cl_cadadr(cl_object cons)
cddadr	cl_object cl_cddadr(cl_object cons)
caaddr	cl_object cl_caaddr(cl_object cons)
cdaddr	cl_object cl_cdaddr(cl_object cons)
cadddr	cl_object cl_cadddr(cl_object cons)
cddddr	cl_object cl_cddddr(cl_object cons)
copy-tree	cl_object cl_copy_tree(cl_object tree)
sublis	cl_object cl_sublis(cl_narg narg, cl_object alist, cl_object tree, ...)

nsublis	cl_object cl_sublis(cl_narg narg, cl_object alist, cl_object tree, ...)
subst	cl_object cl_subst(cl_narg narg, cl_object new, cl_object old, cl_object tree, ...)
subst-if	cl_object cl_subst_if(cl_narg narg, cl_object new, cl_object predicate, cl_object tree, ...)
subst-if-not	cl_object cl_subst_if_not(cl_narg narg, cl_object new, cl_object predicate, cl_object tree, ...)
nsubst	cl_object cl_nsubst(cl_narg narg, cl_object new, cl_object old, cl_object tree, ...)
nsubst-if	cl_object cl_nsubst_if(cl_narg narg, cl_object new, cl_object predicate, cl_object tree, ...)
nsubst-if-not	cl_object cl_nsubst_if_not(cl_narg narg, cl_object new, cl_object predicate, cl_object tree, ...)
tree-equal	cl_object cl_tree_equal(cl_narg narg, cl_object tree1, cl_object tree2, ...)
copy-list	cl_object cl_copy_list(cl_object list)
list	cl_object cl_list(cl_narg narg, ...)
list*	cl_object cl_listX(cl_narg narg, ...)
list-length	cl_object cl_list_length(cl_object list)
listp	cl_object cl_listp(cl_object object)
make-list	cl_object cl_make_list(cl_narg narg, cl_object size, ...)
first	cl_object cl_first(cl_object list)
second	cl_object cl_second(cl_object list)
third	cl_object cl_third(cl_object list)
fourth	cl_object cl_fourth(cl_object list)
fifth	cl_object cl_fifth(cl_object list)
sixth	cl_object cl_sixth(cl_object list)
seventh	cl_object cl_seventh(cl_object list)
eighth	cl_object cl_eighth(cl_object list)
ninth	cl_object cl_ninth(cl_object list)
tenth	cl_object cl_tenth(cl_object list)
nth	cl_object cl_nth(cl_object n, cl_object list)
endp	cl_object cl_endp(cl_object list)
null	cl_object cl_null(cl_object object)
nconc	cl_object cl_nconc(cl_narg narg, ...)
append	cl_object cl_append(cl_narg narg, ...)
revappend	cl_object cl_revappend(cl_object list, cl_object tail)
nreconc	cl_object cl_nreconc(cl_object list, cl_object tail)
butlast	cl_object cl_butlast(cl_narg narg, cl_object list, ...)
nbutlast	cl_object cl_nbutlast(cl_narg narg, cl_object list, ...)
last	cl_object cl_last(cl_narg narg, cl_object list, ...)
ldiff	cl_object cl_ldiff(cl_object list, cl_object object)
tailp	cl_object cl_tailp(cl_object object, cl_object list)
nthcdr	cl_object cl_nthcdr(cl_object n, cl_object list)
rest	cl_object cl_rest(cl_object list)

member	cl_object cl_member(cl_narg narg, cl_object member, cl_object list, ...)
member-if	cl_object cl_member_if(cl_narg narg, cl_object predicate, cl_object list, ...)
member-if-not	cl_object cl_member_if_not(cl_narg narg, cl_object predicate, cl_object list, ...)
mapc	cl_object cl_mapc(cl_narg narg, cl_object function, ...)
mapcar	cl_object cl_mapcar(cl_narg narg, cl_object function, ...)
mapcan	cl_object cl_mapcan(cl_narg narg, cl_object function, ...)
mapl	cl_object cl_mapl(cl_narg narg, cl_object function, ...)
maplist	cl_object cl_maplist(cl_narg narg, cl_object function, ...)
mapcon	cl_object cl_mapcon(cl_narg narg, cl_object function, ...)
acons	cl_object cl_acons(cl_object key, cl_object datum, cl_object alist)
assoc	cl_object cl_assoc(cl_narg narg, cl_object item, cl_object alist, ...)
assoc-if	cl_object cl_assoc_if(cl_narg narg, cl_object predicate, cl_object alist, ...)
assoc-if-not	cl_object cl_assoc_if_not(cl_narg narg, cl_object predicate, cl_object alist, ...)
copy-alist	cl_object cl_copy_alist(cl_object alist)
pairlis	cl_object cl_pairlis(cl_narg narg, cl_object keys, cl_object data, ...)
rassoc	cl_object cl_rassoc(cl_narg narg, cl_object item, cl_object alist, ...)
rassoc-if	cl_object cl_rassoc_if(cl_narg narg, cl_object predicate, cl_object alist, ...)
rassoc-if-not	cl_object cl_rassoc_if_not(cl_narg narg, cl_object predicate, cl_object alist, ...)
get-properties	cl_object cl_get_properties(cl_object plist, cl_object indicator_list)
getf	cl_object cl_getf(cl_narg narg, cl_object plist, cl_object indicator, ...)
intersection	cl_object cl_intersection(cl_narg narg, cl_object list1, cl_object list2, ...)
nintersection	cl_object cl_nintersection(cl_narg narg, cl_object list1, cl_object list2, ...)
adjoin	cl_object cl_adjoin(cl_narg narg, cl_object item, cl_object list, ...)
set-difference	cl_object cl_set_difference(cl_narg narg, cl_object list1, cl_object list2, ...)
nset-difference	cl_object cl_nset_difference(cl_narg narg, cl_object list1, cl_object list2, ...)
set-exclusive-or	cl_object cl_set_exclusive_or(cl_narg narg, cl_object list1, cl_object list2, ...)

nset-exclusive-or	cl_object cl_nset_exclusive_or(cl_narg narg, cl_object list1, cl_object list2, ...)
subsetp	cl_object cl_subsetp(cl_narg narg, cl_object list1, cl_object list2, ...)
union	cl_object cl_union(cl_narg narg, cl_object list1, cl_object list2, ...)
nunion	cl_object cl_nunion(cl_narg narg, cl_object list1, cl_object list2, ...)

2.13 Arrays

2.13.1 Array limits

ECL arrays can have up to 64 dimensions. Common-Lisp constants related to arrays have the following values in ECL.

Constant	**Value**
array-rank-limit	64
array-dimension-limit	most-positive-fixnum
array-total-size-limit	array-dimension-limit

2.13.2 Specializations

When the elements of an array are declared to have some precise type, such as a small or large integer, a character or a floating point number, ECL has means to store those elements in a more compact form, known as a *specialized array*. The list of types for which ECL specializes arrays is platform dependent, but is summarized in the following table, together with the C type which is used internally and the expected size.

Specialized type	**Element C type**	**Size**
bit	-	1 bit
character	unsigned char or uint32_t	Depends on character range
base-char	unsigned char	
fixnum	cl_fixnum	Machine word (32 or 64 bits)
ext:cl-index	cl_index	Machine word (32 or 64 bits)
(signed-byte 8)	int8_t	8 bits
(unsigned-byte 8)	uint8_t	8 bits
(signed-byte 16)	int16_t	16 bits
(unsigned-byte 16)	uint16_t	16 bits
(signed-byte 32)	int32_t	32 bits
(unsigned-byte 32)	uint32_t	32 bits
(signed-byte 64)	int64_t	64 bits
(unsigned-byte 64)	uint64_t	64 bits
single-float or short-float	float	32-bits IEEE float
double-float	double	64-bits IEEE float

long-float	long double	Between 96 and 128 bits
(complex single-float)	float _Complex	64 bits
(complex double-float)	double _Complex	128 bits
(complex long-float)	long double _Complex	Between 192 and 256 bits
t	cl_object	Size of a pointer.

Let us remark that some of these specialized types might not exist in your platform. This is detected using conditional reading and features (See Section 2.10 [Numbers], page 41).

2.13.3 C Reference

2.13.3.1 Types and constants

C types, limits and enumerations

Constants and types

`ECL_ARRAY_RANK_LIMIT` [Constant]
`ECL_ARRAY_DIMENSION_LIMIT` [Constant]
`ECL_ARRAY_TOTAL_LIMIT` [Constant]

`cl_elttype` {*ecl_aet_object, ...*} [enum]

Lisp or C type	Enumeration value	Lisp or C type	Enumeration value
t	ecl_aet_object	(unsigned-byte 1)	ecl_aet_bit
cl_fixnum	ecl_aet_fix	cl_index	ecl_aet_index
(unsigned-byte 8)	ecl_aet_b8	(signed-byte 8)	ecl_aet_i8
(unsigned-byte 16)	ecl_aet_b16	(signed-byte 16)	ecl_aet_i16
(unsigned-byte 32)	ecl_aet_b32	(signed-byte 32)	ecl_aet_i32
(unsigned-byte 64)	ecl_aet_b64	(signed-byte 64)	ecl_aet_i64
ecl_character	ecl_aet_ch	ecl_base_char	ecl_aet_bc
single-float	ecl_aet_sf	double-float	ecl_aet_df
long-float	ecl_aet_lf	(complex long-float)	ecl_aet_clf
(complex single-float)	ecl_aet_csf	(complex double-float)	ecl_aet_cdf

Description This list contains the constants that limit the rank of an array (`ECL_ARRAY_RANK_LIMIT`), the maximum size of each dimension (`ECL_ARRAY_DIMENSION_LIMIT`) and the maximum number of elements in an array (`ECL_ARRAY_TOTAL_LIMIT`).

ECL uses also internally a set of constants to describe the different specialized arrays. The constants form up the enumeration type `cl_elttype`. They are listed in the table above, which associates enumeration values with the corresponding Common Lisp element type.

2.13.3.2 ecl_aet_to_symbol, ecl_symbol_to_aet

To and from element types

Functions

`cl_object ecl_aet_to_symbol` (*cl_elttype param*) [Function]

`cl_elttype ecl_symbol_to_aet` (*cl_object type*) [Function]

Description `ecl_aet_to_symbol` returns the Lisp type associated to the elements of that specialized array class. `ecl_symbol_to_aet` does the converse, computing the C constant that is associated to a Lisp element type.

The functions may signal an error if any of the arguments is an invalid C or Lisp type.

2.13.3.3 Constructors

Creating array and vectors

Functions

`cl_object ecl_alloc_simple_vector` (*cl_index length, cl_elttype element_type*); [Function]

`cl_object si_make_vector` (*cl_object element_type, cl_object length, cl_object adjustablep, cl_object fill_pointerp, cl_object displaced_to, cl_object displacement*); [Function]

`cl_object si_make_array` (*cl_object element_type, cl_object dimensions, cl_object adjustablep, cl_object fill_pointerp, cl_object displaced_to, cl_object displacement*); [Function]

`cl_object si_adjust_vector` (*cl_object vector, cl_object length*); [Function]

Description The function `ecl_alloc_simple_vector` is the simplest constructor, creating a simple vector (i.e. non-adjustable and without a fill pointer), of the given size, preallocating the memory for the array data. The first argument, *element_type*, is a C constant that represents a valid array element type (See `cl_elttype`).

The function `si_make_vector` does the same job but allows creating an array with fill pointer, which is adjustable or displaced to another array.

- element_type is now a Common Lisp type descriptor, which is a symbol or list denoting a valid element type
- dimension is a non-negative fixnum with the vector size.
- fill_pointerp is either ECL_NIL or a non-negative fixnum denoting the fill pointer value.
- displaced_to is either ECL_NIL or a valid array to which the new array is displaced.
- displacement is either ECL_NIL or a non-negative value with the array displacement.

Adjustable vector may be adjusted with the function `si_adjust_vector`.

Finally, the function `si_make_array` does a similar job to `si_make_vector` but its second argument, *dimension*, can be a list of dimensions, to create a multidimensional array.

Examples Create one-dimensional `base-string` with room for 11 characters:

```
cl_object s = ecl_alloc_simple_vector(11, ecl_aet_bc);
```

Create a one-dimensional `array` with a fill pointer

```
cl_object type = ecl_make_symbol("BYTE8","EXT");
```

```
cl_object a = si_make_vector(type, ecl_make_fixnum(16), ECL_NIL, /* adjustable */
                             ecl_make_fixnum(0) /* fill-pointer */,
                             ECL_NIL /* displaced_to */,
                             ECL_NIL /* displacement */);
```

An alternative formulation

```
cl_object type = ecl_make_symbol("BYTE8","EXT");
cl_object a = si_make_array(type, ecl_make_fixnum(16), ECL_NIL, /* adjustable */
                            ecl_make_fixnum(0) /* fill-pointer */,
                            ECL_NIL /* displaced_to */,
                            ECL_NIL /* displacement */);
```

Create a 2-by-3 two-dimensional `array`, specialized for an integer type:

```
cl_object dims = cl_list(2, ecl_make_fixnum(2), ecl_make_fixnum(3));
cl_object type = ecl_make_symbol("BYTE8","EXT");
cl_object a = si_make_array(dims, type, ECL_NIL, /* adjustable */
                            ECL_NIL /* fill-pointer */,
                            ECL_NIL /* displaced_to */,
                            ECL_NIL /* displacement */);
```

2.13.3.4 Accessors

Reading and writing array elements

Functions

`cl_object ecl_aref` (*cl_object array, cl_index row_major_index*); [Function]
`cl_object ecl_aset` (*cl_object array, cl_index row_major_index, cl_object new_value*); [Function]
`cl_object ecl_aref1` (*cl_object vector, cl_index row_major_index*); [Function]
`cl_object ecl_aset1` (*cl_object vector, cl_index row_major_index, cl_object new_value*); [Function]

Description `ecl_aref` accesses an array using the supplied *row_major_index*, checking the array bounds and returning a Lisp object for the value at that position. `ecl_aset` does the converse, storing a Lisp value at the given *row_major_index*.

The first argument to `ecl_aref` or `ecl_aset` is an array of any number of dimensions. For an array of rank `N` and dimensions `d1, d2 ...` up to `dN`, the row major index associated to the indices (`i1,i2,...iN`) is computed using the formula `i1+d1*(i2+d3*(i3+...))`.

`ecl_aref1` and `ecl_aset1` are specialized versions that only work with one-dimensional arrays or vectors. They verify that the first argument is indeed a vector.

All functions above check that the index does not exceed the array bounds, that the values match the array element type and that the argument is an array (or a vector). If these conditions are not met, a `type-error` is signaled.

2.13.3.5 Array properties

Array size, fill pointer, etc.

Functions

`cl_elttype ecl_array_elttype` (*cl_object array*); [Function]
`cl_index ecl_array_rank` (*cl_object array*); [Function]
`cl_index ecl_array_dimension` (*cl_object array, cl_index index*); [Function]

Description These functions query various properties of the arrays. Some of them belong to the list of functions in the Common Lisp package, without any need for specialized versions. More precisely

- `ecl_array_elttype` returns the array element type, with the encoding found in the enumeration cl_elttype.
- `ecl_array_rank` returns the number of dimensions of the vector or array.
- `ecl_array_dimension` queries the dimension of an array, where index is a non-negative integer between 0 and `ecl_array_dimension(array)-1`.

2.13.3.6 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
make-array	cl_object cl_make_array(cl_narg narg, cl_object dimension...)
adjust-array	cl_object cl_adjust_array(cl_narg narg, cl_object array, cl_object dimensions, ...)
adjustable-array-p	cl_object cl_adjustable_array_p(cl_object array)
aref	cl_object cl_aref(cl_narg narg, cl_object array, ...)
(setf aref)	cl_object si_aset(cl_narg narg, cl_object array, ...)
array-dimension	cl_object cl_array_dimension(cl_object array, cl_object index)
array-dimensions	cl_object cl_array_dimensions(cl_object array)
array-element-type	cl_object cl_array_element_type(cl_object array)
array-has-fill-pointer-p	cl_object cl_array_has_fill_pointer_p(cl_object array)
array-displacement	cl_object cl_array_displacement(cl_object array)
array-in-bounds-p	cl_object cl_array_in_bounds_p(cl_narg narg, cl_object array, ...)
array-rank	cl_object cl_array_rank(cl_object array)
array-row-major-index	cl_object cl_array_row_major_index(cl_narg narg, cl_object array, ...)
array-total-size	cl_object cl_array_total_size(cl_object array)
arrayp	cl_object cl_arrayp(cl_object array)
fill-pointer	cl_object cl_fill_pointer(cl_object array)
(setf fill-pointer)	cl_object si_fill_pointer_set(cl_object array, cl_object fill_pointer)
row-major-aref	cl_object cl_row_major_aref(cl_object array, cl_object index)
(setf row-major-aref)	cl_object si_row_major_aset(cl_object array, cl_object index, cl_object value)

upgraded-array-element-type	cl_object cl_upgraded_array_element_type(cl_narg narg, cl_object typespec, ...)
simple-vector-p	cl_object cl_simple_vector_p(cl_object object)
svref	cl_object cl_svref(cl_object simple_vector, cl_object index)
(setf svref)	cl_object si_svset(cl_object simple_vector, cl_object index, cl_object value)
vector	cl_object cl_vector(cl_narg narg, ...)
vector-pop	cl_object cl_vector_pop(cl_object vector)
vector-push	cl_object cl_vector_push(cl_object new_element, cl_object vector)
vector-push-extend	cl_object cl_vector_push_extend(cl_narg narg, cl_object new_element, cl_object vector, ...)
vectorp	cl_object cl_vectorp(cl_object object)
bit	cl_object cl_bit(cl_narg narg, cl_object bit_array, ...)
(setf bit)	cl_object si_aset(cl_narg narg, cl_object array, ...)
sbit	cl_object cl_sbit(cl_narg narg, cl_object bit_array, ...)
(setf sbit)	cl_object si_aset(cl_narg narg, cl_object array, ...)
bit-and	cl_object cl_bit_and(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-andc1	cl_object cl_bit_andc1(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-andc2	cl_object cl_bit_andc2(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-eqv	cl_object cl_bit_eqv(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-ior	cl_object cl_bit_ior(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-nand	cl_object cl_bit_nand(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-nor	cl_object cl_bit_nor(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-orc1	cl_object cl_bit_orc1(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-orc2	cl_object cl_bit_orc1(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-xor	cl_object cl_bit_xor(cl_narg narg, cl_object array1, cl_object array2, ...)
bit-not	cl_object cl_bit_not(cl_narg narg, cl_object array, ...)
bit-vector-p	cl_object cl_bit_vector_p(cl_object object)
simple-bit-vector-p	cl_object cl_simple_bit_vector_p(cl_object object)

2.14 Strings

2.14.1 String types & Unicode

The ECL implementation of strings is ANSI Common-Lisp compliant. There are basically four string types as shown in Table 2.7. As explained in Section 2.11 [Characters], page 50, when Unicode support is disabled, `character` and `base-character` are the same type and the last two string types are equivalent to the first two.

Abbreviation	Expanded type	Remarks
string	(array character (*))	8 or 32 bits per character, adjustable.
simple-string	(simple-array character (*))	8 or 32 bits per character, not adjustable nor displaced.
base-string	(array base-char (*))	8 bits per character, adjustable.
simple-base-string	(simple-array base-char (*))	8 bits per character, not adjustable nor displaced.

Table 2.7: Common Lisp string types

It is important to remember that strings with unicode characters can only be printed readably when the external format supports those characters. If this is not the case, ECL will signal a `serious-condition`. This condition will abort your program if not properly handled.

2.14.2 C reference

2.14.2.1 Base string constructors

Building strings of C data

Functions

`cl_object ecl_alloc_adjustable_base_string` (*cl_index length*); [Function]

`cl_object ecl_alloc_simple_base_string` (*cl_index length*); [Function]

`cl_object ecl_make_simple_base_string` (*const char* data, cl_fixnum length*); [Function]

`cl_object ecl_make_constant_base_string` (*const char* data, cl_fixnum length*); [Function]

Description These are different ways to create a base string, which is a string that holds a small subset of characters, the `base-char`, with codes ranging from 0 to 255.

`ecl_alloc_simple_base_string` creates an empty string with that much space for characters and a fixed length. The string does not have a fill pointer and cannot be resized, and the initial data is unspecified

`ecl_alloc_adjustable_base_string` is similar to the previous function, but creates an adjustable string with a fill pointer. This means that the length of the string can be changed and the string itself can be resized to accommodate more data.

The other constructors create strings but use some preexisting data. `ecl_make_simple_base_string` creates a string copying the data that the user supplies, and using freshly allocated memory. `ecl_make_constant_base_string` on the other hand,

does not allocate memory, but simply uses the supplied pointer as buffer for the string. This last function should be used with care, ensuring that the supplied buffer is not deallocated. If the *length* argument of these functions is -1, the length is determined by `strlen`.

2.14.2.2 String accessors

Reading and writing characters into a string

Functions

`ecl_character ecl_char` (*cl_object string, cl_index index*); [Function]

`ecl_character ecl_char_set` (*cl_object string, cl_index index, ecl_character c*); [Function]

Description Access to string information should be done using these two functions. The first one implements the equivalent of the `char` function from Common Lisp, returning the character that is at position *index* in the string *string*.

The counterpart of the previous function is `ecl_char_set`, which implements `(setf char)` and stores character *c* at the position *index* in the given string.

Both functions check the type of their arguments and verify that the indices do not exceed the string boundaries. Otherwise they signal a `serious-condition`.

2.14.2.3 Converting Unicode strings

Converting between different encodings. See Section 2.19.1.3 [Streams - External formats], page 74, for a list of supported encodings (external formats).

Functions

`ext:octets-to-string` *octets* **&key** (*external-format* **:default**) (*start* **0**) (*end* **nil**) [Function]

Decode a sequence of octets (i.e. 8-bit bytes) into a string according to the given external format. *octets* must be a vector whose elements have a size of 8-bit. The bounding index designators *start* and *end* optionally denote a subsequence to be decoded. Signals an `ext:character-decoding-error` if the decoding fails.

`ext:string-to-octets` *string* **&key** (*external-format* **:default**) (*start* **0**) (*end* **nil**) (*null-terminate* **nil**) [Function]

Encode a string into a sequence of octets according to the given external format. The bounding index designators *start* and *end* optionally denote a subsequence to be encoded. If *null-terminate* is true, add a terminating null byte. Signals an `ext:character-encoding-error` if the encoding fails.

`cl_object ecl_decode_from_cstring` (*const char *string, cl_fixnum length, cl_object external_format*) [Function]

Decode a C string of the given length into a Lisp string using the specified external format. If *length* is -1, the length is determined by `strlen`. Returns `NULL` if the decoding fails.

`cl_fixnum ecl_encode_to_cstring` (*char *output, cl_fixnum output_length, cl_object input, cl_object external_format*) [Function]

Encode the Lisp string *input* into a C string of the given length using the specified external format. Returns the number of characters necessary to encode the Lisp string (including the null terminator). If this is larger than *output_length*, *output* is unchanged. Returns -1 if the encoding fails.

`cl_object ecl_decode_from_unicode_wstring` (*const wchar_t *string, cl_fixnum length*) [Function]

`cl_fixnum ecl_encode_to_unicode_wstring` (*wchar_t *output, cl_fixnum output_length, cl_object input*) [Function]

These functions work the same as `ecl_decode_from_cstring`, `ecl_encode_to_cstring`, except that the external format used is either utf-8, utf-16 or utf-32 depending on whether `sizeof(wchar_t)` is 1, 2, or 4 respectively.

2.14.2.4 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
simple-string-p	cl_object cl_simple_string_p(cl_object string)
char	cl_object cl_char(cl_object string, cl_object index)
(setf char)	cl_object si_char_set(cl_object string, cl_object index, cl_object char)
schar	cl_object cl_schar(cl_object string, cl_object index)
(setf schar)	cl_object si_char_set(cl_object string, cl_object index, cl_object char)
string	cl_object cl_string(cl_object x)
string-upcase	cl_object cl_string_upcase(cl_narg narg, cl_obejct string, ...)
string-downcase	cl_object cl_string_downcase(cl_narg narg, cl_obejct string, ...)
string-capitalize	cl_object cl_string_capitalize(cl_narg narg, cl_obejct string, ...)
nstring-upcase	cl_object cl_nstring_upcase(cl_narg narg, cl_obejct string, ...)
nstring-downcase	cl_object cl_nstring_downcase(cl_narg narg, cl_obejct string, ...)
nstring-capitalize	cl_object cl_nstring_capitalize(cl_narg narg, cl_obejct string, ...)
string-trim	cl_object cl_string_trim(cl_object character_bag, cl_object string)
string-left-trim	cl_object cl_string_left_trim(cl_object character_bag, cl_object string)
string-right-trim	cl_object cl_string_right_trim(cl_object character_bag, cl_object string)
string	cl_object cl_string(cl_object x)
string=	cl_object cl_stringE(cl_narg narg, cl_object string1, cl_object string2, ...)
string/=	cl_object cl_stringNE(cl_narg narg, cl_object string1, cl_object string2, ...)
string<	cl_object cl_stringL(cl_narg narg, cl_object string1, cl_object string2, ...)
string>	cl_object cl_stringG(cl_narg narg, cl_object string1, cl_object string2, ...)

string<=	cl_object cl_stringLE(cl_narg narg, cl_object string1, cl_object string2, ...)
string>=	cl_object cl_stringGE(cl_narg narg, cl_object string1, cl_object string2, ...)
string-equal	cl_object cl_string_equal(cl_narg narg, cl_object string1, cl_object string2, ...)
string-not-equal	cl_object cl_string_not_equal(cl_narg narg, cl_object string1, cl_object string2, ...)
string-lessp	cl_object cl_string_lessp(cl_narg narg, cl_object string1, cl_object string2, ...)
string-greaterp	cl_object cl_string_greaterp(cl_narg narg, cl_object string1, cl_object string2, ...)
string-not-greaterp	cl_object cl_string_not_greaterp(cl_narg narg, cl_object string1, cl_object string2, ...)
string-not-lessp	cl_object cl_string_not_lessp(cl_narg narg, cl_object string1, cl_object string2, ...)
stringp	cl_object cl_stringp(cl_object x)
make-string	cl_object cl_make_string(cl_narg narg, cl_object size, ...)

2.15 Sequences

2.15.1 C Reference

2.15.1.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
concatenate	cl_object cl_concatenate(cl_narg narg, cl_object result_type, ...)
copy-seq	cl_object cl_copy_seq(cl_object sequence)
count	cl_object cl_count(cl_narg narg, cl_object item, cl_object sequence, ...)
count-if	cl_object cl_count_if(cl_narg narg, cl_object predicate, cl_object sequence, ...)
count-if-not	cl_object cl_count_if_not(cl_narg narg, cl_object predicate, cl_object sequence, ...)
delete	cl_object cl_delete(cl_narg narg, cl_object item, cl_object sequence, ...)
delete-if	cl_object cl_delete_if(cl_narg narg, cl_object test, cl_object sequence, ...)
delete-if-not	cl_object cl_delete_if_not(cl_narg narg, cl_object test, cl_object sequence, ...)
delete-duplicates	cl_object cl_delete_duplicates(cl_narg narg, cl_object sequence, ...)
elt	cl_object cl_elt(cl_object sequence, cl_object index)
(setf elt)	cl_object si_elt_set(cl_object sequence, cl_object index, cl_object value)

fill	cl_object cl_fill(cl_narg narg, cl_object sequence, cl_object item, ...)
find	cl_object cl_find(cl_narg narg, cl_object item, cl_object sequence, ...)
find-if	cl_object cl_find_if(cl_narg narg, cl_object predicate, cl_object sequence, ...)
find-if-not	cl_object cl_find_if_not(cl_narg narg, cl_object predicate, cl_object sequence, ...)
length	cl_object cl_length(cl_object x)
make-sequence	cl_object cl_make_sequence(cl_narg narg, cl_object result_type, cl_object size, ...)
map	cl_object cl_map(cl_narg narg, cl_object result_type, cl_object function, , ...)
map-into	cl_object cl_map_into(cl_narg narg, cl_object result_sequence, cl_object function, ...)
merge	cl_object cl_merge(cl_narg narg, cl_object result_type, cl_object sequence1, cl_object sequence2, cl_object predicate, ...)
mismatch	cl_object cl_mismatch(cl_narg narg, cl_object sequence1, cl_object sequence2, ...)
nreverse	cl_object cl_nreverse(cl_object sequence)
nsubstitute	cl_object cl_nsubstitute(cl_narg narg, cl_object newitem, cl_object olditem, cl_object sequence, ...)
nsubstitute-if	cl_object cl_nsubstitute_if(cl_narg narg, cl_object newitem, cl_object predicate, cl_object sequence, ...)
nsubstitute-if-not	cl_object cl_nsubstitute_if_not(cl_narg narg, cl_object newitem, cl_object predicate, cl_object sequence, ...)
position	cl_object cl_position(cl_narg narg, cl_object item, cl_object sequence, ...)
position-if	cl_object cl_position_if(cl_narg narg, cl_object predicate, cl_object sequence, ...)
position-if-not	cl_object cl_position_if_not(cl_narg narg, cl_object predicate, cl_object sequence, ...)
reduce	cl_object cl_reduce(cl_narg narg, cl_object function, cl_object sequence, ...)
remove	cl_object cl_remove(cl_narg narg, cl_object item, cl_object sequence, ...)
remove-if	cl_object cl_remove_if(cl_narg narg, cl_object test, cl_object sequence, ...)
remove-if-not	cl_object cl_remove_if_not(cl_narg narg, cl_object test, cl_object sequence, ...)
remove-duplicates	cl_object cl_remove_duplicates(cl_narg narg, cl_object sequence, ...)
replace	cl_object cl_replace(cl_narg narg, cl_object sequence1, cl_object sequence2, ...)
reverse	cl_object cl_reverse(cl_object sequence)
search	cl_object cl_search(cl_narg narg, cl_object sequence1, cl_object sequence2, ...)

sort	cl_object cl_sort(cl_narg narg, cl_object sequence, cl_object predicate, ...)
stable-sort	cl_object cl_stable_sort(cl_narg narg, cl_object sequence, cl_object predicate, ...)
subseq	cl_object cl_subseq(cl_narg narg, cl_object sequence, cl_object start, ...)
substitute	cl_object cl_substitute(cl_narg narg, cl_object newitem, cl_object olditem, cl_object sequence, ...)
substitute-if	cl_object cl_substitute_if(cl_narg narg, cl_object newitem, cl_object predicate, cl_object sequence, ...)
substitute-if-not	cl_object cl_substitute_if_not(cl_narg narg, cl_object newitem, cl_object predicate, cl_object sequence, ...)

2.16 Hash tables

2.16.1 Extensions

2.16.1.1 Weakness in hash tables

Weak hash tables allow the garbage collector to reclaim some of the entries if they are not strongly referenced elsewhere. ECL supports four kinds of weakness in hash tables: `:key`, `:value`, `:key-and-value` and `:key-or-value`.

To make hash table weak, programmer has to provide `:weakness` keyword argument to `cl:make-hash-table` with the desired kind of weakness value (`nil` means that the hash table has only strong references).

For more information see Weak References - Data Types and Implementation (`https://www.haible.de/bruno/papers/cs/weak/WeakDatastructures-writeup.html`) by Bruno Haible.

`ext:hash-table-weakness` *ht* [Function]

Returns type of the hash table weakness. Possible return values are: `:key`, `:value`, `:key-and-value`, `:key-or-value` or `nil`.

2.16.1.2 Thread-safe hash tables

By default ECL doesn't protect hash tables from simultaneous access for performance reasons. Read and write access may is synchronized when `:synchronized` keyword argument to `make-hash-table` is `t` - `(make-hash-table :synchronized t)`.

`ext:hash-table-synchronized-p` *ht* [Function]

Predicate answering whether hash table is synchronized or not.

2.16.1.3 Hash tables serialization

`ext:hash-table-content` *ht* [Function]

Returns freshly consed list of pairs `(key . val)` being contents of the hash table.

`ext:hash-table-fill` *ht values* [Function]

Fills *ht* with *values* being list of `(key . val)`. Hash table may have some content already, but conflicting keys will be overwritten.

2.16.1.4 Custom equivalence predicate

`make-hash-table` may accept arbitrary `:test` keyword for the equivalence predicate. If it is not one of the standard predicates (`:eq`, `:eql`, `:equal`, `:equalp`) a keyword argument `:hashing-function` must be a function accepting one argument and returning a positive fixnum. Otherwise the argument is ignored.

2.16.1.5 Example

```
CL-USER> (defparameter *ht*
           (make-hash-table :synchronized t
                            :weakness :key-or-value))
*HT*

CL-USER> (ext:hash-table-weakness *ht*)
:KEY-OR-VALUE

CL-USER> (ext:hash-table-synchronized-p *ht*)
T

CL-USER> (ext:hash-table-fill *ht* '((:foo 3) (:bar 4) (:quux 5)))
#<hash-table 000055b1229e0b40>

CL-USER> (ext:hash-table-content *ht*)
((#<weak-pointer 000055b121866350> . #<weak-pointer 000055b121866320>)
 (#<weak-pointer 000055b121866370> . #<weak-pointer 000055b121866360>)
 (#<weak-pointer 000055b121866390> . #<weak-pointer 000055b121866380>))
```

2.16.2 C Reference

2.16.2.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
clrhash	cl_object cl_clrhash(cl_object hash_table)
gethash	cl_object cl_gethash(cl_narg narg, cl_object key, cl_object hash_table, ...)
(setf gethash)	cl_object si_hash_set(cl_object key, cl_object hash_table, cl_object value)
hash-table-count	cl_object cl_hash_table_count(cl_object hash_table)
hash-table-p	cl_object cl_hash_table_p(cl_object hash_table)
hash-table-rehash-size	cl_object cl_hash_table_rehash_size(cl_object hash_table)
hash-table-rehash-threshold	cl_object cl_hash_table_rehash_threshold(cl_object hash_table)
hash-table-size	cl_object cl_hash_table_size(cl_object hash_table)
hash-table-test	cl_object cl_hash_table_test(cl_object hash_table)
make-hash-table	cl_object cl_make_hash_table(cl_narg narg, ...)

maphash	cl_object cl_maphash(cl_object function, cl_object hash_table)
remhash	cl_object cl_remhash(cl_object key, cl_object hash_table)
sxhash	cl_object cl_sxhash(cl_object object)

2.17 Filenames

2.17.1 Syntax

A pathname in the file system of Common-Lisp consists of six elements: host, device, directory, name, type and version. Pathnames are read and printed using the `#P` reader macro followed by the namestring. A namestring is a string which represents a pathname. The syntax of namestrings for logical pathnames is well explained in the ANSI [ANSI, see [Bibliography], page 203] and it can be roughly summarized as follows:

```
[hostname:][;][directory-item;]0 or more[name][.type[.version]]
  hostname = word
  directory-item = wildcard-word
  type, name = wildcard-word without dots
```

Here, *wildcard-word* is a sequence of any character excluding `#\Null` and dots. *word* is like a *wildcard-word* but asterisks are excluded.

The way ECL parses a namestring is by first looking for the *hostname* component in the previous template. If it is found and it corresponds to a previously defined logical hostname, it assumes that the namestring corresponds to a logical pathname. If *hostname* is not found or it is not a logical hostname, then ECL tries the physical pathname syntax

```
[device:][[//hostname]/][directory-item/]0 or more[name][.type]
  device, hostname = word
  directory-item = wildcard-word
  type = wildcard-word without dots
  name = [.]wildcard-word
```

If this syntax also fails, then the namestring is not a valid pathname string and a `parse-error` will be signaled.

It is important to remark that in ECL, all physical namestrings result into pathnames with a version equal to `:newest`. Pathnames which are not logical and have any other version (i. e. `nil` or a number), cannot be printed readably, but can produce a valid namestring which results of ignoring the version.

Finally, an important rule applies to physical namestrings: if a namestring contains one or more periods '.', the last period separates the namestring into the file name and the filetype. However, a namestring with a single leading period results in a name with a period in it. This is for compatibility with Unix filenames such as `.bashrc`, where the leading period indicates that the file is hidden.

The previous rule has in important consequence, because it means that if you want to create a pathname without a name, you have to do it explicitely. In other words, `".*"` is equivalent to `(make-pathname :name ".*" :type nil)`, while `(make-pathname :name nil :type :wild)` creates a pathname whose type is a wildcard.

The following table illustrates how the physical pathnames work with practical examples.

Namestring	Name	Type	Directory	Device
"foo.lsp"	"foo"	"lsp"	nil	nil
".bashrc"	".bashrc"	nil	nil	nil
".ecl.lsp"	".ecl"	"lsp"	nil	nil
"foo.*"	"foo"	:wild	nil	nil
"*.*"	:wild	:wild	nil	nil
"ecl/build/bare.lsp"	"bare"	"lsp"	(:relative "ecl" "build")	nil
"ecl/build/"	nil	nil	(:relative "ecl" "build")	nil
"../../ecl/build/"	nil	nil	(:relative :up :up "ecl" "build")	nil
"/etc/"	nil	nil	(:absolute "etc")	nil
"C:/etc/"	nil	nil	(:absolute "etc")	"C"
".*"	".*"	nil	nil	nil
#.(make-pathname :type "*")	nil	:wild	nil	nil

Table 2.8: Examples of physical namestrings

2.17.2 Wild pathnames and matching

ECL accepts four kind of wildcards in pathnames.

- A single wildcard in a directory component, file name, type or version is parsed as the `:wild` value. See for instance `"*.*"`, `"/home/*/.bashrc"`, etc
- A double wildcard in a directory component, such as in `"/home/**/"` is parsed as the `:wild-inferiors`, and matches any number of directories, even nested ones, such as: `/home/`, `/home/jlr`, `/home/jlr/lib`, etc.
- An isolated wildcard `"log*.txt"` matches any number of characters: `log.txt`, `log_back.txt`, etc.
- A question mark `"log?.txt"` matches a single character: `log1.txt`, `log2.txt`...

The matching rules in Common Lisp and ECL are simple but have some unintuitive consequences when compared to Unix/DOS rules. The most important one is that directories must always end with a trailing slash `/`, as in `#p"/my/home/directory/"`. Second to that, `nil` values can only be matched by `nil` and `:wild`. Hence, `"*"` can only match files without file type. For some examples see Section 2.18.1 [Files - Dictionary], page 72.

2.17.3 C Reference

2.17.3.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	C function
directory-namestring	cl_object cl_directory_namestring(cl_object pathname)

enough-namestring	cl_object cl_enough_namestring(cl_narg narg, cl_object pathname, ...)
file-namestring	cl_object cl_file_namestring(cl_object pathname)
host-namestring	cl_object cl_host_namestring(cl_object pathname)
load-logical-pathname-translations	cl_object cl_load_logical_pathname_translations(cl_object host)
logical-pathname-translations	cl_object cl_logical_pathname_translations(cl_object host)
logical-pathname	cl_object cl_logical_pathname(cl_object pathspec)
make-pathname	cl_object cl_make_pathname(cl_narg narg, ...)
merge-pathnames	cl_object cl_merge_pathnames(cl_narg narg, cl_object pathname,...)
namestring	cl_object cl_namestring(cl_object pathname)
parse-namestring	cl_object cl_parse_namestring(cl_narg narg, cl_object thing, ...)
pathname	cl_object cl_pathname(cl_object pathspec)
pathname-device	cl_object cl_pathname_device(cl_narg narg, cl_object pathname, ...)
pathname-directory	cl_object cl_pathname_directory(cl_narg narg, cl_object pathname, ...)
pathname-host	cl_object cl_pathname_host(cl_narg narg, cl_object pathname, ...)
pathname-match-p	cl_object cl_pathname_match_p(cl_object pathname, cl_object wildcard)
pathname-name	cl_object cl_pathname_name(cl_narg narg, cl_object pathname, ...)
pathname-type	cl_object cl_pathname_type(cl_narg narg, cl_object pathname, ...)
pathname-version	cl_object cl_pathname_version(cl_object pathname)
pathnamep	cl_object cl_pathnamep(cl_object object)
translate-logical-pathname	cl_object cl_translate_logical_pathname(cl_narg narg, cl_object pathname, ...)
translate-pathname	cl_object cl_translate_pathname(cl_narg narg, cl_object source, cl_object from_wildcard, cl_object to_wildcard, ...)
wild-pathname-p	cl_object cl_wild_pathname_p(cl_narg narg, cl_object pathname, ...)

2.18 Files

2.18.1 Dictionary

`directory` *pathspec* [Function]

This function does not have any additional arguments other than the ones described in ANSI [ANSI, see [Bibliography], page 203]. To list files and directories, it follows the rules for matching pathnames described in Section 2.17.2 [Filenames - Wild pathnames and matching], page 71. In short, you have the following practical examples:

Argument	Meaning
`"/home/jlr/*.*"`	List all files in directory `/home/jlr/` Note that it lists only files, not directories!
`"/home/jlr/*"`	Same as before, but only files without type.
`"/home/jlr/*/"`	List all directories contained in `/home/jlr/`. Nested directories are not navigated.
`"/home/jlr/**/*.*"`	List all files in all directories contained in `/home/jlr/`, recursively. Nested directories are navigated.

Table 2.9: Examples of using `directory`

`rename-file` *filespec new-name* **&key** (*if-exists* **:error**) [Function]

In addition to the arguments described in ANSI [ANSI, see [Bibliography], page 203], the `rename-file` function in ECL has an `:if-exists` keyword argument that specifies what happens when a file with the new name already exists. Valid values of this argument are:

Argument	Behaviour of the `rename-file` function
`:error`	Signal an error
`:supersede`, `t`	Overwrite the existing file
`nil`	Don't overwrite the existing file, don't signal an error

2.18.2 C Reference

2.18.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
delete-file	cl_object cl_delete_file(cl_object filespec)
directory	cl_object cl_directory(cl_narg narg, cl_object pathspec, ...)
ensure-directories-exist	cl_object cl_ensure_directories_exist(cl_narg narg, cl_object pathspec, ...)
file-author	cl_object cl_file_author(cl_object pathspec)
file-error-pathname	[Only in Common Lisp]
file-write-date	cl_object cl_file_write_date(cl_object pathspec)
probe-file	cl_object cl_probe_file(cl_object pathspec)
rename-file	cl_object cl_rename_file(cl_narg narg, cl_object filespec, cl_object new_name, ...)
truename	cl_object cl_truename(cl_object filespec)

2.19 Streams

2.19.1 ANSI Streams

2.19.1.1 Supported types

ECL implements all stream types described in ANSI [ANSI, see [Bibliography], page 203]. Additionally, when configured with option `--enable-clos-streams`, ECL includes a version of Gray streams where any object that implements the appropriate methods (`stream-input-p`, `stream-read-char`, etc) is a valid argument for the functions that expect streams, such as `read`, `print`, etc.

2.19.1.2 Element types

ECL distinguishes between two kinds of streams: character streams and byte streams. *Character streams* only accept and produce characters, written or read one by one, with `write-char` or `read-char`, or in chunks, with `write-sequence` or any of the Lisp printer functions. Character operations are conditioned by the external format, as described in Section 2.19.1.3 [Streams - External formats], page 74.

ANSI Common Lisp also supports binary streams. Here input and output is performed in chunks of bits. Binary streams are created with the function open passing as argument a subtype of integer and the implementation is free to round up that integer type to the closest size it supports. In particular ECL rounds up the size to a multiple of a byte. For example, the form `(open "foo.bin" :direction :output :element-type '(unsigned-byte 13))`, will open the file `foo.bin` for writing, using 16-bit words as the element type.

2.19.1.3 External formats

An *external format* is an encoding for characters that maps character codes to a sequence of bytes, in a one-to-one or one-to-many fashion. External formats are also known as "character encodings" in the programming world and are an essential ingredient to be able to read and write text in different languages and alphabets.

ECL has one of the most complete supports for *external formats*, covering all of the usual codepages from the Windows and Unix world, up to the more recent UTF-8, UCS-2 and UCS-4 formats, all of them with big and small endian variants, and considering different encodings for the newline character.

However, the set of supported external formats depends on the size of the space of character codes. When ECL is built with Unicode support (the default option), it can represent all known characters from all known codepages, and thus all external formats are supported. However, when ECL is built with the restricted character set, it can only use one codepage (the one provided by the C library), with a few variants for the representation of end-of-line characters.

In ECL, an external format designator is defined recursively as either a symbol or a list of symbols. The grammar is as follows

```
external-format-designator :=
   symbol |
   ( {symbol}+ )
```

and the table of known symbols is shown below. Note how some symbols (`:cr`, `:little-endian`, etc.) just modify other external formats.

Symbols	Codepage or encoding	Unicode required
`:cr`	`#\Newline` is `Carriage Return`	No
`:crlf`	`#\Newline` is `Carriage Return` followed by `Linefeed`	No
`:lf`	`#\Newline` is `Linefeed`	No
`:little-endian`	Modify UCS to use little-endian encoding.	No
`:big-endian`	Modify UCS to use big-endian encoding.	No
`:utf-8 :utf8`	Unicode UTF-8	Yes
`:ucs-2 :ucs2 :utf-16 :utf16 :unicode`	UCS-2 encoding with BOM. Defaults to big-endian when writing or if no BOM is detected when reading.	Yes
`:ucs-2le :ucs2le :utf-16le`	UCS-2 with little-endian encoding	Yes
`:ucs-2be :ucs2be :utf-16be`	UCS-2 with big-endian encoding	Yes
`:ucs-4 :ucs4 :utf-32 :utf32`	UCS-4 encoding with BOM. Defaults to big-endian when writing or if no BOM is detected when reading.	Yes
`:ucs-4le :ucs4le :utf-32le`	UCS-4 with little-endian encoding	Yes
`:ucs-4be :ucs4be :utf-32be`	UCS-4 with big-endian encoding	Yes
`:iso-8859-1 :iso8859-1 :latin-1 :cp819 :ibm819`	Latin-1 encoding	Yes
`:iso-8859-2 :iso8859-2 :latin-2 :latin2`	Latin-2 encoding	Yes
`:iso-8859-3 :iso8859-3 :latin-3 :latin3`	Latin-3 encoding	Yes
`:iso-8859-4 :iso8859-4 :latin-4 :latin4`	Latin-4 encoding	Yes
`:iso-8859-5 :cyrillic`	Latin-5 encoding	Yes
`:iso-8859-6 :arabic :asmo-708 :ecma-114`	Latin-6 encoding	Yes
`:iso-8859-7 :greek8 :greek :ecma-118`	Greek encoding	Yes
`:iso-8859-8 :hebrew`	Hebrew encoding	Yes
`:iso-8859-9 :latin-5 :latin5`	Latin-5 encoding	Yes
`:iso-8859-10 :iso8859-10 :latin-6 :latin6`	Latin-6 encoding	Yes
`:iso-8859-13 :iso8859-13 :latin-7 :latin7`	Latin-7 encoding	Yes
`:iso-8859-14 :iso8859-14 :latin-8 :latin8`	Latin-8 encoding	Yes
`:iso-8859-15 :iso8859-15 :latin-9 :latin9`	Latin-7 encoding	Yes
`:dos-cp437 :ibm-437`	IBM CP 437	Yes
`:dos-cp850 :ibm-850 :cp850`	Windows CP 850	Yes
`:dos-cp852 :ibm-852`	IBM CP 852	Yes

2.19.2 Dictionary

2.19.2.1 File Stream Extensions

`open` *filespec* **&key** *direction element-type if-exists if-does-not-exist external-format close-on-exec nonblock* [Function]

Additional options for open include:

`:close-on-exec`
: Child processes don't inherit a copy of this stream: new processes created by `fork` and `exec` (for example by calling `ext:run-program`) close the stream after calling `exec`. Defaults to `t`.

`:nonblock`
: Open fifos or device files in nonblocking mode. Defaults to `nil`.

These options are ignored on operating systems which do not support them.

`ext:set-buffering-mode` *stream mode* [Function]

Control the buffering mode of a stream

Synopsis

stream an ANSI stream

mode one of `nil`, `:none`, `:line`, `:line-buffered`, `:full` or `:full-buffered`

returns The supplied stream

Description If *mode* is `nil` or `:none`, *stream* will not be buffered, if it is `:line` or `:line-buffered` resp. `:full` or `:fully-buffered`, *stream* will be line resp. fully buffered. If the stream does not support buffering, nothing will happen.

`ext:file-stream-fd` *file-stream* [Function]

Return the POSIX file descriptor of *file-stream* as an integer

2.19.2.2 Binary Stream Extensions

`ext:unread-byte` *stream byte* [Function]

Similar to `unread-char` but for unreading the last element to the input binary stream. See also `gray:stream-unread-byte`.

`ext:peek-byte` *stream eof-value* [Function]

Similar to `peek-char` but for peeking the next element from the output binary stream. See also `gray:stream-peek-byte`.

2.19.2.3 External Format Extensions

`ext:*default-external-format*` [Variable]

Default external format to use for reading from streams, dealing with filenames, etc. The default is to use utf-8 encoding if ECL is built with Unicode support.

`ext:all-encodings` [Function]

Return a list of all supported external formats

`ext:character-coding-error` [Condition]
Character coding error

Class Precedence List `ext:character-coding-error`, `error`, `serious-condition`, `condition`, `t`

Methods

`ext:character-coding-error-external-format` *condition* [Function]

returns The external format of *condition*

Description Superclass of `ext:character-encoding-error` and `ext:character-decoding-error`.

`ext:character-encoding-error` [Condition]
Character encoding error

Class Precedence List `ext:character-encoding-error`, `ext:character-coding-error`, `error`, `serious-condition`, `condition`, `t`

Methods

`ext:character-encoding-error-code` *condition* [Function]

returns The character code of the character, which can't be encoded

Description Condition for characters, which can't be encoded with some external format.

`ext:character-decoding-error` [Condition]
Character decoding error

Class Precedence List `ext:character-decoding-error`, `ext:character-coding-error`, `error`, `serious-condition`, `condition`, `t`

Methods

`ext:character-decoding-error-octects` *condition* [Function]

returns A list of integers with the values of the `unsigned char`'s which can't be decoded.

Description Condition for characters, which can't be decoded with some external format.

`ext:stream-encoding-error` [Condition]
Stream encoding error

Class Precedence List `ext:stream-encoding-error`, `ext:character-encoding-error`, `ext:character-coding-error`, `stream-error`, `error`, `serious-condition`, `condition`, `t`

Description This condition is signaled when trying to write a character to a stream, which can't be encoded with the streams external format.

`ext:stream-decoding-error` [Condition]

Stream decoding error

Class Precedence List `ext:stream-decoding-error`, `ext:character-decoding-error`, `ext:character-coding-error`, `stream-error`, `error`, `serious-condition`, `condition`, `t`

Description This condition is signaled when trying to read a character from a stream, which can't be decoded with the streams external format.

`ext:encoding-error` *stream external-format code* [Function]

Signal a `ext:stream-encoding-error` with the given *external-format* and *code*. Make a restart available so that the error can be ignored or the character can be replaced with a different one.

`ext:decoding-error` *stream external-format octects* [Function]

Signal a `ext:stream-decoding-error` with the given *external-format* and *octets*. Make a restart available so that the error can be ignored or the octets can be replaced with a character.

2.19.2.4 Sequence Streams

`ext:sequence-stream` [System Class]

Class Precedence List `ext:sequence-stream`, `stream`, `t`

Description Sequence streams work similar to string streams for vectors. The supplied vectors that the streams read from or write to must have a byte sized element type, i.e. `(signed-byte 8)`, `(unsigned-byte 8)` or `base-char`.

The semantics depend on the vector element type and the external format of the stream. If no external format is supplied and the element type is an integer type, the stream is a binary stream and accepts only integers of the same type as the element type of the vector. Otherwise, the stream accepts both characters and integers and converts them using the given external format. If the element type is `base-char`, the elements of the vectors are treated as bytes. This means that writing a character may use multiple elements of the vector, whose `char-code`s will be equal to the values of the bytes comprising the character in the given external format.

`ext:make-sequence-input-stream` *vector* **&key** (*start* **0**) (*end* **nil**) (*external-format* **nil**) [Function]

Create a sequence input stream with the subsequence bounded by *start* and *end* of the given vector.

When *end* is `nil`, then the end bound is the vector's length. When the vector is adjusted the end bound is updated as well.

`ext:make-sequence-output-stream` *vector* **&key** (*external-format* **nil**) [Function]

Create a sequence output stream.

Example:

Using sequence streams to convert to a UTF8 encoded base string

```
CL-USER> (defvar *output* (make-array 20 :element-type 'base-char :adjustable t :fill-
```

```
*OUTPUT*
CL-USER> (defvar *stream* (ext:make-sequence-output-stream *output* :external-format :
*STREAM*
CL-USER> (write-string "Spätzle mit Soß'" *stream*)
"Spätzle mit Soß'"
CL-USER> *output*
"SpÃ¤tzle mit SoÃ\237'"
```

2.19.3 C Reference

2.19.3.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
broadcast-stream-streams	cl_object cl_broadcast_stream_streams(cl_object broadcast_stream)
clear-input	cl_object cl_clear_input(cl_narg narg, ...)
clear-output	cl_object cl_clear_output(cl_narg narg, ...)
close	cl_object cl_close(cl_narg narg, cl_object stream, ...)
concatenated-stream-streams	cl_object cl_concatenated_stream_streams(cl_object concatenated_stream)
echo-stream-input-stream	cl_object cl_echo_stream_input_stream(cl_object echo_stream)
echo-stream-output-stream	cl_object cl_echo_stream_output_stream(cl_object echo_stream)
file-length	cl_object cl_file_position(cl_narg narg, cl_object file_stream, ...)
file-position	cl_object cl_file_position(cl_object stream)
file-string-length	cl_object cl_file_string_length(cl_object stream, cl_object object)
finish-output	cl_object cl_finish_output(cl_narg narg, ...)
force-output	cl_object cl_force_output(cl_narg narg, ...)
fresh-line	cl_object cl_fresh_line(cl_narg narg, ...)
get-output-stream-string	cl_object cl_get_output_stream_string(cl_object string_output_stream)
input-stream-p	cl_object cl_input_stream_p(cl_object stream)
interactive-stream-p	cl_object cl_interactive_stream_p(cl_object stream)
listen	cl_object cl_listen(cl_narg narg, cl_object stream, ...)
make-broadcast-stream	cl_object cl_make_broadcast_stream(cl_narg narg, ...)
make-concatenated-stream	cl_object cl_make_concatenated_stream(cl_narg narg, ...)
make-echo-stream	cl_object cl_make_echo_stream(cl_object input, cl_object output)
make-string-input-stream	cl_object cl_make_string_input_stream(cl_narg narg, cl_object string, ...)
make-string-output-stream	cl_object cl_make_string_output_stream(cl_narg narg, ...)

make-two-way-stream	cl_object cl_make_two_way_stream(cl_object input, cl_object output)
make-synonym-stream	cl_object cl_make_synonym_stream(cl_object symbol)
open	cl_object cl_open(cl_narg narg, cl_object filespec, ...)
open-stream-p	cl_object cl_open_stream_p(cl_object stream)
output-stream-p	cl_object cl_output_stream_p(cl_object stream)
peek-char	cl_object cl_peek_char(cl_narg narg, ...)
read-byte	cl_object cl_read_byte(cl_narg narg, cl_object stream, ...)
read-char	cl_object cl_read_char(cl_narg narg, ...)
read-char-no-hang	cl_object cl_read_char_no_hang(cl_narg narg, ...)
read-line	cl_object cl_read_line(cl_narg narg, ...)
read-sequence	cl_object cl_read_sequence(cl_narg narg, cl_object sequence, cl_object stream, ...)
stream-element-type	cl_object cl_stream_element_type(cl_object stream)
stream-error-stream	[Only in Common Lisp]
stream-external-format	cl_object cl_stream_external_format(cl_object stream)
(setf stream-external-format)	cl_object si_stream_external_format_set(cl_object stream, cl_object format)
streamp	cl_object cl_streamp(cl_object object)
synonym-stream-symbol	cl_object cl_synonym_stream_symbol(cl_object synonym_stream)
terpri	cl_object cl_terpri(cl_narg narg, ...)
two-way-stream-input-stream	cl_object cl_two_way_stream_input_stream(cl_object two_way_stream)
two-way-stream-output-stream	cl_object cl_two_way_stream_output_stream(cl_object two_way_stream)
unread-char	cl_object cl_unread_char(cl_narg narg, cl_object character, ...)
write-byte	cl_object cl_write_byte(cl_object byte, cl_object stream)
write-char	cl_object cl_write_char(cl_narg narg, cl_object character, ...)
write-line	cl_object cl_write_line(cl_narg narg, cl_object string, ...)
write-string	cl_object cl_write_string(cl_narg narg, cl_object string, ...)
write-sequence	cl_object cl_write_sequence(cl_narg narg, cl_object sequence, cl_object stream, ...)
y-or-n-p	cl_object cl_y_or_n_p(cl_narg narg, ...)
yes-or-no-p	cl_object cl_yes_or_no_p(cl_narg narg, ...)

2.20 Printer

In all situations where the rules are well specified, ECL prints objects according to ANSI [ANSI, see [Bibliography], page 203]. The specification leaves however a number of cases as implementation dependent behavior. The output of ECL in those cases is summarized in Table 2.11. Except for the types `character` and `random-state`, most of those examples regard non-standard written forms `#<...>` cannot be read back using `read`. These printed representations are just informative and should not be used to reconstruct or compare objects.

Lisp type	Format	Remarks
package	#<package *name*>	
random-state	#<random-state *array*>	
bitvector	#<bit-vector *unique-id*>	Only when *print-array* is false.
vector	#<vector *unique-id*>	Only when *print-array* is false.
array	#<array *unique-id*>	Only when *print-array* is false.
hash-table	#<hash-table *unique-id*>	
readtable	#<readtable *unique-id*>	
interpreted function	#<bytecompiled-function *name-or-id*>	Name is a symbol.
machine compiled function	#<compiled-function *name*>	Name is a symbol.
input-stream	#<input stream "*filename*">	An stream that reads from *filename*.
output-stream	#<output stream "*filename*">	An stream that writes to *filename*.
probe-stream	#<probe stream "*filename*">	
string-input-stream	#<string-input stream from "*string-piece*">	The string is the text left to be read.
string-output-stream	#<string-output stream *unique-id*>	
two-way-stream	#<two-way stream *unique-id*>	
echo-stream	#<echo stream *unique-id*>	
synonym-stream	#<synonym stream to *symbol*>	
broadcast-stream	#<broadcast stream *unique-id*>	
concatenated-stream	#<concatenated stream *unique-id*>	
closed-stream	#<closed ...>	The dots denote any of the above stream forms.

Table 2.11: Implementation-specific printed representation

2.20.1 C Reference

2.20.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
copy-pprint-dispatch	cl_object cl_copy_pprint_dispatch(cl_narg narg, ...)
pprint-dispatch	cl_object cl_pprint_dispatch(cl_narg narg, cl_object object, ...)
pprint-fill	cl_object cl_pprint_fill(cl_narg narg, cl_object stream, cl_object object, ...)
pprint-linear	cl_object cl_pprint_linear(cl_narg narg, cl_object stream, cl_object object, ...)
pprint-tabular	cl_object cl_pprint_tabular(cl_narg narg, cl_object stream, cl_object object, ...)
pprint-indent	cl_object cl_pprint_indent(cl_narg narg, cl_object relative_to, cl_object n, ...)
pprint-newline	cl_object cl_pprint_newline(cl_narg narg, cl_object kind, ...)
pprint-tab	cl_object cl_pprint_tab(cl_narg narg, cl_object kind, cl_object colnum, cl_object colinc, ...)
print-object	[Only in Common Lisp]
set-pprint-dispatch	cl_object cl_set_pprint_dispatch(cl_narg narg, cl_object ype_spec, cl_object function, ...)
write	cl_object cl_write(cl_narg narg, cl_object object, ...)
prin1	cl_object cl_prin1(cl_narg narg, cl_object object, ...)
princ	cl_object cl_princ(cl_narg narg, cl_object object, ...)
print	cl_object cl_print(cl_narg narg, cl_object object, ...)
pprint	cl_object cl_pprint(cl_narg narg, cl_object object, ...)
write-to-string	cl_object cl_write_to_string(cl_narg narg, cl_object object, ...)
prin1-to-string	cl_object cl_prin1_to_string(cl_object object)
princ-to-string	cl_object cl_princ_to_string(cl_object object)
print-not-readable-object	[Only in Common Lisp]
format	cl_object cl_format(cl_narg narg, cl_object stream, cl_object string, ...)

2.21 Reader

2.21.1 `*read-supress*`

The behaviour of `*read-supress*` is not fully specified in the standard. ECL tries to be as liberal as possible in the syntax that it accepts when `*read-suppress*` is true. Errors are only signaled in the following cases:

- End of file
- Unbalanced parantheses
- Invalid dispatching macro characters such as `<` or `)` (Undefined dispatching macro characters don't produce errors)

2.21.2 C Reference

2.21.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
copy-readtable	cl_object cl_copy_readtable(cl_narg narg, ...)
make-dispatch-macro-character	cl_object cl_make_dispatch_macro_character(cl_narg narg, cl_object char, ...)
read	cl_object cl_read(cl_narg narg, ...)
read-preserving-whitespace	cl_object cl_read_preserving_whitespace(cl_narg narg, ...)
read-delimited-list	cl_object cl_read_delimited_list(cl_narg narg, cl_object char, ...)
read-from-string	cl_object cl_read_from_string(cl_narg narg, cl_object string, ...)
readtable-case	cl_object cl_readtable_case(cl_object readtable)
(setf readtable-case)	cl_object si_readtable_case_set(cl_object readtable, cl_object mode)
readtablep	cl_object cl_readtablep(cl_object object)
get-dispatch-macro-character	cl_object cl_get_dispatch_macro_character(cl_narg narg, cl_object disp_char, cl_object sub_char, ...)
set-dispatch-macro-character	cl_object cl_set_dispatch_macro_character(cl_narg narg, cl_object disp_char, cl_object sub_char, cl_object function, ...)
get-macro-character	cl_object cl_get_macro_character(cl_narg narg, cl_object char, ...)
set-macro-character	cl_object cl_set_macro_character(cl_narg narg, cl_object char, cl_object function, ...)
set-syntax-from-char	cl_object cl_set_syntax_from_char(cl_narg narg, cl_object to_char, cl_object from_char, ...)

2.22 System construction

2.22.1 Dictionary

2.22.1.1 Extensions

See also Section 3.1 [System building], page 89.

`compile-file` *input-file* **&key** *output-file verbose print external-format target system-p load c-file h-file data-file* [Function]

Additional options for compile-file include:

`:system-p`
: Create an object file (.o)

`:target`
: Cross compilation target, see Section 3.1.3 [Cross compilation], page 97. Can be either the target information obtained by calling `c:read-target-info` or a pathname pointing to the installation directory of the target ECL for which to compile.

`:load`
: Load the output file after compilation. If a `:target` argument is given alongside `:load`, load the source file in the target environment.

`:c-file, :h-file, :data-file`
: Boolean flags controlling whether to keep temporary files created by the compiler

`compile-file-pathname` *input-file* **&key** *output-file type system-p* [Function]

Additional options for compile-file-pathname include:

`:type`
: Specify the type of output file. One of `:fasl` (default, alias `:fas`), `:c`, `:h`, `:data`, `:object`, `:program`, `:shared-library` (alias `:dll`), `:static-library` (alias `:library`, `:lib`), `:precompiled-header` or `:import-library` (MSVC only)

`:system-p`
: If no type is supplied, default to `:object` type instead of `:fasl`

`with-compilation-unit` (**&key** *override target*) **&body** *body* [Macro]

Additional options for with-compilation-unit include:

`:target`
: Cross compilation target, see Section 3.1.3 [Cross compilation], page 97. Any compilation operation produces outputs for the specified target system. This includes compile-file, disassemble and all ECL specific functions described in the section on Section 3.1.1 [Compiling with ECL], page 89. The only exception is compile which continues to work as usual.

2.22.2 C Reference

2.22.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
compile-file	[Only in Common Lisp]
compile-file-pathname	[Only in Common Lisp]
load	cl_object cl_load(cl_narg narg, cl_object pathname, ...)
provide	cl_object cl_provide(cl_object module_name)
require	cl_object cl_require(cl_narg narg, cl_object module_name, ...)

2.23 Environment

2.23.1 Dictionary

`disassemble` *function-designator** [Function]

Display the assembly code of a function

Synopsis

function-designator
: A symbol which is bound to a function in the global environment, or a lambda form

Description As specified in ANSI [ANSI, see [Bibliography], page 203] this function outputs the internal representation of a compiled function, or of a lambda form, as it would look after being compiled.

ECL only has a particular difference: it has two different compilers, one based on bytecodes and one based on the C language. The output will thus depend on the arguments and on which compiler is active at the moment in which this function is run.

- If the argument is a bytecompiled function or a lambda form, it will be processed by the active compiler and the appropriate output (bytecodes or C) will be shown.
- If the argument is a C-compiled form, disassembling the function by showing its C source code is not possible, since that would require saving not only the lambda form of the function, but also the precise configuration of the compiler when the function was compiled. Hence no output will be shown.

`ed` *x?* [Function]

Invoke an editor on the file or object specified by *x*.

Synopsis (ed *x?*)

x `nil`, a file path, or an object to edit.

Description Starts the editor (on a file or an object if named). Functions from the list *ext:*ed-functions** are called in order with *x* as an argument until one of them returns non-`nil`; these functions are responsible for signalling a `file-error` to indicate failure to perform an operation on the file system. If no function returns a non-`nil` value or *ext:*ed-functions** is `nil` then a `simple-error` will be signalled.

The Common Lisp specification states that the *x* argument is either `nil`, a function name, or an instance of *string* or *pathname* and that a *type-error* may be signalled if is not one of these types. ECL does not check the type of *x* and thus permits any object to be passed to the hook functions. This allows for the possibility of editing other objects that have a representation in source code such as class definitions. Therefore, the hook functions should not make any assumptions about the type of *x* and should instead return `nil` if there is not an approriate edit method for a specific value of *x*.

By default *ext:*ed-functions** contains a single function that attempts to run the program named in the environment variable `EDITOR`. If this environment variable is not set then the fallback program is `vi`.

`trace` *function-name** [Macro]

Follow the execution of functions

Synopsis (trace *function-name**)

function-name
{*symbol* | (*symbol* [*option form*]*)}

symbol A symbol which is bound to a function in the global environment. Not evaluated.

option One of `:break`, `:break-after`, `:cond-before`, `:cond-after`, `:cond`, `:print`, `:print-after`, `:step`

form A lisp form evaluated in an special environment (or a list of forms for `:print` and `:print-after`).

returns List of symbols with traced functions.

Description Causes one or more functions to be traced. Each *function-name* can be a symbol which is bound to a function, or a list containing that symbol plus additional options. If the function bound to that symbol is called, information about the arguments and output of this function will be printed. Trace options will modify the amount of information and when it is printed.

Not that if the function is called from another function compiled in the same file, tracing might not be enabled. If this is the case, to enable tracing, recompile the caller with a `notinline` declaration for the called function.

`trace` returns a name list of those functions that were traced by the call to `trace`. If no *function-name* is given, `trace` simply returns a name list of all the currently traced functions.

Trace options cause the normal printout to be suppressed, or cause extra information to be printed. Each option is a pair of an option keyword and a value form. If an already traced function is traced again, any new options replace the old options and a warning might be printed. The lisp *form* accompanying the option is evaluated in an environment where *sys::args* contains the list of arguments to the function.

The following options are defined:

`:cond`, `:cond-before`, `:cond-after`
: If `:cond-before` is specified, then `trace` does nothing unless *form* evaluates to true at the time of the call. `:cond-after` is similar, but suppresses the initial printout, and is tested when the function returns. `:cond` tries both before and after.

`:step` If *form* evaluates to true, the stepper is entered.

`:break`, `:break-after`
: If specified, and *form* evaluates to true, then the debugger is invoked at the start of the function or at the end of the function according to the respective option.

`:print`, `:print-after`
: In addition to the usual printout, the result of evaluating each entry of the list of forms contained in *form* is printed at the start of the function or at the end of the function, depending on the option.

See also the following example:

```
> (defun abc (x)
    (if (>= x 10)
        x
        (abc (+ x (abc (1+ x))))))
> (trace abc)

> (abc 9)
```

```
1> (ABC 9)
| 2> (ABC 10)
| <2 (ABC 10)
| 2> (ABC 19)
| <2 (ABC 19)
<1 (ABC 19)
19
> (untrace abc)

(ABC)
;; Break if the first argument of the function is greater than 10
> (trace (abc :break (>= (first si::args) 10)))

((ABC :BREAK (>= (FIRST SI::ARGS) 10)))
> (abc 9)
1> (ABC 9)
| 2> (ABC 10)

Condition of type: SIMPLE-CONDITION
tracing ABC
Available restarts:

1. (CONTINUE) Return from BREAK.
2. (RESTART-TOPLEVEL) Go back to Top-Level REPL.

Broken at ABC. In: #<process TOP-LEVEL 0x1842f80>.
>>
```

2.23.2 C Reference

2.23.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	**C function**
decode-universal-time	cl_object cl_decode_universal_time(cl_narg narg, cl_object universal_time, ...)
encode-universal-time	cl_object cl_encode_universal_time(cl_narg narg, cl_object second, cl_object minute, cl_object hour, cl_object date, cl_object month, cl_object year, ...)
get-universal-time	cl_object cl_get_universal_time(void)
get-decoded-time	cl_object cl_get_decoded_time(void)
sleep	cl_object cl_sleep(cl_object seconds)
apropos	cl_object cl_apropos(cl_narg narg, cl_object string, ...)
apropos-list	cl_object cl_apropos_list(cl_narg narg, cl_object string, ...)
describe	cl_object cl_describe(cl_narg narg, cl_object object, ...)
describe-object	[Only in Common Lisp]

get-internal-real-time	cl_object cl_get_internal_real_time(void)
get-internal-run-time	cl_object cl_get_internal_run_time(void)
disassemble	[Only in Common Lisp]
documentation	[Only in Common Lisp]
room	[Only in Common Lisp]
ed	[Only in Common Lisp]
inspect	cl_object cl_inspect(cl_object object)
dribble	cl_object cl_dribble(cl_narg narg, ...)
lisp-implementation-type	cl_object cl_lisp_implementation_type(void)
lisp-implementation-version	cl_object cl_lisp_implementation_version(void)
short-site-name	cl_object cl_short_site_name()
long-site-name	cl_object cl_long_site_name()
machine-instance	cl_object cl_machine_instance()
machine-type	cl_object cl_machine_type()
machine-version	cl_object cl_machine_version()
software-type	cl_object cl_software_type()
software-version	cl_object cl_software_version()
user-homedir-pathname	cl_object cl_user_homedir_pathname(cl_narg narg, ...)

3 Extensions

3.1 System building

3.1.1 Compiling with ECL

In this section we will introduce topics on compiling Lisp programs. ECL is especially powerful on combining lisp programs with C programs. You can embed ECL as a lisp engine in C programs, or call C functions via Section 3.3 [Foreign Function Interface], page 106. We explain file types generated by some compilation approaches. For the examples, a GNU/Linux system and gcc as a development environment are assumed.

You can generate the following files with ECL:

1. Portable FASL file (.fasc)
2. Native FASL file (.fas, .fasb)
3. Object file (.o)
4. Static library
5. Shared library
6. Executable file

Relations among them are depicted below:

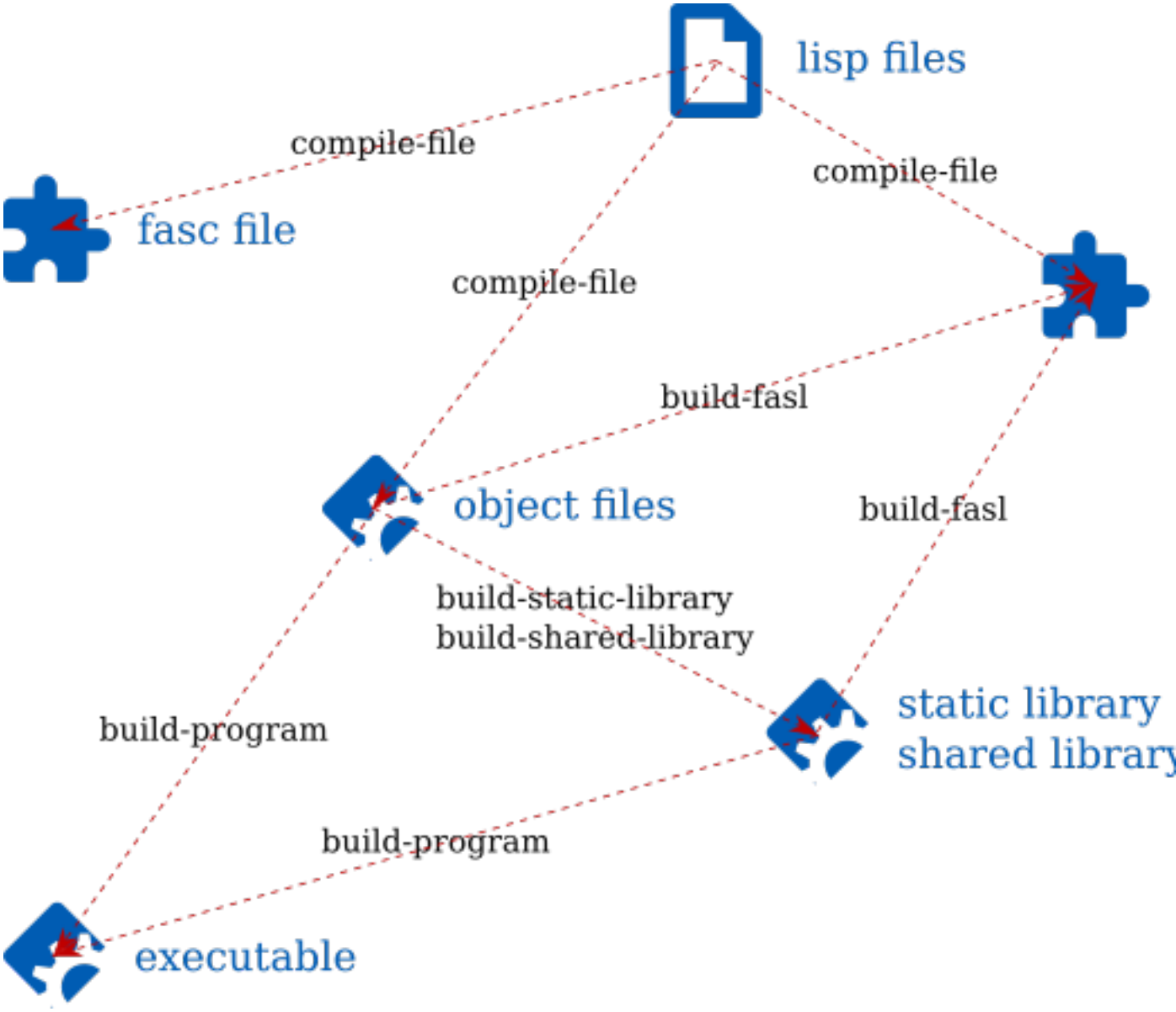


Figure 3.1: Build file types

3.1.1.1 Portable FASL

ECL provides two compilers (bytecodes compiler, and C/C++ compiler). Portable FASL files are built from source lisp files by the bytecodes compiler. Generally FASC files are portable across architectures and operating systems providing a convenient way of shipping portable modules. Portable FASL files may be concatenated, what leads to bundles. FASC files are faster to compile, but generally slower to run.

```
;; install bytecodes compiler
```

```
(ext:install-bytecodes-compiler)

;; compile hello.lisp file to hello.fasc
(compile-file "hello1.lisp")
(compile-file "hello2.lisp")

;; reinitialize C/C++ compiler back
(ext:install-c-compiler)

;; FASC file may be loaded dynamically from lisp program
(load "hello1.fasc")

;; ... concatenated into a bundle with other FASC
(with-open-file (output "hello.fasc"
                        :direction :output
                        :if-exists :supersede)
  (ext:run-program
   "cat" '("hello1.fasc" "hello2.fasc")  :output output))

;; ... and loaded dynamically from lisp program
(load "hello.fasc")
```

3.1.1.2 Native FASL

If you want to make a library which is loaded dynamically from a lisp program, you should choose the fasl file format. Under the hood native fasls are just shared library files.

This means you can load fasl files with `dlopen` and initialize it by calling a init function from C programs, but this is not an intended usage. The recommended usage is to load fasl files by calling the load lisp function. To work with *Native FASL files* ECL has to be compiled with `--enable-shared` configure option (enabled by default).

Creating a fasl file from one lisp file is very easy.

```
(compile-file "hello.lisp")
```

To create a fasl file from more lisp files, firstly you have to compile each lisp file into an object file, and then combine them with `c:build-fasl`.

```
;; generates hello.o
(compile-file "hello.lisp" :system-p t)
;; generates goodbye.o
(compile-file "goodbye.lisp" :system-p t)

;; generates hello-goodbye.fas
(c:build-fasl "hello-goodbye"
              :lisp-files '("hello.o" "goodbye.o"))

;; fasls may be built from mix of objects and libraries (both shared and
;; static)
(c:build-fasl "mixed-bundle"
              :lisp-files '("hello1.o" "hello2.a" "hello3.so"))
```

3.1.1.3 Object file

Object files work as an intermediate file format. If you want to compile more than two lisp files, you might better to compile with a `:system-p t` option, which generates object files (instead of a fasl).

On linux systems, ECL invokes `gcc -c` to generate object files.

An object file consists of some functions in C:

- Functions corresponding to Lisp functions
- The initialization function which registers defined functions on the lisp environment

Consider the example below.

```
(defun say-hello ()
  (print "Hello, world"))
```

During compilation, this simple lisp program is translated into the C program, and then compiled into the object file. The C program contains two functions:

- `static cl_object L1say_hello`: 'say-hello' function
- `ECL_DLLEXPORT void _eclwm2nNauJEfEnD_CLSxi0z(cl_object flag)`: initialization function

In order to use these object files from your C program, you have to call initialization functions before using lisp functions (such as `say-hello`). However the name of an init function is seemed to be randomized and not user-friendly. This is because object files are not intended to be used directly.

ECL provides other user-friendly ways to generate compiled lisp programs (as static/shared libraries or executables), and in each approach, object files act as intermediate files.

3.1.1.4 Static library

ECL can compile lisp programs to static libraries, which can be linked with C programs. A static library is created by `c:build-static-library` with some compiled object files.

```
;; generates hello.o
(compile-file "hello.lsp" :system-p t)
;; generates goodbye.o
(compile-file "goodbye.lsp" :system-p t)

;; generates libhello-goodbye.a
(c:build-static-library "hello-goodbye"
                        :lisp-files '("hello.o" "goodbye.o")
                        :init-name "init_hello_goodbye")
```

When you use a static/shared library, you have to call its init function. The name of this function is specified by the `:init-name` option. In this example, it is then `init_hello_goodbye`. The usage of this function is shown below:

```
#include <ecl/ecl.h>
extern void init_hello_goodbye(cl_object cblock);

int
main(int argc, char **argv)
```

```
{
    /* setup the lisp runtime */
    cl_boot(argc, argv);

    /* call the init function via ecl_init_module */
    ecl_init_module(NULL, init_hello_goodbye);

    /* ... */

    /* shutdown the lisp runtime */
    cl_shutdown();

    return 0;
}
```

Because the program itself does not know the type of the init function, a prototype declaration is inserted. After booting up the lisp environment, it invokes `init_hello_goodbye` via `ecl_init_module`. `init_hello_goodbye` takes an argument, and `ecl_init_module` supplies an appropriate one. Now that the initialization is finished, we can use functions and other stuff defined in the library.

DEPRECATED `read_VV` - equivalent to `ecl_init_module`

3.1.1.5 Shared library

Almost the same as with a static library. The user has to use `c:build-shared-library`:

```
;; generates hello.o
(compile-file "hello.lsp" :system-p t)
;; generates goodbye.o
(compile-file "goodbye.lsp" :system-p t)

;; generates libhello-goodbye.so
(c:build-shared-library "hello-goodbye"
                        :lisp-files '("hello.o" "goodbye.o")
                        :init-name "init_hello_goodbye")
```

3.1.1.6 Executable

ECL supports the generation of executable files. To create a standalone executable from a lisp program, compile all lisp files to object files. After that, calling `c:build-program` creates the executable:

```
;; generates hello.o
(compile-file "hello.lsp" :system-p t)
;; generates goodbye.o
(compile-file "goodbye.lsp" :system-p t)

;; generates hello-goodbye
(c:build-program "hello-goodbye"
                 :lisp-files '("hello.o" "goodbye.o"))
```

Like with native FASL, the program may be built also from libraries.

3.1.1.7 Summary

In this section, some file types that can be compiled with ECL were introduced. Each file type has an adequate purpose:

- Object file: intermediate file format for others
- Fasl files: loaded dynamically via the `load` lisp function
- Static library: linked with and used from C programs
- Shared library: loaded dynamically and used from C programs
- Executable: standalone executable

ECL provides a high-level interface `c:build-*` for each native format. In case of *Portable FASL* the bytecodes compiler is needed.

3.1.2 Compiling with ASDF

For larger systems involving more complex file dependencies, or for systems that are portable across different Common Lisp implementations, it may be better to define systems using asdf (`https://common-lisp.net/project/asdf/`).

ECL provides a useful extension for asdf called `asdf:make-build`, which offers an abstraction for building libraries directly from system definitions. Note that this extension is only available in the ASDF that is shipped with ECL; it may not be available from an ASDF installed from the system or from Quicklisp.

To download dependencies you may use Quicklisp (`https://www.quicklisp.org`) to load your system (with dependencies defined). Make sure you can successfully load and run your library in the ECL REPL (or `*slime-repl*`). Don't worry about other libraries loaded in your image – ECL will only build and pack libraries your project depends on (that is, all dependencies you put in your `.asd` file, and their dependencies - nothing more, despite the fact that other libraries may be loaded).

3.1.2.1 Example code to build

An example project is included in the ECL source distribution in the `examples/asdf_with_dependence/` directory.

This project depends on the `alexandria` library and consists of a system definition (`example-with-dep.asd`), package definition (`package.lisp`), and the actual library code (`example.lisp`).

Before following the steps below, you must configure ASDF to find your systems (`https://asdf.common-lisp.dev/asdf/Configuring-ASDF-to-find-your-systems.html`). You can either copy or symlink the example directory in one of the standard ASDF locations, or push the path of the example directory to your `asdf:*central-registry*`, for example:

```
(push "./" asdf:*central-registry*)
```

3.1.2.2 Build it as an single executable

Use this in the REPL to make an executable:

```
(asdf:make-build :example-with-dep
                 :type :program
                 :move-here #P"./"
```

```
                    :epilogue-code '(progn (example:test-function 5)
                                           (si:exit)))
```

Here the `:epilogue-code` is executed after loading our library; we can use arbitrary Lisp forms here. You can also put this code in your Lisp files and directly build them without this `:epilogue-code` option to achieve the same result. Running the program in a console will display the following and exit:

```
Factorial of 5 is: 120
```

3.1.2.3 Build it as shared library and use in C

Use this in the REPL to make a shared library:

```
(asdf:make-build :example-with-dep
                 :type :shared-library
                 :move-here #P"./"
                 :monolithic t
                 :init-name "init_dll_example")
```

Here `:monolithic t` means that ECL will compile the library and all its dependencies into a single library named `example-with-dep--all-systems.so`. The `:move-here` parameter is self-explanatory. `:init-name` sets the name of the initialization function. Each library linked from C/C++ code must be initialized, and this is a mechanism to specify the initialization function's name.

To use it, we write a simple C program:

```
/* test.c */
#include <ecl/ecl.h>
extern void init_dll_example(cl_object);

int main (int argc, char **argv) {

  cl_boot(argc, argv);
  ecl_init_module(NULL, init_dll_example);

  /* do things with the Lisp library */
  cl_eval(c_string_to_object("(example:test-function 5)"));

  cl_shutdown();
  return 0;
}
```

Compile the file using a standard C compiler (note we're linking to `libecl.so` with `-lecl`, which provides the lisp runtime[1]):

```
gcc test.c example-with-dep--all-systems.so -o test -lecl
```

If ECL is installed in a non-standard location you may need to provide flags for the compiler and the linker. You may read them with:

```
ecl-config --cflags
ecl-config --libs
```

[1] You may also link ECL runtime statically. That is not covered in this walkthrough.

Since our shared object is not in the standard location, you need to provide `LD_LIBRARY_PATH` pointing to the current directory to run the application:

```
LD_LIBRARY_PATH=‘pwd‘ ./test
```

This will show:

```
Factorial of 5 is: 120
```

You can also build all dependent libraries separately as a few `.so` files and link them together. For example, if you are building a library called `complex-example`, that depends on `alexandria` and `cl-fad`, you can do the following (in the REPL):

```
(asdf:make-build :complex-example
                 :type :shared-library
                 :move-here #P"./"
                 :init-name "init_example")

(asdf:make-build :alexandria
                 :type :shared-library
                 :move-here #P"./"
                 :init-name "init_alexandria")

(asdf:make-build :cl-fad
                 :type :shared-library
                 :move-here #P"./"
                 :init-name "init_fad")

(asdf:make-build :bordeaux-threads
                 :type :shared-library
                 :move-here #P"./"
                 :init-name "init_bt")
```

Note that we haven't specified `:monolithic t`, so we need to build `bordeaux-threads` as well because `cl-fad` depends on it. The building sequence doesn't matter and the resultant `.so` files can also be used in your future programs if these libraries are not modified.

We need to initialize all these modules using `ecl_init_module` in the correct order. (`bordeaux-threads` must be initialized before `cl-fad`; `cl-fad` and `alexandria` must be initialized before `complex-ecample`.)

Here is a code snippet (not a full program):

```
extern void init_fad(cl_object);
extern void init_alexandria(cl_object);
extern void init_bt(cl_object);
extern void init_example(cl_object);

/* call these *after* cl_boot(argc, argv);
   if B depends on A, you should first init A then B. */
ecl_init_module(NULL, init_bt);
ecl_init_module(NULL, init_fad);
ecl_init_module(NULL, init_alexandria);
ecl_init_module(NULL, init_example);
```

3.1.2.4 Build it as static library and use in C

To build a static library, use:

```
(asdf:make-build :example-with-dep
                 :type :static-library
                 :move-here #P"./"
                 :monolithic t
                 :init-name "init_example")
```

This will generate `example-with-dep--all-systems.a` in the current directory which we need to initialize with the `init_example` function. Compile it using:

```
gcc test.c example-with-dep--all-systems.a -o test-static -lecl
```

Then run it:

```
./test-static
```

This will show:

```
Factorial of 5 is: 120
```

Note we don't need to pass the current path in `LD_LIBRARY_PATH` here, since our Lisp library is statically bundled with the executable. The result is the same as the shared library example above. You can also build all dependent libraries separately as static libraries.

3.1.3 Cross compilation

ECL supports cross compiling Lisp files for a target system that differs from the host system on which the compilation takes place. This section of the manual describes how to use this feature and explains important things to keep in mind when cross compiling Common Lisp.

3.1.3.1 Getting started with cross compilation

To get started, follow the steps described below:

1. Cross compile ECL itself following the steps described in Section 1.1 [Building ECL], page 11.
2. Step one will create a Lisp file called `target-info.lsp` containing all the information necessary for the ECL compiler to create compiled files for the chosen target system. Load this information into a running ECL process on the host system by calling `c:read-target-info` on this file.
3. Supply the resulting target information to the `:target` option of `compile-file` or `with-compilation-unit`.

For an example, consider two files `a.lisp` and `b.lisp` which are supposed to be linked into a shared library. This can be accomplished with the following steps:

```
(defvar *target* (c:read-target-info "/path/to/ecl/installation/lib/ecl-xx.x.x/target-

(compile-file "a.lisp" :target *target* :system-p t)
(load "a.lisp") ; make macro definitions in "a.lisp" accessible to "b.lisp"
(compile-file "b.lisp" :target *target* :system-p t)

(with-compilation-unit (:target *target*)
  (c:build-shared-library "example"
```

```
                                :lisp-files '("a.o" "b.o")
                                :init-name "init_example"))
```

Cross compilation using ASDF is not supported yet but is planned for the future.

3.1.3.2 Limitations and pitfalls for cross compiling Common Lisp

The ubiquity with which typical Common Lisp programs run code at compile time in macros or other similar constructs makes cross compilation somewhat more challenging than in other programming languages. Since compilation happens on a different system than the one in which the code generated by a macro is run, differences between the host and target environment can lead to bugs if the macro hasn't been written with cross compilation in mind. Consider for instance the following example macro iterating over prime number in the range from `a` to `b`:

```
(defmacro do-prime-numbers ((a b) p &body body)
  `(loop with ,p = (next-prime ,a)
         until (> ,p ,b)
         do ,(if (typep b 'fixnum)
                 `(locally (declare (fixnum ,p))
                    ,@body)
                 `(progn ,@body))
            (setf ,p (next-prime (1+ ,p)))))
```

While this macro works fine in a standard setting, it will produce an incorrect declaration when cross compiling if the size of a fixnum is larger in the host than in the target and the upper bound `b` is larger than `most-positive-fixnum` in the target but smaller than `most-positive-fixnum` in the host. In general, any observable difference between target and host system may contribute to such issues. In the list below, we collect a number of such potential issues and explain strategies to solve them.

- Different type or subtype definitions. This may arise from differing word sizes, as in the fixnum example above, or different feature support such as complex floating point number support in the C compiler leading to differing subtype relationships for complex numbers in Lisp. There are two main strategies for dealing with this problem.
  1. Avoid the problem by rewriting the code not to rely on target specific information. For instance, in the above example we can change the declaration as follows to avoid relying on fixnums:

     ```
     (defmacro do-prime-numbers ((a b) p &body body)
       `(loop with ,p = (next-prime ,a)
              until (> ,p ,b)
              do (locally (declare (type (integer ,a ,b) ,p))
                   ,@body)
                 (setf ,p (next-prime (1+ ,p)))))
     ```

     If the declared integer type is a subtype of fixnum, the ECL compiler will automatically take this into account and optimize the same as if a fixnum declaration had been made.
  2. Use the lexical environment parameter to typep and subtypep. Both functions take an optional parameter which can be used to supply information about type relationships in the target system. In the above example, this works as follows:

```
(defmacro do-prime-numbers ((a b) p &body body &environment env)
  `(loop with ,p = (next-prime ,a)
         until (> ,p ,b)
         do ,(if (typep b 'fixnum env)
                 `(locally (declare (fixnum ,p))
                    ,@body)
                 `(progn ,@body))
            (setf ,p (next-prime (1+ ,p)))))
```

It is important, however, not to blindly modify every occurrence of `typep` and `subtypep` in this way. Some calls to these functions may check for types in the host environment, in which case they should not receive an environment parameter. It is necessary to know for which environment the call is meant in order to decide what to do.

- Different value of `*features*`, leading to mismatched `#+` and `#-` read time conditionals. During cross compilation, `*features*` will be rebound to the value in the target system.[2] However, macros that are defined in the host system will see the host `*features*` during read time. Consider again the example given above where `a.lisp` was cross compiled, then loaded before `b.lisp` was cross compiled. If `a.lisp` contains a macro definition

  ```
  (defmacro my-macro (...)
    #+android `(do-something ...)
    #-android `(do-something-else ...))
  ```

  then `my-macro` will expand to `do-something` in `a.lisp` but `do-something-else` in `b.lisp` if cross compiling for the android target. There are several ways to deal with this issue:

  1. Load source files containing macro definitions in an environment where `*features*` has the same value as in the target system. This can be accomplished either by wrapping `load` in a `with-compilation-unit` call with a `:target` option given or by using the `:load` keyword option of `compile-file` in conjunction with `:target`. In the above example, we would compile as follows:

     ```
     (compile-file "a.lisp" :target *target* :system-p t :load t)
     (compile-file "b.lisp" :target *target* :system-p t)
     ```

     Note that this will not work if the file contains read time conditionals selecting between different compile time code paths due to differences in the host system (let's say for example some code generation from a file that is located in a different location for different compilation hosts).

  2. Replace read time conditionals by lookups that run during macro expansion, e.g.

     ```
     (defmacro my-macro (...)
       (if (member :android *features*)
           `(do-something ...)
           `(do-something-else ...)))
     ```

[2] Changes to `*features*` during compilation are carried over in the target information structure. This means that in a scenario where one cross compiles a file `a.lisp` containing `(eval-when (:compile-toplevel) (push :my-feature *features*))` before cross compiling another file `b.lisp` containing `#+my-feature`, the read time conditional in `b.lisp` will evaluate to true.

3. Add feature keywords for the target system to `*features*` in the host before compilation. This is best suited for conditionals which only add but not subtract code (keywords which only appear in `#+` conditionals). In our example, we would call

```
(push :android *features*)
```

   before starting cross compilation.

- Floating point accuracy issues. ECL relies on the C standard library for numerical functions which may return slightly different values for different systems. If your code depends sensitively on numerical functions returning bitwise identical results, it is best not to run any numerical code at compile time and restrict yourself to only running run time code.

Another option to avoid the aformentioned issues entirely is to use emulation instead of cross compilation. Even if full-blown emulation is too complicated, simply cross compiling from a version of ECL compiled for the same word size as the target system would for instance avoid the fixnum issues in our example macro alltogether.

Whether the compiler is switched to cross compilation mode or not can be diagnosed from the presence of a `:cross` keyword in `*features*`. If necessary, this can be used to select different code paths as well.

3.1.4 C compiler configuration

ECL provides some global variables to customize which C compiler and compiler options to use:

3.1.4.1 Compiler flags

It is not required to surround the compiler flags with quotes or use slashes before special characters.

`string c:*user-cc-flags*` [Variable]
Flags and options to be passed to the C compiler when building FASL, shared libraries and standalone programs.

`string c:*user-linker-flags*` [Variable]
Flags for options (e.g. `-Wl,foo` flags, usually in the `$LDFLAGS` variable in autoconf) to be passed to the linker when building FASL, shared libraries and standalone programs.

`string c:*user-linker-libs*` [Variable]
Flags for libraries (e.g. `-lfoo` flags, usually in the `$LIBS` variable in autoconf) to be passed to the linker when building FASL, shared libraries and standalone programs.

`string c:*cc-optimize*` [Variable]
Optimize options to be passed to the C compiler.

`string c:*user-ld-flags*` [Variable]
DEPRECATED Flags and options to be passed to the linker when building FASL, shared libraries and standalone programs.

3.1.4.2 Compiler & Linker programs

`string c::*cc*` [Variable]
This variable controls how the C compiler is invoked by ECL. One can set the variable appropriately adding for instance flags which the C compiler may need to exploit special hardware features (e.g. a floating point coprocessor).

`string c::*ld*` [Variable]
This variable controls the linker which is used by ECL.

`string c::*ranlib*` [Variable]
Name of the 'ranlib' program on the hosting platform.

`string c::*ar*` [Variable]
Name of the 'ar' program on the hosting platform.

`string c::*ecl-include-directory*` [Variable]
Directory where the ECL header files for the target platform are located.

`string c::*ecl-library-directory*` [Variable]
Directory where the ECL library files for the target platform are located.

3.2 Operating System Interface

3.2.1 Command line arguments

`string ext:*help-message*` [Variable]
Command line help message. Initial value is ECL help message. This variable contains the help message which is output when ECL is invoked with the `--help`.

`list-of-pathname-designators ext:*lisp-init-file-list*` [Variable]
ECL initialization files. Initial value is `'("~/.ecl" "~/.eclrc")`. This variable contains the names of initialization files that are loaded by ECL or embedding programs. The loading of initialization files happens automatically in ECL unless invoked with the option `--norc`. Whether initialization files are loaded or not is controlled by the command line options rules, as described in `ext:process-command-args`.

`list-of-lists ext:+default-command-arg-rules+` [Variable]
ECL command line options. This constant contains a list of rules for parsing the command line arguments. This list is made of all the options which ECL accepts by default. It can be passed as first argument to `ext:process-command-args`, and you can use it as a starting point to extend ECL.

`ext:command-args` [Function]
Original list of command line arguments. This function returns the list of command line arguments passed to either ECL or the program it was embedded in. The output is a list of strings and it corresponds to the argv vector in a C program. Typically, the first argument is the name of the program as it was invoked. You should not count on the filename to be resolved.

`ext:process-command-args` **&key** *args rules* [Function]

args A list of strings. Defaults to the output of `ext:command-args`.

rules A list of lists. Defaults to the value of `ext:+default-command-arg-rules+`.

This function processes the command line arguments passed to either ECL or the program that embeds it. It uses the list of rules rules, which has the following syntax:

`(option-name nargs template [:stop | :noloadrc | :loadrc]*)`

`option-name`
A string with the option prefix as typed by the user. For instance `--help`, `-?`, `--compile`, etc.

`nargs` A non-negative integer denoting the number of arguments taken by this option.

`template` A lisp form, not evaluated, where numbers from 0 to nargs will be replaced by the corresponding option argument.

`:stop` If present, parsing of arguments stops after this option is found and processed. The list of remaining arguments is passed to the rule. ECL's top-level uses this option with the `--` command line option to set `ext:*unprocessed-ecl-command-args*` to the list of remaining arguments.

`:noloadrc, :loadrc`
Determine whether the lisp initialization files in `ext:*lisp-init-file-list*` will be loaded before processing all forms.

`ext:process-command-args` works as follows. First of all, it parses all the command line arguments, except for the first one, which is assumed to contain the program name. Each of these arguments is matched against the rules, sequentially, until one of the patterns succeeds.

A special name `*default*`, matches any unknown command line option. If there is no `*default*` rule and no match is found, an error is signaled. For each rule that succeeds, the function constructs a lisp statement using the template.

After all arguments have been processed, `ext:process-command-args`, and there were no occurrences of `:noloadrc`, the first existing file listed in `ext:*lisp-init-file-list*` will be loaded. Finally, the list of lisp statements will be evaluated.

The following piece of code implements the ls command using lisp. Instructions for building this program are found under `examples/cmdline/ls.lsp`.

```
(setq ext:*help-message* "
ls [--help | -?] filename*
     Lists the file that match the given patterns.
")

(defun print-directory (pathnames)
  (format t "~{~A~%~}"
```

```
            (mapcar #'(lambda (x) (enough-namestring x (si::getcwd)))
                    (mapcan #'directory (or pathnames '("*.*" "*/"))))))

(defconstant +ls-rules+
  '(("--help" 0 (progn (princ ext:*help-message* *standard-output*) (ext:quit 0)))
    ("-?" 0 (progn (princ ext:*help-message* *standard-output*) (ext:quit 0)))
    ("*DEFAULT*" 1 (print-directory 1) :stop)))

(let ((ext:*lisp-init-file-list* NIL)) ; No initialization files
  (handler-case (ext:process-command-args :rules +ls-rules+)
                (error (c)
                       (princ ext:*help-message* *error-output*)
                       (ext:quit 1))))
(ext:quit 0)
```

3.2.2 External processes

ECL provides several facilities for invoking and communicating with external processes. If one just wishes to execute some program, without caring for its output, then probably `ext:system` is the best function. In all other cases it is preferable to use `ext:run-program`, which opens pipes to communicate with the program and manipulate it while it runs on the background.

External process is a structure created with `ext:run-program` (returned as third value). It is programmer responsibility, to call `ext:external-process-wait` on finished processes, however during garbage collection object will be finalized.

`ext:external-process-pid` *process* [Function]
Returns process PID or `nil` if already finished.

`ext:external-process-status` *process* [Function]
Updates process status. `ext:external-process-status` calls `ext:external-process-wait` if process has not finished yet (non-blocking call). Returns two values:

`status` - member of `(:abort :error :exited :signaled :stopped :resumed :running)`

`code` - if process exited it is a returned value, if terminated it is a signal code. Otherwise NIL.

`ext:external-process-wait` *process wait* [Function]
If the second argument is non-NIL, function blocks until external process is finished. Otherwise status is updated. Returns two values (see `ext:external-process-status`).

`ext:terminate-process` *process* **&optional** *force* [Function]
Terminates external process. May signal an error if the process has already finished.

`ext:external-process-input` *process* [Function]
`ext:external-process-output` *process* [Function]
`ext:external-process-error-stream` *process* [Function]
Process stream accessors (read-only).

`ext:run-program` *command argv* **&key** *input output error wait environ if-input-does-not-exist if-output-exists if-error-exists external-format #+windows escape-arguments* [Function]

`ext:run-program` creates a new process specified by the *command* argument. *argv* are the standard arguments that can be passed to a program. For no arguments, use `nil` (which means that just the name of the program is passed as arg 0).

`ext:run-program` will return three values - two-way stream for communication, return code or `nil` (if process is called asynchronously), and `ext:external-process` object holding process state.

It is programmer responsibility to call `ext:external-process-wait` on finished process, however ECL associates Section 3.6.5 [Finalization], page 149, with the object calling it when the object is garbage collected. If process didn't finish but is not referenced, finalizer will be invoked once more during next garbage collection.

The `&key` arguments have the following meanings:

input Either `t`, `nil`, a pathname, a string, a stream or `:stream`. If `t` the standard input for the current process is inherited. If `nil`, `/dev/null` is used. If a pathname (or a string), the file so specified is used. If a stream, all the input is read from that stream and sent to the subprocess. If `:stream`, the `ext:external-process-input` slot is filled in with a stream that sends its output to the process. Defaults to `:stream`.

if-input-does-not-exist
Can be one of: `:error` to generate an error `:create` to create an empty file `nil` (the default) to return nil from `ext:run-program`

output Either `t`, `nil`, a pathname, a string, a stream, or `:stream`. If `t`, the standard output for the current process is inherited. If `nil`, `/dev/null` is used. If a pathname (or as string), the file so specified is used. If a stream, all the output from the process is written to this stream. If `:stream`, the `ext:external-process-output` slot is filled in with a stream that can be read to get the output. Defaults to `:stream`.

if-output-exists
Can be one of: `:error` (the default) to generate an error, `:supersede` to supersede the file with output from the program, `:append` to append output from the program to the file or `nil` to return `nil` from `ext:run-program`.

error Same as `:output`, except that `:error` can also be specified as `:output` in which case all error output is routed to the same place as normal output. Defaults to `:output`.

if-error-exists
Same as `:if-output-exists`.

wait If non-`nil` (default), wait until the created process finishes. If `nil`, continue running Lisp until the program finishes.

environ A list of STRINGs describing the new Unix environment (as in `"man environ"`). The default is to copy the environment of the current process.

To extend existing environment (instead of replacing it), use `:environ (append *my-env* (ext:environ))`.

If non-`nil` `environ` argument is supplied, then first argument to `ext:run-program`, *command*, must be full path to the file.

external-format
: The external-format to use for `:input`, `:output`, and `:error` STREAMs.

Windows specific options:

escape-arguments
: Controls escaping of the arguments passed to `CreateProcess`.

3.2.3 FIFO files (named pipes)

Named pipe (known as fifo) may be created on UNIX with a shell command mkfifo. They can be opened in non-blocking mode by using `:nonblock t` option for `open`. `ext:file-kind` will return for such file `:fifo`. Since it is impossible to guess how many characters are available in this special file `file-length` function will return `nil`.

3.2.4 Operating System Interface Reference

`ext:system` *command* [Function]
: Run shell command ignoring its output. Uses fork.

`ext:make-pipe` [Function]
: Creates a pipe and wraps it in a two way stream.

`ext:quit` **&optional** *exit-code kill-all-threads* [Function]
: This function abruptly stops the execution of the program in which ECL is embedded. Depending on the platform, several other functions will be invoked to free resources, close loaded modules, etc.

The exit code is the code seen by the parent process that invoked this program. Normally a code other than zero denotes an error.

If *kill-all-threads* is non-`nil`, tries to gently kill and join with running threads.

`ext:environ` [Function]
`ext:getenv` *variable* [Function]
`ext:setenv` *variable value* [Function]
: Environment accessors.

`ext:getpid` [Function]
`ext:getuid` [Function]
`ext:getcwd` **&optional** (*change-default-pathname-defaults* **NIL**) [Function]
`ext:chdir` *directory* **&optional** (*change-default-pathname-defaults* **T**) [Function]
`ext:file-kind` *filename follow-symlinks-p* [Function]
`ext:copy-file` *filename destination-filename* [Function]
`ext:chmod` *filename mode* [Function]
: Common operating system functions.

3.3 Foreign Function Interface

3.3.1 What is a FFI?

A Foreign Function Interface, or FFI for short, is a means for a programming language to interface with libraries written in other programming languages, the foreign code. You will see this concept most often being used in interpreted environments, such as Python, Ruby or Lisp, where one wants to reuse the big number of libraries written in C and C++ for dealing with graphical interfaces, networking, filesystems, etc.

A FFI is made of at least three components:

Foreign objects management
: This is the data that the foreign code will use. A FFI needs to provide means to build and manipulate foreign data, with automatic conversions to and from lisp data types whenever possible, and it also has to deal with issues like garbage collection and finalization.

Foreign code loader
: To actually use a foreign routine, the code must reside in memory. The process of loading this code and finding out the addresses of the routines we want to use is normally done by an independent component.

Foreign function invocation
: This is the part of the FFI that deals with actually calling the foreign routines we want to use. For that one typically has to tell the FFI what are the arguments that these routines expect, what are the calling conventions and where are these routines to be found.

On top of these components sits a higher level interface written entirely in lisp, with which you will actually declare and use foreign variables, functions and libraries. In the following sections we describe both the details of the low-level components (See Section 3.3.2 [Two kinds of FFI], page 106, and Section 3.3.3 [Foreign objects], page 107), and of the higher level interface (See Section 3.3.4 [Higher level interfaces], page 109). It is highly recommended that you read all sections.

3.3.2 Two kinds of FFI

ECL allows for two different approaches when building a FFI. Both approaches have a different implementation philosophy and affect the places where you can use the FFI and how.

Static FFI (SFFI)
: For every foreign function and variable you might need to use, a wrapper is automatically written in C with the help of ffi:c-inline. These wrappers are compiled using an ordinary C compiler and linked against both the foreign libraries you want to use and against the ECL library. The result is a FASL file that can be loaded from ECL and where the wrappers appear as ordinary lisp functions and variables that the user may directly invoked.

Dynamic FFI (DFFI)
: First of all, the foreign libraries are loaded in memory using the facilities of the operating system. Similar routines are used to find out and register the

memory location of all the functions and variables we want to use. Finally, when actually accessing these functions, a little piece of assembly code does the job of translating the lisp data into foreign objects, storing the arguments in the stack and in CPU registers, calling the function and converting back the output of the function to lisp.

ECL for this purpose utilizes *libffi (*`https://sourceware.org/libffi/`*)*, a portable foreign-function interface library.

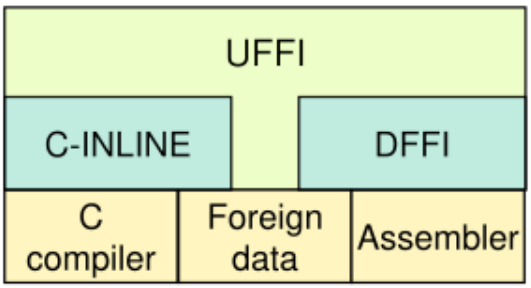


Figure 3.2: FFI components

As you see, the first approach uses rather portable techniques based on a programming language (C, C++) which is strongly supported by the operating system. The conversion of data is performed by a calling routines in the ECL library and we need not care about the precise details (organizing the stack, CPU registers, etc) when calling a function: the compiler does this for us.

On the other hand, the dynamic approach allows us to choose the libraries we load at any time, look for the functions and invoke them even from the toplevel, but it relies on unportable techniques and requires the developers to know very well both the assembly code of the machine the code runs on and the calling conventions of that particular operating system. For these reasons ECL doesn't maintain it's own implementation of the DFFI but rather relies on the third party library.

ECL currently supports the static method on all platforms, and the dynamical one a wide range of the most popular ones, shown in the section *Supported Platforms* at `https://sourceware.org/libffi/`.

You can test if your copy of ECL was built with DFFI by inspecting whether the symbol `:dffi` is present in the list from variable `*features*`.

3.3.3 Foreign objects

While the foreign function invocation protocols differ strongly between platforms and implementations, foreign objects are pretty easy to handle portably. For ECL, a foreign object is just a bunch of bytes stored in memory. The lisp object for a foreign object encapsulates several bits of information:

- A list or a symbol specifying the C type of the object.
- The pointer to the region of memory where data is stored.
- A flag determining whether ECL can automatically manage that piece of memory and deallocated when no longer in use.

A foreign object may contain many different kinds of data: integers, floating point numbers, C structures, unions, etc. The actual type of the object is stored in a list or a symbol which is understood by the higher level interface (See Section 3.3.4 [Higher level interfaces], page 109).

The most important component of the object is the memory region where data is stored. By default ECL assumes that the user will perform manual management of this memory, deleting the object when it is no longer needed. The first reason is that this block may have been allocated by a foreign routine using `malloc()`, or `mmap()`, or statically, by referring to a C constant. The second reason is that foreign functions may store references to this memory which ECL is not aware of and, in order to keep these references valid, ECL should not attempt to automatically destroy the object.

In many cases, however, it is desirable to automatically destroy foreign objects once they have been used. The higher level interfaces UFFI and CFFI (`https://common-lisp.net/project/cffi/`) provide tools for doing this. For instance, in the following example adapted from the UFFI documentation, the string *name* is automatically deallocated

```
(ffi:def-function ("gethostname" c-gethostname)
  ((name (* :unsigned-char))
   (len :int))
  :returning :int)

(ffi:with-foreign-object (name '(:array :unsigned-char 256))
    (if (zerop (c-gethostname (ffi:char-array-to-pointer name) 256))
        (format t "Hostname: ~S" (ffi:convert-from-foreign-string name))
        (error "gethostname() failed.")))
```

3.3.3.1 C Reference

`cl_object ecl_make_foreign_data` (*cl_object tag, cl_index size, void *data*) [Function]

Description This function creates a Lisp "foreign object" that points to a C data. Use this function to pass a data from C to Lisp.

tag denotes the data type (See Section 3.3.7.1 [Primitive Types], page 115) *size* is a number of elements in *data* (or 0 for a raw pointer) *data* is a pointer to C data (either an object or an array)

The C macro `ecl_make_pointer(pointer)` expands to `ecl_make_foreign_data(ECL_NIL, 0, (pointer))`.

`void *ecl_foreign_data_pointer_safe(cl_object` *f*`)` [Function]

Description This function returns a C pointer for the given Lisp foreign object. Lisp foreign objects are constructed with functions `ecl_make_foreign_data` and `ecl_allocate_foreign_data`.

The wrapper created with `ecl_make_foreign_data` is a subject of garbage collection, but the pointer itself is not "freed", because the data producer is "C world" and the consumer is "Lisp world".

This is different from `si:allocate-foreign-object` where the data producer is "Lisp world" and the consumer is "C world". In that case the wrapper is not collected unless

explicitly released with `si:free-foreign-object` in which case the allocated foreign data is also released.

`char *ecl_base_string_pointer_safe(cl_object` *f*`)` [Function]
Description This function returns a pointer to a simple base string *f*. If *f* is not a simple base string this function signals an error.

`cl_object ecl_null_terminated_base_string(cl_object` *s*`)` [Function]
Description Tries to coerce a string to a simple base string suitable for `ecl_base_string_pointer_safe`. This function may cons data.

3.3.4 Higher level interfaces

Up to now we have only discussed vague ideas about how a FFI works, but you are probably more interested on how to actually code all these things in lisp. You have here three possibilities:

- ECL supplies a high level interface which is compatible with UFFI up to version 1.8 (api for >=v2.0 is provided by cffi-uffi-compat system shipped with CFFI). Code designed for UFFI library should run mostly unchanged with ECL. Note, that api resides in ffi package, not uffi, to prevent conflicts with cffi-uffi-compat. New code shouldn't use this interface preferring CFFI (`https://common-lisp.net/project/cffi/`).
- The CFFI (`https://common-lisp.net/project/cffi/`) library features a complete backend for ECL. This method of interfacing with the foreign libraries is preferred over using UFFI.
- ECL's own low level interface. Only to be used if ECL is your deployment platform. It features some powerful constructs that allow you to mix arbitrary C and lisp code.

In the following two subsections we will discuss two practical examples of using the native UFFI and the CFFI library.

UFFI example

The example below shows how to use UFFI in an application. There are several important ingredients:

- You need to specify the libraries you use and do it at the toplevel, so that the compiler may include them at link time.
- Every function you will use has to be declared using `ffi:def-function`.
- In the cases of headers not used by ECL, a header to include might need to be specified using `ffi:clines`.

```
#|
Build and load this module with (compile-file "uffi.lsp" :load t)
|#
;;
;; This toplevel statement notifies the compiler that we will
;; need this shared library at runtime. We do not need this
;; statement in windows or modern macOS.
;; The actually needed path to libm might be different on different systems.
;;
```

```
#-(or ming32 windows darwin)
(ffi:load-foreign-library "/usr/lib/libm.so")
;;
;; With this other statement, we import the C function sin(),
;; which operates on IEEE doubles.
;;
(ffi:def-function ("sin" c-sin) ((arg :double))
                  :returning :double)
;;
;; We now use this function and compare with the lisp version.
;;
(format t "~%Lisp sin:~t~d~%C sin:~t~d~%Difference:~t~d"
(sin 1.0d0) (c-sin 1.0d0) (- (sin 1.0d0) (c-sin 1.0d0)))
```

CFFI example

The CFFI (`https://common-lisp.net/project/cffi/`) library is an independent project and it is not shipped with ECL. If you wish to use it you can go to their homepage, download the code and build it using ASDF.

CFFI differs slightly from UFFI in that functions may be used even without being declared beforehand.

```
#|
Build and load this module with (compile-file "cffi.lsp" :load t)
|#
;;
;; This toplevel statement notifies the compiler that we will
;; need this shared library at runtime. We do not need this
;; statement in windows or macOS.
;;
#-(or ming32 windows darwin)
(cffi:load-foreign-library "/usr/lib/libm.so")
;;
;; With this other statement, we import the C function sin(),
;; which operates on IEEE doubles.
;;
(cffi:defcfun ("sin" c-sin) :double '(:double))
;;
;; We now use this function and compare with the lisp version.
;;
(format t "~%Lisp sin:~t~d~%C sin:~t~d~%Difference:~t~d"
(sin 1.0d0) (c-sin 1.0d0) (- (sin 1.0d0) (c-sin 1.0d0)))
;;
;; The following also works: no declaration!
;;
(let ((c-cos (cffi:foreign-funcall "cos" :double 1.0d0 :double)))
   (format t "~%Lisp cos:~t~d~%C cos:~t~d~%Difference:~t~d"
(cos 1.0d0) c-cos (- (cos 1.0d0) c-cos)))
```

SFFI example (low level inlining)

To compare with the previous pieces of code, we show how the previous programs would be written using `ffi:clines` and `ffi:c-inline`.

```
#|
Build and load this module with (compile-file "ecl.lsp" :load t)
|#
;;
;; With this other statement, we import the C function sin(), which
;; operates on IEEE doubles. Notice that we include the C header to
;; get the full declaration.
;;
(defun c-sin (x)
  (ffi:clines "#include <math.h>")
  (ffi:c-inline (x) (:double) :double "sin(#0)" :one-liner t))
;;
;; We now use this function and compare with the lisp version.
;;
(format t "~%Lisp sin:~t~d~%C sin:~t~d~%Difference:~t~d"
(sin 1.0d0) (c-sin 1.0d0) (- (sin 1.0d0) (c-sin 1.0d0)))
```

3.3.5 SFFI Reference

Reference

`ffi:clines` *c/c++-code** [Special Form]

Insert C declarations and definitions

c/c++-code
: One or more strings with C definitions. Not evaluated.

returns No value.

Description This special form inserts C code from strings passed in the *arguments* directly in the file that results from compiling lisp sources. Contrary to `ffi:c-inline`, this function may have no executable statements, accepts no input value and returns no value.

The main use of `ffi:clines` is to declare or define C variables and functions that are going to be used later in other FFI statements. All statements from *arguments* are grouped at the beginning of the produced header file.

`ffi:clines` is a special form that can only be used in lisp compiled files as a toplevel form. Other uses will lead to an error being signaled, either at the compilation time or when loading the file.

Examples In this example the `ffi:clines` statement is required to get access to the C function `cos`:

```
(ffi:clines "#include <math.h>")
(defun cos (x)
  (ffi:c-inline (x) (:double) :double "cos(#0)" :one-liner t))
```

`ffi:c-inline` (*lisp-values*) (*arg-c-types*) *return-type* *c/c++-code* [Special Form]
&key (*side-effects* **t**) (*one-liner* **nil**)

Inline C code in a lisp form

lisp-values One or more lisp expressions. Evaluated.

arg-c-types
One or more valid FFI types. Evaluated.

return-type
Valid FFI type or `(values ffi-type*)`.

c/c++-code
String containing valid C code plus some valid escape forms.

one-liner Boolean indicating, if the expression is a valid R-value. Defaults to `nil`.

side-effects
Boolean indicating, if the expression causes side effects. Defaults to `t`.

returns One or more lisp values.

Description This is a special form which can be only used in compiled code and whose purpose is to execute some C code getting and returning values from and to the lisp environment.

The first argument *lisp-values* is a list of lisp forms. These forms are going to be evaluated and their lisp values will be transformed to the corresponding C types denoted by the elements in the list *arg-c-types*.

The input values are used to create a valid C expression using the template in *C/C++-code*. This is a string of arbitrary size which mixes C expressions with two kind of escape forms.

The first kind of escape form are made of a hash and a letter or a number, as in: `#0`, `#1`, ..., until `#z`. These codes are replaced by the corresponding input values. The second kind of escape form has the format `@(return [n])`, it can be used as lvalue in a C expression and it is used to set the n-th output value of the `ffi:c-inline` form.

When the parameter *one-liner* is true, then the C template must be a simple C statement that outputs a value. In this case the use of `@(return)` is not allowed. When the parameter *one-liner* is false, then the C template may be a more complicated block form, with braces, conditionals, loops and spanning multiple lines. In this case the output of the form can only be set using `@(return)`.

Parameter *side-effects* set to false will indicate, that the functions causes no side-effects. This information is used by the compiler to optimize the resulting code. If *side-effects* is set to false, but the function may cause the side effects, then results are undefined.

Note that the conversion between lisp arguments and FFI types is automatic. Note also that `ffi:c-inline` cannot be used in interpreted or bytecompiled code! Such usage will signal an error.

Examples The following example implements the transcendental function SIN using the C equivalent:

```
(ffi:c-lines "#include <math.h>")
```

```
(defun mysin (x)
  (ffi:c-inline (x) (:double) :double
    "sin(#0)"
    :one-liner t
    :side-effects nil))
```

This function can also be implemented using the `@(return)` form as follows:

```
(defun mysin (x)
  (ffi:c-inline (x) (:double) :double
    "@(return)=sin(#0);"
    :side-effects nil))
```

The following example is slightly more complicated as it involves loops and two output values:

```
(defun sample (x)
  (ffi:c-inline (x (+ x 2)) (:int :int) (values :int :int) "{
    int n1 = #0, n2 = #1, out1 = 0, out2 = 1;
    while (n1 <= n2) {
      out1 += n1;
      out2 *= n1;
      n1++;
    }
    @(return 0)= out1;
    @(return 1)= out2;
    }"
   :side-effects nil))
```

`ffi:c-progn` *args* **&body** *body* [Special Form]

Interleave C statements with the Lisp code

args Lisp arguments. Evaluated.

returns No value.

Description This form is used for it's side effects. It allows for interleaving C statements with the Lisp code. The argument types doesn't have to be declared – in such case the objects type in the C world will be `cl_object`.

Examples

```
(lambda (i)
  (let* ((limit i)
         (iterator 0)
         (custom-var (cons 1 2)))
    (declare (:int limit iterator))
    (ffi:c-progn (limit iterator custom-var)
                 "cl_object cv = #2;"
                 "ecl_print(cv, ECL_T);"
                 "for (#1 = 0; #1 < #0; #1++) {"
                 (format t "~&Iterator: ~A, I: ~A~%" iterator i)
                 "}")))
```

`ffi:defcallback` *name ret-type arg-desc* **&body** *body* [Special Form]

name Name of the lisp function.

ret-type Declaration of the return type which function returns.

arg-desc List of pairs `(arg-name arg-type)`.

body Function body.

returns Pointer to the defined callback.

Description Defines Lisp function and generates a callback for the C world, which may be passed to these functions. Note, that this special operator has also a dynamic variant (with the same name and interface).

`ffi:defcbody` *name arg-types result-type c-expression* [Macro]

Define C function under the lisp name

name Defined function name.

arg-types Argument types of the defined Lisp function.

result-type
Result type of the C function (may be `(values ...)`.

returns Defined function name.

Description The compiler defines a Lisp function named by *name* whose body consists of the C code of the string *c-expression*. In the *c-expression* one can reference the arguments of the function as `#0`, `#1`, etc.

The interpreter ignores this form.

`ffi:defentry` *name arg-types c-name* **&key** *no-interrupts* [Macro]

name Lisp name for the function.

arg-types Argument types of the C function.

c-name If *c-name* is a list, then C function result type is declared as `(car c-name)` and its name is `(string (cdr c-name))`.

If it's an atom, then the result type is `:object`, and function name is `(string c-name)`.

returns Lisp function `name`.

Description The compiler defines a Lisp function named by *name* whose body consists of a calling sequence to the C language function named by *c-name*.

The interpreter ignores this form.

`ext:with-backend` **&key** *bytecodes c/c++* [Special Form]

Use different code depending on the backend.

Description Depending on whether the bytecodes or C compiler is used, this form will emit the code given in the corresponding keyword argument.

Examples

```
CL-USER> (defmacro test ()
```

```
            '(ext:with-backend :c/c++ "c/c++" :bytecodes "bytecodes"))
TEST
CL-USER> (test)
"bytecodes"
CL-USER> (funcall (compile nil (lambda () (test))))

;;; OPTIMIZE levels: Safety=2, Space=0, Speed=3, Debug=3
"c/c++"
```

`ffi:defla` *name args* **&body** *body* [Macro]

Provide Lisp alternative for interpreted code.

Description Used to DEFine Lisp Alternative. For the interpreter, `ffi:defla` is equivalent to `defun`, but the compiler ignores this form.

3.3.6 DFFI Reference

`ffi:*use-dffi*` [Variable]

This variable controls whether DFFI is used or not.

3.3.7 UFFI Reference

3.3.7.1 Primitive Types

Primitive types have a single value, these include characters, numbers, and pointers. They are all symbols in the keyword package.

‘`:char`’
‘`:unsigned-char`’
: Signed/unsigned 8-bits. Dereferenced pointer returns a character.

‘`:byte`’
‘`:unsigned-byte`’
: Signed/unsigned 8-bits. Dereferenced pointer returns an integer.

‘`:short`’
‘`:unsigned-short`’
‘`:int`’
‘`:unsigned-int`’
‘`:long`’
‘`:unsigned-long`’
: Standard integer types (16-bit, 32-bit and 32/64-bit).

‘`:int16-t`’
‘`:uint16-t`’
‘`:int32-t`’
‘`:uint32-t`’
‘`:int64-t`’
‘`:uint64-t`’
: Integer types with guaranteed bitness.

‘`:float`’
‘`:double`’ Floating point numerals (32-bit and 64-bit).

‘`:long-double`’
Floating point numeral (usually 80-bit, at least 64-bit, exact bitness is compiler/architecture/platform dependent).

‘`:csfloat`’
‘`:cdfloat`’
‘`:clfloat`’
Complex floating point numerals. These types exist only when ECL is built with c99complex support.

‘`:cstring`’
A `NULL` terminated string used for passing and returning characters strings with a C function.

‘`:void`’ The absence of a value. Used to indicate that a function does not return a value.

‘`:pointer-void`’
Points to a generic object.

‘`*`’ Used to declare a pointer to an object.

‘`:object`’ A generic lisp object (i.e. a `cl_object` in C)

Reference

`ffi:def-constant` *name value* **&key** (*export* **nil**) [Macro]
Binds a symbol to a constant.

name A symbol that will be bound to the value.

value An evaluated form that is bound the the name.

export When `t`, the name is exported from the current package. Defaults to `nil`.

returns Constant name.

Description This is a thin wrapper around `defconstant`. It evaluates at compile-time and optionally exports the symbol from the package.

Examples

```
(ffi:def-constant pi2 (* 2 pi))
(ffi:def-constant exported-pi2 (* 2 pi) :export t)
```

Side Effects Creates a new special variable.

`ffi:def-foreign-type` *name definition* [Macro]
Defines a new foreign type

name A symbol naming the new foreign type.

value A form that is not evaluated that defines the new foreign type.

returns Foreign type designator (*value*).

Description Defines a new foreign type

Examples

```
(ffi:def-foreign-type my-generic-pointer :pointer-void)
(ffi:def-foreign-type a-double-float :double-float)
(ffi:def-foreign-type char-ptr (* :char))
```

Side effects Defines a new foreign type.

`ffi:null-char-p` *char* [Function]

Tests a character for NULL value

char A character or integer.

returns A boolean flag indicating if *char* is a NULL value.

Description A predicate testing if a character or integer is NULL. This abstracts the difference in implementations where some return a character and some return a integer whence dereferencing a C character pointer.

Examples

```
(ffi:def-array-pointer ca :unsigned-char)
  (let ((fs (ffi:convert-to-foreign-string "ab")))
    (values (ffi:null-char-p (ffi:deref-array fs 'ca 0))
            (ffi:null-char-p (ffi:deref-array fs 'ca 2))))
;; => NIL T
```

3.3.7.2 Aggregate Types

Overview

Aggregate types are comprised of one or more primitive types.

Reference

`ffi:def-enum` *name fields* **&key** *separator-string* [Macro]

Defines a C enumeration

name A symbol that names the enumeration.

fields A list of field definitions. Each definition can be a symbol or a list of two elements. Symbols get assigned a value of the current counter which starts at 0 and increments by 1 for each subsequent symbol. It the field definition is a list, the first position is the symbol and the second position is the value to assign the the symbol. The current counter gets set to 1+ this value.

returns A string that governs the creation of constants. The default is `"#"`.

Description Declares a C enumeration. It generates constants with integer values for the elements of the enumeration. The symbols for the these constant values are created by the concatenation of the enumeration name, separator-string, and field symbol. Also creates a foreign type with the name name of type `:int`.

Examples

```
(ffi:def-enum abc (:a :b :c))
;; Creates constants abc#a (1), abc#b (2), abc#c (3) and defines
;; the foreign type "abc" to be :int

(ffi:def-enum efoo (:e1 (:e2 10) :e3) :separator-string "-")
;; Creates constants efoo-e1 (1), efoo-e2 (10), efoo-e3 (11) and defines
;; the foreign type efoo to be :int
```

Side effects Creates a `:int` foreign type, defines constants.

`ffi:def-struct` *name* **&rest** *fields* [Macro]

Defines a C structure

name A symbol that names the structure.

fields A variable number of field definitions. Each definition is a list consisting of a symbol naming the field followed by its foreign type.

Description Declares a structure. A special type is available as a slot in the field. It is a pointer that points to an instance of the parent structure. It's type is `:pointer-self`.

Examples

```
(ffi:def-struct foo (a :unsigned-int)
  (b    (* :char))
  (c    (:array :int 10))
  (next :pointer-self))
```

Side effects Creates a foreign type.

`ffi:get-slot-value` *obj type field* [Function]

Retrieves a value from a slot of a structure

obj A pointer to the foreign structure.

type The name of the foreign structure.

field The name of the desired field in the foreign structure.

returns The value of the *field* in the structure *obj*.

Description Accesses a slot value from a structure. This is generalized and can be used with `setf`.

Examples

```
(ffi:get-slot-value foo-ptr 'foo-structure 'field-name)
(setf (ffi:get-slot-value foo-ptr 'foo-structure 'field-name) 10)
```

`ffi:get-slot-pointer` *obj type field* [Function]

Retrieves a pointer from a slot of a structure

obj A pointer to the foreign structure.

type The name of the foreign structure.

field The name of the desired field in the foreign structure.

returns The value of the pointer *field* in the structure *obj*.

Description This is similar to `ffi:get-slot-value`. It is used when the value of a slot is a pointer type.

Examples

```
(ffi:get-slot-pointer foo-ptr 'foo-structure 'my-char-ptr)
```

`ffi:def-array-pointer` *name type* [Macro]

Defines a pointer to an array of *type*

name A name of the new foreign type.

type The foreign type of the array elements.

Description Defines a type that is a pointer to an array of *type*.

Examples

```
(ffi:def-array-pointer byte-array-pointer :unsigned-char)
```

Side effects Defines a new foreign type.

`ffi:deref-array` *array type position* [Function]

Dereference an array

array A foreign array.

type The foreign type of the *array*.

position An integer specifying the position to retrieve from the *array*.

returns The value stored in the *position* of the *array*.

Description Dereferences (retrieves) the value of the foreign array element. `setf`-able.

Examples

```
(ffi:def-array-pointer ca :char)
  (let ((fs (ffi:convert-to-foreign-string "ab")))
    (values (ffi:null-char-p (ffi:deref-array fs 'ca 0))
    (ffi:null-char-p (ffi:deref-array fs 'ca 2))))
;; => NIL T
```

`ffi:def-union` *name* **&rest** *fields* [Macro]

Defines a foreign union type

name A name of the new union type.

fields A list of fields of the union in form `(field-name field-type)`.

Description Defines a foreign union type.

Examples

```
(ffi:def-union test-union
  (a-char :char)
  (an-int :int))

(let ((u (ffi:allocate-foreign-object 'test-union)))
  (setf (ffi:get-slot-value u 'test-union 'an-int) (+ 65 (* 66 256))))
```

```
  (prog1
     (ffi:ensure-char-character (ffi:get-slot-value u 'test-union 'a-char))
   (ffi:free-foreign-object u)))
;; => #\A
```

Side effects Defines a new foreign type.

3.3.7.3 Foreign Objects

Overview

Objects are entities that can allocated, referred to by pointers, and can be freed.

Reference

`ffi:allocate-foreign-object` *type* **&optional** *size* [Function]

Allocates an instance of a foreign object

type The type of foreign object to allocate. This parameter is evaluated.

size An optional size parameter that is evaluated. If specified, allocates and returns an array of *type* that is *size* members long. This parameter is evaluated.

returns A pointer to the foreign object.

Description Allocates an instance of a foreign object. It returns a pointer to the object.

Examples

```
(ffi:def-struct ab (a :int) (b :double))
;; => (:STRUCT (A :INT) (B :DOUBLE))
(ffi:allocate-foreign-object 'ab)
;; => #<foreign AB>
```

`ffi:free-foreign-object` *ptr* [Function]

Frees memory that was allocated for a foreign object

ptr A pointer to the allocated foreign object to free.

Description Frees memory that was allocated for a foreign object.

`ffi:with-foreign-object` (*var type*) **&body** *body* [Macro]

Wraps the allocation, binding and destruction of a foreign object around a body of code

var Variable name to bind.

type Type of foreign object to allocate. This parameter is evaluated.

body Code to be evaluated.

returns The result of evaluating the body.

Description This function wraps the allocation, binding, and destruction of a foreign object around the body of code.

Examples

```
(defun gethostname2 ()
  "Returns the hostname"
  (ffi:with-foreign-object (name '(:array :unsigned-char 256))
    (if (zerop (c-gethostname (ffi:char-array-to-pointer name) 256))
        (ffi:convert-from-foreign-string name)
        (error "gethostname() failed."))))
```

`ffi:size-of-foreign-type` *ftype* [Macro]

Returns the number of data bytes used by a foreign object type

ftype A foreign type specifier. This parameter is evaluated.

returns Number of data bytes used by a foreign object *ftype*.

Description Returns the number of data bytes used by a foreign object type. This does not include any Lisp storage overhead.

Examples

```
(ffi:size-of-foreign-type :unsigned-byte)
;; => 1
(ffi:size-of-foreign-type 'my-100-byte-vector-type)
;; => 100
```

`ffi:pointer-address` *ptr* [Function]

Returns the address of a pointer

ptr A pointer to a foreign object.

returns An integer representing the pointer's address.

Description Returns the address as an integer of a pointer.

`ffi:deref-pointer` *ptr ftype* [Function]

Dereferences a pointer

ptr Pointer to a foreign object.

ftype Foreign type of the object being pointed to.

returns The value of the object where the pointer points.

Description Returns the object to which a pointer points. `setf`-able.

Notes Casting of the pointer may be performed with `ffi:with-cast-pointer` together with `ffi:deref-pointer`/`ffi:deref-array`.

Examples

```
(let ((intp (ffi:allocate-foreign-object :int)))
  (setf (ffi:deref-pointer intp :int) 10)
  (prog1
      (ffi:deref-pointer intp :int)
    (ffi:free-foreign-object intp)))
;; => 10
```

`ffi:ensure-char-character` *object* [Function]

Ensures that a dereferenced `:char` pointer is a character

object Either a character or a integer specifying a character code.

returns A character.

Description Ensures that an objects obtained by dereferencing `:char` and `:unsigned-char` pointers is a lisp character.

Examples

```
(let ((fs (ffi:convert-to-foreign-string "a")))
  (prog1
      (ffi:ensure-char-character (ffi:deref-pointer fs :char))
    (ffi:free-foreign-object fs)))
;; => #\a
```

Exceptional Situations Depending upon the implementation and what UFFI expects, this macro may signal an error if the object is not a character or integer.

`ffi:ensure-char-integer` *object* [Function]

Ensures that a dereferenced `:char` pointer is an integer

object Either a character or a integer specifying a character code.

returns An integer.

Description Ensures that an objects obtained by dereferencing `:char` and `:unsigned-char` pointers is a lisp integer.

Examples

```
(let ((fs (ffi:convert-to-foreign-string "a")))
  (prog1
      (ffi:ensure-char-integer (ffi:deref-pointer fs :char))
    (ffi:free-foreign-object fs)))
;; => 96
```

Exceptional Situations Depending upon the implementation and what UFFI expects, this macro may signal an error if the object is not a character or integer.

`ffi:make-null-pointer` *ftype* [Function]

Create a NULL pointer of a specified type

ftype A type of object to which the pointer refers.

returns The NULL pointer of type *ftype*.

`ffi:null-pointer-p` *ptr* [Function]

Tests a pointer for NULL value

ptr A foreign object pointer.

returns The boolean flag.

`ffi:+null-cstring-pointer+` [Variable]

A NULL cstring pointer. This can be used for testing if a cstring returned by a function is NULL.

`ffi:with-cast-pointer` (*var ptr ftype*) **&body** *body* [Macro]

Wraps a body of code with a pointer cast to a new type

var Symbol which will be bound to the casted object.

ptr Pointer to a foreign object.

ftype A foreign type of the object being pointed to.

returns The value of the object where the pointer points.

Description Executes *body* with *ptr* cast to be a pointer to type *ftype*. *var* will be bound to this value during the execution of *body*.

Examples

```
(ffi:with-foreign-object (size :int)
  ;; FOO is a foreign function returning a :POINTER-VOID
  (let ((memory (foo size)))
    (when (mumble)
      ;; at this point we know for some reason that MEMORY points
      ;; to an array of unsigned bytes
      (ffi:with-cast-pointer (memory :unsigned-byte)
        (dotimes (i (deref-pointer size :int))
          (do-something-with
              (ffi:deref-array memory '(:array :unsigned-byte) i)))))))
```

`ffi:def-foreign-var` *name type module* [Macro]

Defines a symbol macro to access a variable in foreign code

name A string or list specifying the symbol macro's name. If it is a string, that names the foreign variable. A Lisp name is created by translating `#\_` to `#\-` and by converting to upper-case.

If it is a list, the first item is a string specifying the foreign variable name and the second it is a symbol stating the Lisp name.

type A foreign type of the foreign variable.

module Either a string specifying the module (or library) the foreign variable resides in, `:default` if no module needs to be loaded or `nil` to use SFFI.

Description Defines a symbol macro which can be used to access (get and set) the value of a variable in foreign code.

Examples

C code defining foreign structure, standalone integer and the accessor:

```
int baz = 3;

typedef struct {
  int x;
  double y;
} foo_struct;

foo_struct the_struct = { 42, 3.2 };
```

```
int foo () {
  return baz;
}
```

Lisp code defining C structure, function and a variable:

```
(ffi:def-struct foo-struct
  (x :int)
  (y :double))

(ffi:def-function ("foo" foo) ()
  :returning :int
  :module "foo")

(ffi:def-foreign-var ("baz" *baz*) :int "foo")
(ffi:def-foreign-var ("the_struct" *the-struct*) foo-struct "foo")

*baz*           ;; => 3
(incf *baz*)    ;; => 4
(foo)           ;; => 4
```

3.3.7.4 Foreign Strings

Overview

UFFI has functions to two types of C-compatible strings: cstrings and foreign strings. cstrings are used only as parameters to and from functions. In some implementations a cstring is not a foreign type but rather the Lisp string itself. On other platforms a cstring is a newly allocated foreign vector for storing characters. The following is an example of using cstrings to both send and return a value.

```
(ffi:def-function ("getenv" c-getenv)
    ((name :cstring))
  :returning :cstring)

(defun my-getenv (key)
  "Returns an environment variable, or NIL if it does not exist"
  (check-type key string)
  (ffi:with-cstring (key-native key)
    (ffi:convert-from-cstring (c-getenv key-native))))
```

In contrast, foreign strings are always a foreign vector of characters which have memory allocated. Thus, if you need to allocate memory to hold the return value of a string, you must use a foreign string and not a cstring. The following is an example of using a foreign string for a return value.

```
(ffi:def-function ("gethostname" c-gethostname)
    ((name (* :unsigned-char))
     (len :int))
  :returning :int)
```

```
(defun gethostname ()
  "Returns the hostname"
  (let* ((name (ffi:allocate-foreign-string 256))
         (result-code (c-gethostname name 256))
         (hostname (when (zerop result-code)
                     (ffi:convert-from-foreign-string name))))
    ;; UFFI does not yet provide a universal way to free
    ;; memory allocated by C's malloc. At this point, a program
    ;; needs to call C's free function to free such memory.
    (unless (zerop result-code)
      (error "gethostname() failed."))))
```

Foreign functions that return pointers to freshly allocated strings should in general not return cstrings, but foreign strings. (There is no portable way to release such cstrings from Lisp.) The following is an example of handling such a function.

```
(ffi:def-function ("readline" c-readline)
    ((prompt :cstring))
  :returning (* :char))

(defun readline (prompt)
  "Reads a string from console with line-editing."
  (ffi:with-cstring (c-prompt prompt)
    (let* ((c-str (c-readline c-prompt))
           (str (ffi:convert-from-foreign-string c-str)))
      (ffi:free-foreign-object c-str)
      str)))
```

Reference

`ffi:convert-from-cstring` *object* [Macro]

Converts a `cstring` to a Lisp string

object A `cstring`

returns A Lisp string

Description Converts a Lisp string to a cstring. This is most often used when processing the results of a foreign function that returns a cstring.

`ffi:convert-to-cstring` *object* [Macro]

Converts a Lisp string to a `cstring`

object A Lisp string

returns A `cstring`

Description Converts a Lisp string to a `cstring`. The `cstring` should be freed with `ffi:free-cstring`.

Side Effects This function allocates memory.

`ffi:convert-from-cstring` *cstring* [Macro]

Free memory used by *cstring*

cstring `cstring` to be freed.

Description Frees any memory possibly allocated by `ffi:convert-to-cstring`. On ECL, a `cstring` is just the Lisp string itself.

`ffi:with-cstring` (*cstring string*) **&body** *body* [Macro]

Binds a newly created `cstring`

cstring A symbol naming the `cstring` to be created.

string A Lisp string that will be translated to a `cstring`.

body The body of where the *cstring* will be bound.

returns Result of evaluating the *body*.

Description Binds a symbol to a `cstring` created from conversion of a *string*. Automatically frees the *cstring*.

Examples

```
(ffi:def-function ("getenv" c-getenv)
    ((name :cstring))
  :returning :cstring)

(defun getenv (key)
  "Returns an environment variable, or NIL if it does not exist"
  (check-type key string)
  (ffi:with-cstring (key-cstring key)
    (ffi:convert-from-cstring (c-getenv key-cstring))))
```

`ffi:with-cstrings` *bindings* **&body** *body* [Macro]

Binds a newly created `cstrings`

bindings List of pairs *(cstring string)*, where *cstring* is a name for a `cstring` translated from Lisp string *string*.

body The body of where the *bindings* will be bound.

returns Result of evaluating the *body*.

Description Binds a symbols to a `cstrings` created from conversion of a *string*s. Automatically frees the *cstring*s. This macro works similar to `let*`. Based on `with-cstring`.

`ffi:convert-from-foreign-string` *foreign-string* **&key** *length* (*null-terminated-p* **t**) [Macro]

Converts a foreign string into a Lisp string

foreign-string A foreign string.

length The length of the foreign string to convert. The default is the length of the string until a NULL character is reached.

null-terminated-p
: A boolean flag with a default value of `t`. When true, the string is converted until the first NULL character is reached.

returns A Lisp string.

Description Returns a Lisp string from a foreign string. Can translate ASCII and binary strings.

`ffi:convert-to-foreign-string` *string* [Macro]

Converts a Lisp string to a foreign string

string A Lisp string.

returns A foreign string.

Description Converts a Lisp string to a foreign string. Memory should be freed with `ffi:free-foreign-object`.

`ffi:allocate-foreign-string` *size* **&key** *unsigned* [Macro]

Allocates space for a foreign string

size The size of the space to be allocated in bytes.

unsigned A boolean flag with a default value of `t`. When true, marks the pointer as an `:unsigned-char`.

returns A foreign string which has undefined contents.

Description Allocates space for a foreign string. Memory should be freed with `ffi:free-foreign-object`.

`ffi:with-foreign-string` (*foreign-string string*) **&body** *body* [Macro]

Binds a newly allocated `foreign-string`

foreign-string
: A symbol naming the `foreign string` to be created.

string A Lisp string that will be translated to a `foreign string`.

body The body of where the *foreign-string* will be bound.

returns Result of evaluating the *body*.

Description Binds a symbol to a `foreign-string` created from conversion of a *string*. Automatically deallocates the *foreign-string*.

Examples

`ffi:with-foreign-strings` *bindings* **&body** *body* [Macro]

Binds a newly created `foreign string`

bindings List of pairs *(foreign-string string)*, where *foreign-string* is a name for a `foreign string` translated from Lisp string *string*.

body The body of where the *bindings* will be bound.

returns Result of evaluating the *body*.

Description Binds a symbols to a `foreign-string`s created from conversion of a *string*s. Automatically frees the *foreign-string*s. This macro works similar to `let*`. Based on `ffi:with-foreign-string`.

3.3.7.5 Functions and Libraries

Reference

`ffi:def-function` *name args* **&key** *module* (*returning* **:void**) (*call* **:cdecl**) [Macro]

name A string or list specifying the function name. If it is a string, that names the foreign function. A Lisp name is created by translating `#\_` to `#\-` and by converting to upper-case in case-insensitive Lisp implementations. If it is a list, the first item is a string specifying the foreign function name and the second it is a symbol stating the Lisp name.

args A list of argument declarations. If `nil`, indicates that the function does not take any arguments.

module Either a string specifying which module (or library) that the foreign function resides, `:default` if no module needs to be loaded or `nil` to use SFFI.

call Function calling convention. May be one of `:default`, `:cdecl`, `:sysv`, `:stdcall`, `:win64` and `:unix64`.

This argument is used only when we're using the dynamic function interface. If ECL is built without the DFFI support, then it uses SFFI the *call* argument is ignored.

returning A declaration specifying the result type of the foreign function. `:void` indicates that the function does not return any value.

Description Declares a foreign function.

Examples

```
(ffi:def-function "gethostname"
    ((name (* :unsigned-char))
     (len :int))
  :returning :int)
```

`ffi:load-foreign-library` *filename* **&key** *module supporting-libraries force-load system-library* [Macro]

filename A string or pathname specifying the library location in the filesystem.

module **IGNORED** A string designating the name of the module to apply to functions in this library.

supporting-libraries
IGNORED A list of strings naming the libraries required to link the foreign library.

force-load **IGNORED** Forces the loading of the library if it has been previously loaded.

system-library
Denotes if the loaded library is a system library (accessible with the correct linker flags). If `t`, then SFFI is used and the linking is performed after compilation of the module. Otherwise (default) both SFFI and DFFI are used, but SFFI only during the compilation.

returns A generalized boolean *true* if the library was able to be loaded successfully or if the library has been previously loaded, otherwise `nil`.

Description Loads a foreign library. Ensures that a library is only loaded once during a session.

Examples

```
(ffi:load-foreign-library #p"/usr/lib/libmagic.so.1")
;; => #<codeblock "/usr/lib/libmagic.so">
```

Side Effects Loads the foreign code into the Lisp system.

Affected by Ability to load the file.

`ffi:find-foreign-library` *names directories* **&key** *drive-letters types* [Function]

Finds a foreign library file

names A string or list of strings containing the base name of the library file.

directories A string or list of strings containing the directory the library file.

drive-letters
A string or list of strings containing the drive letters for the library file.

types A string or list of strings containing the file type of the library file. Default is `nil`. If `nil`, will use a default type based on the currently running implementation.

returns A path containing the path to the *first* file found, or `nil` if the library file was not found.

Description Finds a foreign library by searching through a number of possible locations. Returns the path of the first found file.

Examples

```
(ffi:find-foreign-library '("libz" "libmagic")
                          '("/usr/local/lib/" "/usr/lib/")
                          :types '("so" "dll"))
;; => #P"/usr/lib/libz.so.1.2.8"
```

3.4 Native threads

3.4.1 Tasks, threads or processes

On most platforms, ECL supports native multithreading. That means there can be several tasks executing lisp code on parallel and sharing memory, variables and files. The interface for multitasking in ECL, like those of most other implementations, is based on a set of functions and types that resemble the multiprocessing capabilities of old Lisp Machines.

This backward compatibility is why tasks or threads are called "processes". However, they should not be confused with operating system processes, which are made of programs running in separate contexts and without access to each other's memory.

The implementation of threads in ECL is purely native and based on Posix Threads wherever available. The use of native threads has advantages. For instance, they allow for non-blocking file operations, so that while one task is reading a file, a different one is performing a computation.

As mentioned above, tasks share the same memory, as well as the set of open files and sockets. This manifests on two features. First of all, different tasks can operate on the same lisp objects, reading and writing their slots, or manipulating the same arrays. Second, while threads share global variables, constants and function definitions they can also have thread-local bindings to special variables that are not seen by other tasks.

The fact that different tasks have access to the same set of data allows both for flexibility and a greater risk. In order to control access to different resources, ECL provides the user with locks, as explained in the next section.

3.4.2 Processes (native threads)

Process is a primitive representing native thread.

3.4.3 Processes dictionary

`cl_object mp_all_processes` () [Function]

`mp:all-processes` [Function]

Returns the list of processes associated to running tasks. The list is a fresh new one and can be destructively modified. However, it may happen that the output list is not up to date, because some of the tasks have expired before this copy is returned.

`cl_object mp_exit_process` () *ecl_attr_noreturn* [Function]

`mp:exit-process` [Function]

When called from a running task, this function immediately causes the task to finish. When invoked from the main thread, it is equivalent to invoking `ext:quit` with exit code 0.

`cl_object mp_interrupt_process` (*cl_object process, cl_object function*) [Function]

`mp:interrupt-process` *process function* [Function]

Interrupt a task. This function sends a signal to a running *process*. When the task is free to process that signal, it will stop whatever it is doing and execute the given function.

WARNING: Use with care! Interrupts can happen anywhere, except in code regions explicitely protected with `mp:without-interrupts`. This can lead to dangerous situations when interrupting functions which are not thread safe. In particular, one has to consider:

- Reentrancy: Functions, which usually are not called recursively can be re-entered during execution of the interrupt.
- Stack unwinding: Non-local jumps like `throw` or `return-from` in the interrupting code will handle `unwind-protect` forms like usual. However, the cleanup forms of an `unwind-protect` can still be interrupted. In that case the execution flow will jump to the next `unwind-protect`.

Note also that no guarantees are made that functions from the Common Lisp standard or ECL extensions are interrupt safe (although most of them will be). In particular, the compiler (`compile` and `compile-file` functions), FFI calls and aquire/release functions for multithreading synchronization objects like mutexes or condition variables should not be interrupted by `mp:interrupt-process`.

Example:

Kill a task that is doing nothing (See `mp:process-kill`).

```
(flet ((task-to-be-killed ()
         ;; Infinite loop
         (loop (sleep 1))))
  (let ((task (mp:process-run-function 'background #'task-to-be-killed)))
    (sleep 10)
    (mp:interrupt-process task 'mp:exit-process)))
```

`cl_object mp_make_process` (*cl_narg narg, ...*) [Function]

`mp:make-process` **&key** *name initial-bindings* [Function]
Create a new thread. This function creates a separate task with a name set to *name* and no function to run. See also `mp:process-run-function`. Returns newly created process.

If *initial-bindings* is false, the new process inherits local bindings to special variables (i.e. binding a special variable with `let` or `let*`) from the thread that enables it, otherwise the new thread initially possesses no local bindings.

`cl_object mp_process_active_p` (*cl_object process*) [Function]

`mp:process-active-p` *process* [Function]
Returns `t` when *process* is active, `nil` otherwise. Signals an error if *process* doesn't designate a valid process.

`cl_object mp_process_enable` (*cl_object process*) [Function]

`mp:process-enable` *process* [Function]
The argument to this function should be a process created by `mp:make-process`, which has a function associated as per `mp:process-preset` but which is not yet running. After invoking this function a new thread will be created in which the associated function will be executed. Returns *process* if the thread creation was successful and `nil` otherwise.

```
(defun process-run-function (process-name process-function &rest args)
  (let ((process (mp:make-process name)))
    (apply #'mp:process-preset process function args)
    (mp:process-enable process)))
```

`cl_object mp_process_yield` () [Function]

`mp:process-yield` [Function]
Yield the processor to other threads.

`cl_object mp_process_join` (*cl_object process*) [Function]

`mp:process-join` *process* [Function]

Suspend current thread until *process* exits. Return the result values of the *process* function.

`cl_object mp_process_kill` (*cl_object process*) [Function]

`mp:process-kill` *process* [Function]

Try to stop a running task. Killing a process may fail if the task has disabled interrupts.

Example:

Kill a task that is doing nothing

```
(flet ((task-to-be-killed ()
         ;; Infinite loop
         (loop (sleep 1))))
  (let ((task (mp:process-run-function 'background #'task-to-be-killed)))
    (sleep 10)
    (mp:process-kill task)))
```

`cl_object mp_process_suspend` (*cl_object process*) [Function]

`mp:process-suspend` *process* [Function]

Suspend a running *process*. May be resumed with `mp:process-resume`.

Example:

```
(flet ((ticking-task ()
         ;; Infinite loop
         (loop
            (sleep 1)
            (print :tick))))
  (print "Running task (one tick per second)")
  (let ((task (mp:process-run-function 'background #'ticking-task)))
    (sleep 5)
    (print "Suspending task for 5 seconds")
    (mp:process-suspend task)
    (sleep 5)
    (print "Resuming task for 5 seconds")
    (mp:process-resume task)
    (sleep 5)
    (print "Killing task")
    (mp:process-kill task)))
```

`cl_object mp_process_resume` (*cl_object process*) [Function]

`mp:process-resume` *process* [Function]

Resumes a suspended *process*. See example in `mp:process-suspend`.

`cl_object mp_process_name` (*cl_object process*) [Function]

`mp:process-name` *process* [Function]

Returns the name of a *process* (if any).

`cl_object mp_process_preset` (*cl_narg narg, cl_object process, cl_object function, ...*) [Function]

`mp:process-preset` *process function* **&rest** *function-args* [Function]
Associates a *function* to call with the arguments *function-args*, with a stopped *process*. The function will be the entry point when the task is enabled in the future.

See `mp:process-enable` and `mp:process-run-function`.

`cl_object mp_process_run_function` (*cl_narg narg, cl_object name, cl_object function, ...*) [Function]

`mp:process-run-function` *name function* **&rest** *function-args* [Function]
Create a new process using `mp:make-process`, associate a function to it and start it using `mp:process-preset`.

Example:

```
(flet ((count-numbers (end-number)
         (dotimes (i end-number)
   (format t "~%;;; Counting: ~i" i)
   (terpri)
   (sleep 1))))
  (mp:process-run-function 'counter #'count-numbers 10))
```

`cl_object mp_current_process ()` [Function]

`mp:*current-process*` [Variable]
Returns/holds the current process of a caller.

`cl_object mp_block_signals ()` [Function]

`mp:block-signals` [Function]
Blocks process for interrupts and returns the previous sigmask.

See `mp:interrupt-process`.

`cl_object mp_restore_signals` (*cl_object sigmask*) [Function]

`mp:restore-signals` *sigmask* [Function]
Enables the interrupts from *sigmask*.

See `mp:interrupt-process`.

`mp:without-interrupts` **&body** *body* [Macro]
Executes *body* with all deferrable interrupts disabled. Deferrable interrupts arriving during execution of the *body* take effect after *body* has been executed.

Deferrable interrupts include most blockable POSIX signals, and `mp:interrupt-process`. Does not interfere with garbage collection, and unlike in many traditional Lisps using userspace threads, in ECL `mp:without-interrupts` does not inhibit scheduling of other threads.

Binds `mp:allow-with-interrupts`, `mp:with-local-interrupts` and `mp:with-restored-interrupts` as a local macros.

`mp:with-restored-interrupts` executes the body with interrupts enabled if and only if the `mp:without-interrupts` was in an environment in which interrupts were allowed.

`mp:allow-with-interrupts` allows the `mp:with-interrupts` to take effect during the dynamic scope of its body, unless there is an outer `mp:without-interrupts` without a corresponding `mp:allow-with-interrupts`.

`mp:with-local-interrupts` executes its body with interrupts enabled provided that there is an `mp:allow-with-interrupts` for every `mp:without-interrupts` surrounding the current one. `mp:with-local-interrupts` is equivalent to:

```
(mp:allow-with-interrupts (mp:with-interrupts ...))
```

Care must be taken not to let either `mp:allow-with-interrupts` or `mp:with-local-interrupts` appear in a function that escapes from inside the `mp:without-interrupts` in:

```
(mp:without-interrupts
  ;; The body of the lambda would be executed with WITH-INTERRUPTS allowed
  ;; regardless of the interrupt policy in effect when it is called.
  (lambda () (mp:allow-with-interrupts ...)))

(mp:without-interrupts
  ;; The body of the lambda would be executed with interrupts enabled
  ;; regardless of the interrupt policy in effect when it is called.
  (lambda () (mp:with-local-interrupts ...)))
```

`mp:with-interrupts` **&body** *body* [Macro]

Executes *body* with deferrable interrupts conditionally enabled. If there are pending interrupts they take effect prior to executing *body*.

As interrupts are normally allowed `mp:with-interrupts` only makes sense if there is an outer `mp:without-interrupts` with a corresponding `mp:allow-with-interrupts`: interrupts are not enabled if any outer `mp:without-interrupts` is not accompanied by `mp:allow-with-interrupts`.

3.4.4 Locks (mutexes)

Locks are used to synchronize access to the shared data. Lock may be owned only by a single thread at any given time. Recursive locks may be re-acquired by the same thread multiple times (and non-recursive locks can't).

3.4.5 Locks dictionary

`cl_object ecl_make_lock` (*cl_object name*, *bool recursive*) [Function]

C/C++ equivalent of `mp:make-lock` without *key* arguments.

See `mp:make-lock`.

`mp:make-lock` **&key** *name* (*recursive* **nil**) [Function]

Creates a lock named *name*. If *recursive* is true, a recursive lock is created that can be locked multiple times by the same thread.

`cl_object mp_recursive_lock_p` (*cl_object lock*) [Function]

`mp:recursive-lock-p` *lock* [Function]
Predicate verifying if *lock* is recursive.

`cl_object mp_holding_lock_p` (*cl_object lock*) [Function]

`mp:holding-lock-p` *lock* [Function]
Predicate verifying if the current thread holds *lock*.

`cl_object mp_lock_name` (*cl_object lock*) [Function]

`mp:lock_name` *lock* [Function]
Returns the name of *lock*.

`cl_object mp_lock_owner` (*cl_object lock*) [Function]

`mp:lock-owner` *lock* [Function]
Returns the process owning *lock* or `nil` if the mutex is not owned by any process. For testing whether the current thread is holding a lock see `mp:holding-lock-p`.

`cl_object mp_lock_count` (*cl_object lock*) [Function]

`mp:lock-count` *lock* [Function]
Returns number of times *lock* has been locked.

`cl_object mp_get_lock_wait` (*cl_object lock*) [Function]
Grabs a lock (blocking if *lock* is already taken). Returns `ECL_T`.

`cl_object mp_get_lock_nowait` [Function]
Grabs a lock if free (non-blocking). If *lock* is already taken returns `ECL_NIL`, otherwise `ECL_T`.

`mp:get-lock` *lock* **&optional** (*wait* **t**) [Function]
Tries to acquire a lock. *wait* indicates whether function should block or give up if *lock* is already taken. If *wait* is `nil`, immediately return, if *wait* is a real number *wait* specifies a timeout in seconds and otherwise block until the lock becomes available. If *lock* can't be acquired return `nil`. Successful operation returns `t`. Will signal an error if the mutex is non-recursive and current thread already owns the lock.

`cl_object mp_giveup_lock` (*cl_object lock*) [Function]

`mp:giveup-lock` *lock* [Function]
Releases *lock* and returns `t`. May signal an error if the lock is not owned by the current thread.

`mp:with-lock` (*lock-form*) **&body** *body* [Macro]
Acquire lock for the dynamic scope of *body*, which is executed with the lock held by current thread. Returns the values of body.

3.4.6 Readers-writer locks

Readers-writer (or shared-exclusive) locks allow concurrent access for read-only operations, while write operations require exclusive access. `mp:rwlock` is non-recursive and cannot be used together with condition variables.

3.4.7 Read-Write locks dictionary

`cl_object ecl_make_rwlock` (*cl_object name*) [Function]
C/C++ equivalent of `mp:make-rwlock` without `key` arguments.

See `mp:make-rwlock`.

`mp:make-rwlock` **&key** *name* [Function]
Creates a rwlock named *name*.

`cl_object mp_rwlock_name` (*cl_object lock*) [Function]

`mp:rwlock-name` *lock* [Function]
Returns the name of *lock*.

`cl_object mp_get_rwlock_read_wait` (*cl_object lock*) [Function]
Acquires *lock* (blocks if *lock* is already taken with `mp:get-rwlock-write`. Lock may be acquired by multiple readers). Returns `ECL_T`.

`cl_object mp_get_rwlock_read_nowait` [Function]
Tries to acquire *lock*. If *lock* is already taken with `mp:get-rwlock-write` returns `ECL_NIL`, otherwise `ECL_T`.

`mp:get-rwlock-read` *lock* **&optional** (*wait* **t**) [Function]
Tries to acquire *lock*. *wait* indicates whenever function should block or give up if *lock* is already taken with `mp:get-rwlock-write`.

`cl_object mp_get_rwlock_write_wait` (*cl_object lock*) [Function]
Acquires *lock* (blocks if *lock* is already taken). Returns `ECL_T`.

`cl_object mp_get_rwlock_write_nowait` [Function]
Tries to acquire *lock*. If *lock* is already taken returns `ECL_NIL`, otherwise `ECL_T`.

`mp:get-rwlock-write` *lock* **&optional** (*wait* **t**) [Function]
Tries to acquire *lock*. *wait* indicates whenever function should block or give up if *lock* is already taken.

`cl_object mp_giveup_rwlock_read` (*cl_object lock*) [Function]
`cl_object mp_giveup_rwlock_write` (*cl_object lock*) [Function]

`mp:giveup-rwlock-read` *lock* [Function]
`mp:giveup-rwlock-write` *lock* [Function]
Release *lock*.

`mp:with-rwlock` (*lock operation*) **&body** *body* [Macro]
Acquire rwlock for the dynamic scope of *body* for operation *operation*, which is executed with the lock held by current thread. Returns the values of body.

Valid values of argument *operation* are `:read` or `:write` (for reader and writer access accordingly).

3.4.8 Condition variables

Condition variables are used to wait for a particular condition becoming true (e.g new client connects to the server).

3.4.9 Condition variables dictionary

`cl_object mp_make_condition_variable ()` [Function]

`mp:make-condition-variable` [Function]
Creates a condition variable.

`cl_object mp_condition_variable_wait (`*cl_object cv, cl_object lock*`)` [Function]

`mp:condition-variable-wait` *cv lock* [Function]
Release *lock* and suspend thread until `mp:condition-variable-signal` or `mp:condition-variable-broadcast` is called on *cv*. When thread resumes re-aquire *lock*. Always returns `t`. May signal an error if *lock* is not owned by the current thread.

Note: In some circumstances, the thread may wake up even if no call to `mp:condition-variable-signal` or `mp:condition-variable-broadcast` has happened. It is recommended to check for the condition that triggered the wait in a loop around any `mp:condition-variable-wait` call.

Note: While the condition variable is blocked waiting for a signal or broadcast event, calling `mp:condition-variable-wait` from further threads must be done using the same mutex as that used by the threads that are already waiting on this condition variable. The behaviour is undefined if this constraint is violated.

`cl_object mp_condition_variable_timedwait (`*cl_object cv, cl_object lock, cl_object seconds*`)` [Function]

`mp:condition-variable-timedwait` *cv lock seconds* [Function]
`mp:condition-variable-wait` which timeouts after *seconds* seconds. Returns `nil` on timeout and `t` otherwise. May signal an error if *lock* is not owned by the current thread.

`cl_object mp_condition_variable_signal (`*cl_object cv*`)` [Function]

`mp:condition-variable-signal` *cv* [Function]
Wake up at least one of the waiters of *cv*. Usually, this will wake up only a single thread, but it may also wake up multiple threads. Always returns `t`.

See `mp:condition-variable-wait`.

`cl_object mp_condition_variable_broadcast (`*cl_object cv*`)` [Function]

`mp:condition-variable-broadcast` *cv* [Function]
Wake up all waiters of *cv*. Always returns `t`.

See `mp:condition-variable-wait`.

3.4.10 Semaphores

Semaphores are objects which allow an arbitrary resource count. Semaphores are used for shared access to resources where number of concurrent threads allowed to access it is limited.

3.4.11 Semaphores dictionary

`cl_object ecl_make_semaphore` (*cl_object name*, *cl_fixnum count*) [Function]
C/C++ equivalent of `mp:make-semaphore` without `key` arguments.

See `mp:make-semaphore`.

`mp:make-semaphore` **&key** *name count* [Function]
Creates a counting semaphore *name* with a resource count *count*.

`cl_object mp_semaphore_name` (*cl_object semaphore*) [Function]

`mp:semaphore-name` *semaphore* [Function]
Returns the name of *semaphore*.

`cl_object mp_semaphore_count` (*cl_object semaphore*) [Function]

`mp:semaphore-count` *semaphore* [Function]
Returns the resource count of *semaphore*.

`cl_object mp_semaphore_wait_count` (*cl_object semaphore*) [Function]

`mp:semaphore-wait-count` *semaphore* [Function]
Returns the number of threads waiting on *semaphore*.

`cl_object mp_sempahore_wait(cl_object` *semaphore*, *cl_object count*, *cl_object timeout*) [Function]

`mp:semaphore-wait` *semaphore count timeout* [Function]
Decrement the count of *semaphore* by *count* if the count would not be negative.

Else blocks until the semaphore can be decremented. Returns the old count of *semaphore* on success.

If timeout is not `nil`, it is the maximum number of seconds to wait. If the count cannot be decremented in that time, returns `nil` without decrementing the count.

`cl_object mp_wait_on_semaphore` (*cl_narg n*, *cl_object sem*, ...) [Function]

`mp:wait-on-semaphore` *semaphore* **&key** *count timeout* [Function]
Waits on semaphore until it can grab *count* resources.

Returns resource count before semaphore was acquired.

This function is equivalent to `(mp:semaphore-wait semaphore count timeout)`

`cl_object mp_try_get_semaphore` (*cl_narg n*, *cl_object sem*, ...) [Function]

`mp:try-get-semaphore` *semaphore* **&optional** *count* [Function]
Tries to get a semaphore (non-blocking).

If there is no enough resource returns `nil`, otherwise returns resource count before semaphore was acquired.

This function is equivalent to `(mp:semaphore-wait semaphore count 0)`

`cl_object mp_signal_semaphore` (*cl_narg n*, *cl_object sem*, ...); [Function]

`mp:signal-semaphore` *semaphore* **&optional** (*count* **1**) [Function]
Releases *count* units of a resource on *semaphore*. Returns no values.

3.4.12 Barriers

Barriers are objects which for a group of threads make them stop and they can't proceed until all other threads reach the barrier.

3.4.13 Barriers dictionary

`cl_object ecl_make_barrier` (*cl_object name*, *cl_index count*) [Function]
C/C++ equivalent of `mp:make-barrier` without `key` arguments.

See `mp:make-barrier`.

`mp:make-barrier` *count* **&key** *name* [Function]
Creates a barrier *name* with a thread count *count*.

`mp:barrier-count` *barrier* [Function]
Returns the count of *barrier*.

`mp:barrier-name` *barrier* [Function]
Returns the name of *barrier*.

`mp:barrier-arrivers-count` *barrier* [Function]
Returns the number of threads waiting on *barrier*.

`mp:barrier-wait` *barrier* [Function]
The caller thread waits on *barrier*. When the barrier is saturated then all threads waiting on it are unblocked. Returns `t` if the calling thread had to wait to pass the barrier, `:unblocked` if the barrier is enabled but could be passed without waiting and `nil` if the barrier is disabled.

`mp:barrier-unblock` *barrier* **&key** *reset-count disable kill-waiting* [Function]
Forcefully wakes up all processes waiting on the barrier.

reset-count when used resets *barrier* counter.

disable disables or enables *barrier*. When a barrier is disabled then all calls to `mp:barrier-wait` immedietely return.

kill-waiting is used to kill all woken threads.

Returns no values.

3.4.14 Atomic operations

ECL supports both compare-and-swap and fetch-and-add (which may be faster on some processors) atomic operations on a number of different places. The compare-and-swap macro is user extensible with a protocol similar to `setf`.

3.4.15 Atomic operations dictionary

C Reference

`cl_object ecl_compare_and_swap` (*cl_object *slot*, *cl_object old*, *cl_object new*) [Function]
Perform an atomic compare and swap operation on *slot* and return the previous value stored in *slot*. If the return value is equal to *old* (comparison by `==`), the operation has succeeded. This is a inline-only function defined in `"ecl/ecl_atomics.h"`.

`cl_object ecl_atomic_incf` (*cl_object *slot, cl_object increment*) [Function]

`cl_object ecl_atomic_incf_by_fixnum` (*cl_object *slot, cl_fixnum increment*) [Function]

Atomically increment *slot* by the given increment and return the previous value stored in *slot*. The consequences are undefined if the value of *slot* is not of type `fixnum`. `ecl_atomic_incf` signals an error if *increment* is not of type `fixnum`. This is a inline-only function defined in "ecl/ecl_atomics.h".

`cl_index ecl_atomic_index_incf` (*cl_index *slot*); [Function]

Atomically increment *slot* by 1 and return the new value stored in *slot*.

`cl_object ecl_atomic_get` (*cl_object *slot*) [Function]

Perform a volatile load of the object in *slot* and then atomically set *slot* to `ECL_NIL`. Returns the value previously stored in *slot*.

`void ecl_atomic_push` (*cl_object *slot, cl_object o*) [Function]

`cl_object ecl_atomic_pop` (*cl_object *slot*) [Function]

Like push/pop but atomic.

Lisp Reference

`mp:atomic-incf` *place* **&optional** (*increment* **1**) [Macro]
`mp:atomic-decf` *place* **&optional** (*increment* **1**) [Macro]

Atomically increments/decrements the fixnum stored in *place* by the given *increment* and returns the value of *place* before the increment. Incrementing and decrementing is done using modular arithmetic, so that `mp:atomic-incf` of a place whose value is `most-positive-fixnum` by 1 results in `most-negative-fixnum` stored in place.

Currently the following places are supported:

`car`, `cdr`, `first`, `rest`, `svref`, `symbol-value`, `slot-value`, `clos:standard-instance-access`, `clos:funcallable-standard-instance-access`.

For `slot-value`, the object should have no applicable methods defined for `slot-value-using-class` or `(setf slot-value-using-class)`.

The consequences are undefined if the value of *place* is not of type `fixnum`.

`mp:compare-and-swap` *place old new* [Macro]

Atomically stores *new* in *place* if *old* is `eq` to the current value of *place*. Returns the previous value of *place*: if the returned value is `eq` to *old*, the swap was carried out.

Currently, the following places are supported:

`car`, `cdr`, `first`, `rest`, `svref`, `symbol-plist`, `symbol-value`, `slot-value`, `clos:standard-instance-access`, `clos:funcallable-standard-instance-access`, a structure slot accessor[3] or any other place for which a compare-and-swap expansion was defined by `mp:defcas` or `mp:define-cas-expander`.

[3] The creation of atomic structure slot accessors can be deactivated by supplying a `(:atomic-accessors nil)` option to `defstruct`.

For `slot-value`, `slot-unbound` is called if the slot is unbound unless *old* is `eq` to `si:unbound`, in which case *old* is returned and *new* is assigned to the slot. Additionally, the object should have no applicable methods defined for `slot-value-using-class` or `(setf slot-value-using-class)`.

`mp:atomic-update` *place update-fn* **&rest** *arguments* [Macro]

Atomically updates the CAS-able *place* to the value returned by calling *update-fn* with *arguments* and the old value of *place*. *update-fn* must be a function accepting `(1+ (length arguments))` arguments. Returns the new value which was stored in *place*.

place may be read and *update-fn* may be called more than once if multiple threads are trying to write to *place* at the same time.

Example:

Atomic update of a structure slot. If the update would not be atomic, the result would be unpredictable.

```
(defstruct test-struct
  (slot1 0))
(let ((struct (make-test-struct)))
  (mapc #'mp:process-join
        (loop repeat 100
           collect (mp:process-run-function
                    ""
                    (lambda ()
                      (loop repeat 1000 do
                           (mp:atomic-update (test-struct-slot1 struct) #'1+)
                           (sleep 0.00001))))))
  (test-struct-slot1 struct))
=> 100000
```

`mp:atomic-push` *obj place* [Macro]
`mp:atomic-pop` *place* [Macro]

Like `push`/`pop`, but atomic. *place* must be CAS-able and may be read multiple times before the update succeeds.

`mp:define-cas-expander` *accessor lambda-list* **&body** *body* [Macro]

Define a compare-and-swap expander similar to `define-setf-expander`. Defines the compare-and-swap-expander for generalized-variables `(accessor ...)`. When a form `(mp:compare-and-swap (accessor arg1 ... argn) old new)` is evaluated, the forms given in the body of `mp:define-cas-expander` are evaluated in order with the parameters in `lambda-list` bound to `arg1 ... argn`. The body must return six values

```
(var1 ... vark)
(form1 ... formk)
old-var
new-var
compare-and-swap-form
volatile-access-form
```

in order (Note that `old-var` and `new-var` are single variables, unlike in `define-setf-expander`). The whole `compare-and-swap` form is then expanded into

```
(let* ((var1 from1) ... (vark formk)
       (old-var old-form)
       (new-var new-form))
  compare-and-swap-form).
```

Note that it is up to the user of this macro to ensure atomicity for the resulting compare-and-swap expansions.

Example

`mp:define-cas-expander` can be used to define a more convienient compare-and-swap expansion for a class slot. Consider the following class:

```
(defclass food ()
  ((name :initarg :name)
   (deliciousness :initform 5 :type '(integer 0 10)
                  :accessor food-deliciousness)))

(defvar *spätzle* (make-instance 'food :name "Spätzle"))
```

We can't just use `mp:compare-and-swap` on `*spätzle*`:

```
> (mp:compare-and-swap (food-deliciousness *x*) 5 10)

Condition of type: SIMPLE-ERROR
Cannot get the compare-and-swap expansion of (FOOD-DELICIOUSNESS *X*).
```

We can use `symbol-value`, but let's define a more convenient compare-and-swap expander:

```
(mp:define-cas-expander food-deliciousness (food)
  (let ((old (gensym))
        (new (gensym)))
    (values nil nil old new
            `(progn (check-type ,new (integer 0 10))
                    (mp:compare-and-swap (slot-value ,food 'deliciousness)
                                         ,old ,new))
            `(food-deliciousness ,food))))
```

Now finally, we can safely store our rating:

```
> (mp:compare-and-swap (food-deliciousness *spätzle*) 5 10)

5
```

`mp:defcas` *accessor cas-fun* **&optional** *documentation* [Macro]

Define a compare-and-swap expansion similar to the short form of `defsetf`. Defines an expansion

```
(compare-and-swap (accessor arg1 ... argn) old new)
=> (cas-fun arg1 ... argn old new)
```

Note that it is up to the user of this macro to ensure atomicity for the resulting compare-and-swap expansions.

`mp:remcas` *symbol* [Function]
Remove a compare-and-swap expansion. It is an equivalent of `fmakunbound (setf symbol)` for cas expansions.

`mp:get-cas-expansion` *place* **&optional** *environment* [Function]
Returns the compare-and-swap expansion forms and variables as defined in `mp:define-cas-expander` for *place* as six values.

3.5 Signals and Interrupts

3.5.1 Problems associated to signals

POSIX contemplates the notion of "signals", which are events that cause a process or a thread to be interrupted. Windows uses the term "exception", which includes also a more general kind of errors.

In both cases the consequence is that a thread or process may be interrupted at any time, either by causes which are intrinsic to them (synchronous signals), such as floating point exceptions, or extrinsic (asynchronous signals), such as the process being aborted by the user.

Of course, those interruptions are not always welcome. When the interrupt is delivered and a handler is invoked, the thread or even the whole program may be in an inconsistent state. For instance the thread may have acquired a lock, or it may be in the process of filling the fields of a structure. Furthermore, sometimes the signal that a process receives may not even be related to it, as in the case when a user presses *`Cltr-C`* and a `SIGINT` signal is delivered to an arbitrary thread, or when the process receives the Windows exception `CTRL_CLOSE_EVENT` denoting that the terminal window is being closed.

Understanding this, POSIX restricts severely what functions can be called from a signal handler, thereby limiting its usefulness. However, Common Lisp users expect to be able to handle floating point exceptions and to gracefully manage user interrupts, program exits, etc. In an attempt to solve this seemingly impossible problem, ECL has taken a pragmatic approach that works, it is rather safe, but involves some work on the ECL maintainers and also on users that want to embed ECL as a library.

3.5.2 Kinds of signals

3.5.2.1 Synchronous signals

The name derives from POSIX and it denotes interrupts that occur due to the code that a particular thread executes. They are largely equivalent to C++ and Java exceptions, and in Windows they are called "unchecked exceptions."

Common Lisp programs may generate mostly three kinds of synchronous signals:

- Floating point exceptions, that result from overflows in computations, division by zero, and so on.
- Access violations, such as dereferencing NULL pointers, writing into regions of memory that are protected, etc.
- Process interrupts.

The first family of signals are generated by the floating point processing hardware in the computer, and they typically happen when code is compiled with low security settings, performing mathematical operations without checks.

The second family of signals may seem rare, but unfortunately they still happen quite often. One scenario is wrong code that handles memory directly via FFI. Another one is undetected stack overflows, which typically result in access to protected memory regions. Finally, a very common cause of these kind of exceptions is invoking a function that has been compiled with very low security settings with arguments that are not of the expected type – for instance, passing a float when a structure is expected.

The third family is related to the multiprocessing capabilities in Common Lisp systems and more precisely to the `mp:interrupt-process` function which is used to kill, interrupt and inspect arbitrary threads. In POSIX systems ECL informs a given thread about the need to interrupt its execution by sending a particular signal from the set which is available to the user.

Note that in neither of these cases we should let the signal pass unnoticed. Access violations and floating point exceptions may propagate through the program causing more harm than expected, and without process interrupts we will not be able to stop and cancel different threads. The only question that remains, though, is whether such signals can be handled by the thread in which they were generated and how.

3.5.2.2 Asynchronous signals

In addition to the set of synchronous signals or "exceptions", we have a set of signals that denote "events", things that happen while the program is being executed, and "requests". Some typical examples are:

- Request for program termination (`SIGKILL`, `SIGTERM`).
- Indication that a child process has finished.
- Request for program interruption (`SIGINT`), typically as a consequence of pressing a key combination, e.g. *`Ctrl-C`*.

The important difference with synchronous signals is that we have no thread that causes the interrupt and thus there is no preferred way of handling them. Moreover, the operating system will typically dispatch these signals to an arbitrary thread, unless we set up mechanisms to prevent it. This can have nasty consequences if the incoming signal interrupt a system call, or leaves the interrupted thread in an inconsistent state.

3.5.3 Signals and interrupts in ECL

The signal handling facilities in ECL are constrained by two needs. First of all, we can not ignore the synchronous signals mentioned in Section 3.5.2.1 [Signals and Interrupts - Synchronous signals], page 143. Second, all other signals should cause the least harm to the running threads. Third, when a signal is handled synchronously using a signal handler, the handler should do almost nothing unless we are completely sure that we are in an interruptible region, that is outside system calls, in code that ECL knows and controls.

The way in which this is solved is based on the existence of both synchronous and asynchronous signal handling code, as explained in the following two sections.

3.5.3.1 Handling of asynchronous signals

In systems in which this is possible, ECL creates a signal handling thread to detect and process asynchronous signals (See Section 3.5.2.2 [Signals and Interrupts - Asynchronous signals], page 144). This thread is a trivial one and does not process the signals itself: it communicates with, or launches new signal handling threads to act accordingly to the denoted events.

The use of a separate thread has some nice consequences. The first one is that those signals will not interrupt any sensitive code. The second one is that the signal handling thread will be able to execute arbitrary lisp or C code, since it is not being executed in a sensitive context. Most important, this style of signal handling is the recommended one by the POSIX standards, and it is the one that Windows uses.

The installation of the signal handling thread is dictated by a boot time option, `ECL_OPT_SIGNAL_HANDLING_THREAD` (see Table 1.1 for a summary of boot options), and it will only be possible in systems that support either POSIX or Windows threads.

Systems which embed ECL as an extension language may wish to deactivate the signal handling thread using the previously mentioned option. If this is the case, then they should take appropriate measures to avoid interrupting the code in ECL when such signals are delivered.

Systems which embed ECL and do not mind having a separate signal handling thread can control the set of asynchronous signals which is handled by this thread. This is done again using the appropriate boot options such as `ECL_OPT_TRAP_SIGINT`, `ECL_OPT_TRAP_SIGTERM`, etc. Note that in order to detect and handle those signals, ECL must block them from delivery to any other thread. This means changing the `sigprocmask()` in POSIX systems or setting up a custom `SetConsoleCtrlHandler()` in Windows.

3.5.3.2 Handling of synchronous signals

We have already mentioned that certain synchronous signals and exceptions can not be ignored and yet the corresponding signal handlers are not able to execute arbitrary code. To solve this seemingly impossible contradiction, ECL uses a simple solution, which is to mark the sections of code which are interruptible, and in which it is safe for the handler to run arbitrary code. All other regions would be considered "unsafe" and would be protected from signals and exceptions.

In principle this "marking" of safe areas can be done using POSIX functions such as `pthread_sigmask()` or `sigprocmask()`. However in practice this is slow, as it involves at least a function call, resolving thread-local variables, etc, etc, and it will not work in Windows.

Furthermore, sometimes we want signals to be detected but not to be immediately processed. For instance, when reading from the terminal we want to be able to interrupt the process, but we can not execute the code from the handler, since the C function which is used to read from the terminal, `read()`, may have left the input stream in an inconsistent, or even locked state.

The approach in ECL is more lightweight: we install our own signal handler and use a thread-local variable as a flag that determines whether the thread is executing interrupt safe code or not. More precisely, if the variable `ecl_process_env()->disable_interrupts` is set, signals and exceptions will be postponed and then the information about the signal is

queued. Otherwise the appropriate code is executed: for instance invoking the debugger, jumping to a condition handler, quitting, etc.

Systems that embed ECL may wish to deactivate completely these signal handlers. This is done using the boot options, `ECL_OPT_TRAP_SIGFPE`, `ECL_OPT_TRAP_SIGSEGV`, `ECL_OPT_TRAP_SIGBUS`, `ECL_OPT_TRAP_INTERRUPT_SIGNAL`.

Systems that embed ECL and want to allow handling of synchronous signals should take care to also trap the associated lisp conditions that may arise. This is automatically taken care of by functions such as `si_safe_eval`, and in all other cases it can be solved by enclosing the unsafe code in a `ECL_CATCH_ALL` frame.

3.5.4 Considerations when embedding ECL

There are several approaches when handling signals and interrupts in a program that uses ECL. One is to install your own signal handlers. This is perfectly fine, but you should respect the same restrictions as ECL. Namely, you may not execute arbitrary code from those signal handlers, and in particular it will not always be safe to execute Common Lisp code from there.

If you want to use your own signal handlers then you should set the appropriate options before invoking `cl_boot`, as explained in `ecl_set_option`. Note that in this case ECL will not always be able to detect floating point exceptions.

The other option is to let ECL handle signals itself. This would be safer when the dominant part of the code is Common Lisp, but you may need to protect the code that embeds ECL from being interrupted using either the macros `ecl_disable_interrupts` and `ecl_enable_interrupts` or the POSIX functions `pthread_sigmaks` and `sigprocmask`.

3.5.5 Signals Reference

`mp:with-interrupts` **&body** *body* [Macro]
`mp:without-interrupts` **&body** *body* [Macro]

Execute code with interrupts optionally enabled/disabled, See Section 3.4.3 [Processes dictionary], page 130.

`ext:unix-signal-received` [Condition]

Unix signal condition

Class Precedence List `condition`, `t`

Methods

`ext:unix-signal-received-code` *condition* [Function]

Returns the signal code of *condition*

`ext:get-signal-handler` *code* [Function]

Queries the currently active signal handler for *code*.

`ext:set-signal-handler` *code handler* [Function]

Arranges for the signal *code* to be caught in all threads and sets the signal handler for it. The value of *handler* modifies the signal handling behaviour as follows:

handler is a function designator

The function designated by *handler* will be invoked with no arguments

handler is a symbol denoting a condition type
: A continuable error of the given type will be signaled

handler is equal to *code*
: A condition of type `ext:unix-signal-received` with the corresponding signal code will be signaled

handler is `nil`
: The signal will be caught but no handler will be called

`ext:catch-signal` *code flag* **&key** *process* [Function]

Changes the action taken on receiving the signal *code*. *flag* can be one of the following:

`nil` or `:ignore`
: Ignore the signal

`:default` Use the default signal handling strategy of the operating system

`t` or `:catch`
: Catch the signal and invoke the signal handler as given by `ext:get-signal-handler`

`:mask`, `:unmask`
: Change the signal mask of either a) the not yet enabled *process* or b) the current process, if *process* is not supplied

Returns `t` on success and `nil` on failure.

Example:

```
CL-USER> (ext:catch-signal ext:+SIGPIPE+ :catch)
T
CL-USER> (ext:get-signal-handler ext:+SIGPIPE+)
NIL
CL-USER> (ext:set-signal-handler ext:+SIGPIPE+
                                 #'(lambda ()
                                     (format t "SIGPIPE detected in process: ~a~%" mp:
#<bytecompiled-function 0x25ffca8>
```

Passing the SIGPIPE signal to the ECL program with `killall -s SIGPIPE ecl` results in the output:

```
SIGPIPE detected in process: #<process TOP-LEVEL 0x1ecdfc0>
```

3.6 Memory Management

3.6.1 Introduction

ECL relies on the Boehm-Weiser garbage collector for handling memory, creating and destroying objects, and handling finalization of objects that are no longer reachable. The use of a garbage collector, and in particular the use of a portable one, imposes certain restrictions that may appear odd for C/C++ programmers.

In this section we will discuss garbage collection, how ECL configures and uses the memory management library, what users may expect, how to handle the memory and how to control the process by which objects are deleted.

3.6.2 Boehm-Weiser garbage collector

First of all, the garbage collector must be able to determine which objects are alive and which are not. In other words, the collector must able to find all references to an object. One possibility would be to know where all variables of a program reside, and where is the stack of the program and its size, and parse all data there, discriminating references to lisp objects. To do this precisely one would need a very precise control of the data and stack segments, as well as how objects are laid out by the C compiler. This is beyond ECL's scope and wishes and it can make coexistence with other libraries (C++, Fortran, etc) difficult.

The Boehm-Weiser garbage collector, on the other hand, is a conservative garbage collector. When scanning memory looking for references to live data, it guesses, conservatively, whether a word is a pointer or not. In case of doubt it will consider it to be a pointer and add it to the list of live objects. This may cause certain objects to be retained longer than what an user might expect but, in our experience, this is the best of both worlds and ECL uses certain strategies to minimize the amount of misinterpreted data.

More precisely, ECL uses the garbage collector with the following settings:

- The collector will not scan the data sectors. If you embed ECL in another program, or link libraries with ECL, you will have to notify ECL which variables point to lisp objects.
- The collector is configured to ignore pointers that point to the middle of allocated objects. This minimizes the risk of misinterpreting integers as pointers to live objects.
- It is possible to register finalizers that are invoked when an object is destroyed, but for that you should use ECL's API and understand the restriction described later in Section 3.6.5 [Finalization], page 149.

Except for finalization, which is a questionable feature, the previous settings are not very relevant for Common Lisp programmers, but are crucial for people interested in embedding in or cooperating with other C, C++ or Fortran libraries. Care should be taken when manipulating directly the GC library to avoid interfering with ECL's expectations.

3.6.3 Memory limits

Beginning with version 9.2.1, ECL operates a tighter control of the resources it uses. In particular, it features explicit limits in the four stacks and in the amount of live data. These limits are optional, can be changed at run time, but they allow users to better control the evolution of a program, handling memory and stack overflow gracefully via the Common Lisp condition system.

The customizable limits are listed in Table 3.1, but they need a careful description.

- `ext:heap-size` limits the total amount of memory which is available for lisp objects. This is the memory used when you create conses, arrays, structures, etc.
- `ext:c-stack` controls the size of the stack for compiled code, including ECL's library itself. This limit is less stringent than the others. For instance, when code is compiled with low safety settings, checks for this stack limit are usually omitted, for performance reasons.
- `ext:binding-stack` controls the number of nested bindings for special variables. The current value is usually safe enough, unless you have deep recursive functions that bind special variables, which is not really a good idea.

- `ext:frame-stack` controls the number of nested blocks, tagbodys and other control structures. It affects both interpreted and compiled code, but quite often compiled code optimizes away these stack frames, saving memory and not being affected by this limit.
- `ext:lisp-stack` controls the size of the interpreter stack. It only affects interpreted code.

If you look at Table 3.1, some of these limits may seem very stringent, but they exist to allow detecting and correcting both stack and memory overflow conditions. Larger values can be set systematically either in the `~/.eclrc` initialization file, or using the command line options from the table.

3.6.4 Memory conditions

When ECL surpasses or approaches the memory limits it will signal a Common Lisp condition. There are two types of conditions, `ext:stack-overflow` and `ext:storage-exhausted`, for stack and heap overflows, respectively. Both errors are correctable, as the following session shows:

```
> (defun foo (x) (foo x))

FOO
> (foo 1)
C-STACK overflow at size 1654784. Stack can probably be resized.
Broken at SI:BYTECODES.Available restarts:
1. (CONTINUE) Extend stack size
Broken at FOO.
>> :r1
C-STACK overflow at size 2514944. Stack can probably be resized.
Broken at SI:BYTECODES.Available restarts:
1. (CONTINUE) Extend stack size
Broken at FOO.
>> :q
Top level.
```

3.6.5 Finalization

As we all know, Common-Lisp relies on garbage collection for deleting unreachable objects. However, it makes no provision for the equivalent of a C++ Destructor function that should be called when the object is eliminated by the garbage collector. The equivalent of such methods in a garbage collected environment is normally called a *finalizer*.

ECL includes a simple implementation of finalizers which makes the following promises.

- The finalizer can be any lisp function, let it be compiled or interpreter.
- Finalizers are not invoked during garbage collection. Instead, if an unreachable object is found to have an associated finalizer, it is pushed into a list and *before the next garbage collection cycle*, the finalizer will be invoked.
- If the finalizer is invoked and it makes the object reachable, for instance, by assigning it to a variable, it will not be destroyed, but it will have no longer a finalizer associated to it.

- ECL will strive to call finalizers before the environment is closed and the program is finished, but this mechanism may fail when exiting in a non ordinary way.

The implementation is based on two functions, `ext:set-finalizer` and `ext:get-finalizer`, which allow setting and querying the finalizer functions for certain objects.

3.6.6 Memory Management Reference

Reference

`ext:stack-overflow` [Condition]

Stack overflow condition

Class Precedence List `ext:stack-overflow`, `storage-condition`, `serious-condition`, `condition`, `t`

Methods

`ext:stack-overflow-size` *condition* [Function]

returns A non-negative integer.

`ext:stack-overflow-type` *condition* [Function]

returns A symbol from Table 3.1, except `ext:heap-size`.

Description This condition is signaled when one of the stack limits in Table 3.1 are violated or dangerously approached. It can be handled by resetting the limits and continuing, or jumping to an outer control point.

`ext:storage-exhausted` [Condition]

Memory overflow condition

Class Precedence List `ext:storage-exhausted`, `storage-condition`, `serious-condition`, `condition`, `t`

Description This condition is signaled when ECL exhausts the `ext:heap-size` limit from Table 3.1. In handling this condition ECL follows this logic:

- If the heap size limit was set to 0 (that is no limit), but there is some free space in the safety region ECL frees this space and issues a non-restartable error. The user may jump to an outer point or quit.
- If the heap size had a finite limit, ECL offers the user the chance to resize it, issuing a restartable condition. The user may at this point use `(ext:set-limit 'ext:heap-size 0)` to remove the heap limit and avoid further messages, or use the `(continue)` restart to let ECL enlarge the heap by some amount.
- Independently of the heap size limit, if ECL finds that there is no space to free or to grow, ECL simply quits. There will be no chance to do some cleanup because there is no way to allocate any additional data.

`ext:get-finalizer` *object* [Function]

object Any lisp object.

Description This function returns the finalizer associated to an object, or `nil`.

`ext:get-limit` *concept* [Function]

concept A symbol.

Description Queries the different memory and stack limits that condition ECL's behavior. The value to be queried is denoted by the symbol *concept*, which should be one from the list: Table 3.1

`ext:set-finalizer` *object function* [Function]

Associate a finalizer to an object.

object Any lisp object.

function A function or closure that takes one argument or `nil`.

Description If *function* is `nil`, no finalizer is associated to the object. Otherwise *function* must be a function or a closure of one argument, which will be invoked before the object is destroyed.

Example Close a file associated to an object.

```
(defclass my-class () ((file :initarg :file :initform nil)))

(defun finalize-my-class (x)
 (let ((s (slot-value x 'file)))
   (when s (format t "~%;;; Closing" s) (close s))))

(defmethod initialize-instance :around ((my-instance my-class) &rest args)
  (ext:set-finalizer my-instance #'finalize-my-class)
  (call-next-method))

(progn
  (make-instance 'my-class :file (open "~/.ecl.old" :direction :input))
  nil)

(si::gc t)
(si::gc t)

;; Closing
```

`ext:set-limit` *concept value* [Function]

Set a memory or stack limit.

concept A symbol.

value A positive integer.

Changes the different memory and stack limits that condition ECL's behavior. The value to be changed is denoted by the symbol *concept*, while the *value* is the new maximum size. The valid symbols and units are listed in Table 3.1.

Note that the limit has to be positive, but it may be smaller than the previous value of the limit. However, if the supplied value is smaller than what ECL is using at the moment, the new value will be silently ignored.

Concept	Units	Default	Command line
ext:frame-stack	Nested frames	2048	`--frame-stack`
ext:binding-stack	Bindings	8192	
ext:c-stack	Bytes	128 kilobytes	`--c-stack`
ext:heap-size	Bytes	256 megabytes	`--heap-size`
ext:lisp-stack	Bytes	32 kilobyes	`--lisp-stack`

Table 3.1: Customizable memory limits

3.7 Meta-Object Protocol (MOP)

3.7.1 Introduction

The Meta-Object Protocol is an extension to Common Lisp which provides rules, functions and a type structure to handle the object system. It is a reflective system, where classes are also objects and can be created and manipulated using very well defined procedures.

The Meta-Object Protocol associated to Common Lisp's object system was introduced in a famous book, The Art of the Metaobject Protocol AMOP [AMOP, see [Bibliography], page 203], which was probably intended for the ANSI [ANSI, see [Bibliography], page 203] specification but was dropped because of its revolutionary and then not too well tested ideas.

The AMOP is present, in one way or another, in most Common Lisp implementations, either using proprietary systems or because their implementation of CLOS descended from PCL (Portable CommonLoops). It has thus become a de facto standard and ECL should not be without it.

Unfortunately ECL's own implementation originally contained only a subset of the AMOP. This was a clever decision at the time, since the focus was on performance and on producing a stable and lean implementation of Common Lisp. Nowadays it is however not an option, especially given that most of the AMOP can be implemented with little cost for both the implementor and the user.

So ECL has an almost complete implementation of the AMOP. However, since it was written from scratch and progressed according to user's request and our own innovations, there might still be some missing functionality which we expect to correct in the near future. Please report any feature you miss as a bug through the appropriate channels.

3.8 Gray Streams

`close` *stream* **&key** *abort* [Function]

Unlike the other Gray stream functions, `close` is not specialized on `t` for *stream*. This decision has been taken mainly for the compatibility reasons with some libraries.

`stream-file-position` *stream* **&optional** *position* [Function]

This is used to implement `file-position`. When `position` is not provided it should return the current file position of the stream as non-negative integer or `nil` if the file position cannot be determined. When `position` is supplied the file position of the

stream should be set to that value. If setting the position is successful then `t` should be returned, otherwise `nil` should be returned. The default method always returns `nil`.

`stream-file-length` *stream* [Function]
This is used to implement `file-length`. It returns either a non-negative integer or `nil` if the concept of file length is not meaningful for the stream. The default method will signal a `type-error` with an expected type of `file-stream`. This is required to conform with the "Exceptional Situations" section of `file-length` in the ANSI specification.

`stream-interactive-p` *stream* [Function]
This is used to implement `interactive-stream-p`. It returns a boolean indicating if the stream is interactive. The default method always returns `nil`.

`stream-line-length` *stream* [Function]
Allows the default line length to be specified for the stream. It returns either a non-negative integer or `nil` if the concept of line length is not meaningful for the stream. This value is only used if `*print-right-margin*` is `nil`. The line length is used by the pretty printer and by the format justification directive. The default method returns `nil`.

`stream-read-sequence` *stream sequence* **&optional** *start end* [Function]
This is used to implement `read-sequence`. It should follow the semantics in the ANSI specification. It returns the position of the first element in the sequence that was not updated. The default method calls `stream-read-char` or `stream-read-byte` repeatedly based on the type returned by `stream-element-type`. Element access to the sequence is done via `elt`.

`stream-write-sequence` *stream sequence* **&optional** *start end* [Function]
This is used to implement `write-sequence`. It should follow the semantics in the ANSI specification. It returns sequence without modification. The default method calls `stream-write-char` or `stream-write-byte` repeatedly based on the type returned by `stream-element-type`. Element access to the sequence is done via `elt`.

3.8.1 Gray Stream Extensions

`stream-unread-byte` *stream byte* [Function]
Un-do the last call to `stream-read-byte`, as in `ext:unread-byte`. Return `nil`. Every subclass of `fundamental-binary-input-stream` should define a method for this function. A default method signals an error.

`stream-peek-byte` *stream* [Function]
This is used to implement `ext:peek-byte`. It returns either a byte or `:eof`. A default method calls `stream-read-byte` and `stream-unread-byte`.

3.9 Tree walker

3.10 Local package nicknames

3.10.1 Overview

ECL allows giving packages local nicknames: they allow short and easy-to-use names to be used without fear of name conflict associated with normal nicknames.

A local nickname is valid only when inside the package for which it has been specified. Different packages can use same local nickname for different global names, or different local nickname for same global name.

The keyword `:package-local-nicknames` in `*features*` indicates the support for this feature.

3.10.2 Package local nicknames dictionary

`cl:defpackage` *name* [[*options*]]* [Macro]

Options are extended to include

```
:local-nicknames (local-nickname actual-package-name)*
```

The package has the specified local nicknames for the corresponding actual packages.

Example:

```
(defpackage :bar (:intern "X"))
(defpackage :foo (:intern "X"))
(defpackage :quux (:use :cl) (:local-nicknames (:bar :foo) (:foo :bar)))
(find-symbol "X" :foo) ; => FOO::X
(find-symbol "X" :bar) ; => BAR::X
(let ((*package* (find-package :quux)))
  (find-symbol "X" :foo))               ; => BAR::X
(let ((*package* (find-package :quux)))
  (find-symbol "X" :bar))               ; => FOO::X
```

`ext:package-local-nicknames` *package-designator* [Function]

`cl_object si_package_local_nicknames` (*cl_object package_designator*) [Function]

Returns an alist of `(local-nickname . actual-package)` describing the nicknames local to the designated package.

When in the designated package, calls to `find-package` with any of the local-nicknames will return the corresponding actual-package instead. This also affects all implied calls to `find-package`, including those performed by the reader.

When printing a package prefix for a symbol with a package local nickname, the local nickname is used instead of the real name in order to preserve print-read consistency.

`ext:package-locally-nicknamed-by-list` *package-designator* [Function]

`cl_object si_package_locally_nicknamed_by_list` (*cl_object package_designator*) [Function]

Returns a list of packages which have a local nickname for the designated package.

`ext:add-package-local-nickname` *local-nickname actual-package* **&optional** *package-designator* [Function]

`cl_object si_add_package_local_nickname` (*cl_object local_nickname, cl_object actual_package, cl_object package_designator*) [Function]

Adds *local-nickname* for *actual-package* in the designated package, defaulting to current package. *local-nickname* must be a string designator, and *actual-package* must be a package designator.

Returns the designated package.

Signals a continuable error if *local-nickname* is already a package local nickname for a different package.

When in the designated package, calls to `find-package` with the *local-nickname* will return the package the designated *actual-package* instead. This also affects all implied calls to `find-package`, including those performed by the reader.

When printing a package prefix for a symbol with a package local nickname, the local nickname is used instead of the real name in order to preserve print-read consistency.

`ext:remove-package-local-nickname` *old-nickname* **&optional** *package-designator* [Function]

`cl_object si_remove_package_local_nickname` (*cl_object old_nickname, cl_object package_designator*) [Function]

If the designated package had *old-nickname* as a local nickname for another package, it is removed. Returns true if the nickname existed and was removed, and `nil` otherwise.

3.11 Package locks

3.11.1 Package Locking Overview

ECL borrows parts of the protocol and documentation from SBCL for compatibility. Interface is the same except that the home package for locking is `ext` and that ECL doesn't implement Implementation Packages and a few constructs. To load the extension you need to require `package-locks`:

```
(require '#:package-locks)
```

Package locks protect against unintentional modifications of a package: they provide similar protection to user packages as is mandated to `common-lisp` package by the ANSI specification. They are not, and should not be used as, a security measure.

Newly created packages are by default unlocked (see the `:lock` option to `defpackage`).

The package `common-lisp` and ECL internal implementation packages are locked by default, including `ext`.

It may be beneficial to lock `common-lisp-user` as well, to ensure that various libraries don't pollute it without asking, but this is not currently done by default.

3.11.2 Operations Violating Package Locks

The following actions cause a package lock violation if the package operated on is locked, and `*package*` is not an implementation package of that package, and the action would

cause a change in the state of the package (so e.g. exporting already external symbols is never a violation). Package lock violations caused by these operations signal errors of type `package-error`.

1. Shadowing a symbol in a package.
2. Importing a symbol to a package.
3. Uninterning a symbol from a package.
4. Exporting a symbol from a package.
5. Unexporting a symbol from a package.
6. Changing the packages used by a package.
7. Renaming a package.
8. Deleting a package.
9. Attempting to redefine a function in a locked package.
10. Adding a new package local nickname to a package.
11. Removing an existing package local nickname to a package.

3.11.3 Package Lock Dictionary

`ext:package-locked-p` *package* [Function]
Returns `t` when *package* is locked, `nil` otherwise. Signals an error if *package* doesn't designate a valid package.

`ext:lock-package` *package* [Function]
Locks *package* and returns `t`. Has no effect if package was already locked. Signals an error if package is not a valid *package* designator

`ext:unlock-package` *package* [Function]
Unlocks *package* and returns `t`. Has no effect if *package* was already unlocked. Signals an error if *package* is not a valid package designator.

`ext:without-package-locks` **&body** *body* [Macro]
Ignores all runtime package lock violations during the execution of body. Body can begin with declarations.

`ext:with-unlocked-packages` (**&rest** *packages*) **&body** *body* [Macro]
Unlocks *packages* for the dynamic scope of the *body*. Signals an error if any of *packages* is not a valid package designator.

`cl:defpackage` *name* [[*option*]]* ⇒ *package* [Macro]
Options are extended to include

```
:lock boolean
```

If the argument to `:lock` is `t`, the package is initially locked. If `:lock` is not provided it defaults to `nil`.

Example:

```
(defpackage "FOO" (:export "BAR") (:lock t))

;;; is equivalent to
```

```
(defpackage "FOO") (:export "BAR"))
(lock-package "FOO")
```

3.12 CDR Extensions

ECL currently implements the following specifications of the Common Lisp Document Repository (`https://common-lisp.net/project/cdr/`):

Number	Comments
1	In `clos` package; partial implementation, see Section 3.7 [Meta-Object Protocol (MOP)], page 152,
5	In `ext` package
7	Only if ECL is compiled with `--with-cmuformat` configure option
14	

Table 3.2: Implemented CDR extensions

4 Developer's guide

4.1 Sources structure

4.1.1 src/c

`alloc_2.d`	memory allocation based on the Boehm GC
`all_symbols.d`	name mangler and symbol initialization
`apply.d`	interface to C call mechanism
`arch/*`	architecture dependant code
`array.d`	array routines
`assignment.c`	assignment
`backq.d`	backquote mechanism
`big.d`	bignum routines based on the GMP
`big_ll.d`	bignum emulation with long long
`cfun.d`	compiled functions
`cfun_dispatch.d`	trampolines for functions
`character.d`	character routines
`char_ctype.d`	character properties.
`cinit.d`	lisp initialization
`clos/accessor.d`	dispatch for slots
`clos/cache.d`	thread-local cache for a variety of operations
`clos/gfun.d`	dispatch for generic functions
`clos/instance.d`	CLOS interface
`cmpaux.d`	auxiliaries used in compiled Lisp code
`compiler.d`	bytecode compiler

`cons.d`	list manipulation macros & functions (auto generated)
`disassembler.d`	bytecodes disassembler utilities
`dpp.c`	defun preprocessor
`ecl_constants.h`	constant values for all_symbols.d
`ecl_features.h`	names of features compiled into ECL
`error.d`	error handling
`eval.d`	evaluation
`ffi/backtrace.d`	C backtraces
`ffi/cdata.d`	data for compiled files
`ffi/libraries.d`	shared library and bundle opening / copying / closing
`ffi/mmap.d`	mapping of binary files
`ffi.d`	user defined data types and foreign functions interface
`file.d`	file interface (implementation dependent)
`format.d`	format (this isn't ANSI compliant, we need it for bootstrapping though)
`hash.d`	hash tables
`interpreter.d`	bytecode interpreter
`iso_latin_names.h`	character names in ISO-LATIN-1
`list.d`	list manipulating routines
`load.d`	binary loader (contains also open_fasl_data)
`macros.d`	macros and environment
`main.d`	ecl boot process
`Makefile.in`	Makefile for ECL core library
`mapfun.d`	mapping

`multival.d`	multiple values
`newhash.d`	hashing routines
`num_arith.d`	arithmetic operations
`number.d`	constructing numbers
`numbers/*.d`	arithmetic operations (abs, atan, plusp etc)
`num_co.d`	operations on floating-point numbers (implementation dependent)
`num_log.d`	logical operations on numbers
`num_pred.d`	predicates on numbers
`num_rand.d`	random numbers
`package.d`	packages (OS dependent)
`pathname.d`	pathnames
`predicate.d`	predicates
`print.d`	print
`printer/*.d`	printer utilities and object representations
`read.d`	reader
`reader/parse_integer.d`	
`reader/parse_number.d`	
`reference.d`	reference in Constants and Variables
`sequence.d`	sequence routines
`serialize.d`	serialize a bunch of lisp data
`sse2.d`	SSE2 vector type support
`stacks.d`	binding/history/frame stacks

`string.d`	string routines
`structure.d`	structure interface
`symbol.d`	symbols
`symbols_list.h`	The file has contains all symbols defined in the core.
`tcp.d`	stream interface to TCP
`time.d`	time routines
`typespec.d`	type specifier routines
`unicode/*`	unicode definitions
`unixfsys.d`	Unix file system interface
`unixint.d`	Unix interrupt interface.
`unixsys.d`	Unix shell interface
`vector_push.d`	vector optimizations
`threads/atomic.d`	atomic operations
`threads/barrier.d`	wait barriers
`threads/condition_variable.d`	condition variables for native threads
`threads/mailbox.d`	thread communication queue
`threads/mutex.d`	mutually exclusive locks.
`threads/process.d`	native threads
`threads/queue.d`	waiting queue for threads
`threads/rwlock.d`	POSIX read-write locks
`threads/semaphore.d`	POSIX-like semaphores

4.2 Contributing

4.3 Defun preprocessor

The defun preprocessor allows for convenient definition of Lisp functions with optional and keyword arguments and the use of Lisp symbols in ECL's own C source code. It generates the C code necessary to use optional function arguments and to access symbols in ECL's builtin symbol table.

Usage:

```
dpp [in-file [out-file]]
```

The file named in-file is preprocessed and the output will be written to the file whose name is out-file. If in-file is "-" program is read from standard input, while if out-file is "-" C-program is written to standard output.

The function definition:

```
@(defun name ({var}*
              [&optional {var | (var [initform [svar]])}*]
              [&rest var]
              [&key {var |
                     ({var | (keyword var)} [initform [svar]])}*
                    [&allow_other_keys]]
              [&aux {var | (var [initform])}*])

        C-declaration

@ {

        C-body

} @)
```

name is the name of the lisp function

&optional may be abbreviated as &o.
&rest may be abbreviated as &r.
&key may be abbreviated as &k.
&allow_other_keys may be abbreviated as &aok.
&aux may be abbreviated as &a.

Each variable becomes a C variable.

Each supplied-p parameter becomes a boolean C variable.

Initforms are C expressions. If an expression contains non-alphanumeric characters, it should be surrounded by backquotes (‘).

Function return:

```
@(return {form}*);
```

Return function expands into a lexical block `{}`, so if it's used inside `if/else`, then it should be enclosed, even if we use sole `@(return);`, because ";" will be treated as the next instruction.

Symbols:

```
@'name'
```

Expands into a C statement, whole value is the given symbol from `symbols_list.h`

`@[name]`

Expands into a C statement, whole value is a fixnum corresponding to the index in the builtin symbols table of the given symbol from `symbols_list.h`. Used for handling type errors.

4.4 Manipulating Lisp objects

If you want to extend, fix or simply customize ECL for your own needs, you should understand how the implementation works.

`cl_lispunion` *cons big ratio SF DF longfloat gencomplex csfloat cdfloat clfloat symbol pack hash array vector base_string string stream random readtable pathname bytecodes bclosure cfun cfunfixed cclosure d instance process queue lock rwlock condition_variable semaphore barrier mailbox cblock foreign frame weak sse* [C/C++ identifier]

Union containing all first-class ECL types.

4.4.1 Objects representation

In ECL a lisp object is represented by a type called `cl_object`. This type is a word which is long enough to host both an integer and a pointer. The least significant bits of this word, also called the tag bits, determine whether it is a pointer to a C structure representing a complex object, or whether it is an immediate data, such as a fixnum or a character.



Figure 4.1: Immediate types

The topic of the immediate values and bit fiddling is nicely described in Peter Bex's blog (`http://www.more-magic.net/posts/internals-data-representation.html`) describing Chicken Scheme (`http://www.call-cc.org/`) internal data representation. We could borrow some ideas from it (like improving `fixnum` bitness and providing more immediate values). All changes to code related to immediate values should be carefully **benchmarked**.

The `fixnums` and characters are called immediate data types, because they require no more than the `cl_object` datatype to store all information. All other ECL objects are non-immediate and they are represented by a pointer to a cell that is allocated on the heap. Each cell consists of several words of memory and contains all the information related to that object. By storing data in multiples of a word size, we make sure that the least significant bits of a pointer are zero, which distinguishes pointers from immediate data.

In an immediate datatype, the tag bits determine the type of the object. In non-immediate datatypes, the first byte in the cell contains the secondary type indicator, and distinguishes between different types of non immediate data. The use of the remaining bytes differs for each type of object. For instance, a cons cell consists of three words:

```
+---------+----------+
| CONS    |          |
+---------+----------+
|     car-pointer    |
+--------------------+
|     cdr-pointer    |
+--------------------+
```

Note, that this is one of the possible implementation of `cons`. The second one (currently default) uses the immediate value for the `list` and consumes two words instead of three. Such implementation is more memory and speed efficient (according to the comments in the source code):

```
/*
 * CONSES
 *
 * We implement two variants. The "small cons" type carries the type
 * information in the least significant bits of the pointer. We have
 * to do some pointer arithmetics to find out the CAR / CDR of the
 * cons but the overall result is faster and memory efficient, only
 * using two words per cons.
 *
 * The other scheme stores conses as three-words objects, the first
 * word carrying the type information. This is kept for backward
 * compatibility and also because the oldest garbage collector does
 * not yet support the smaller datatype.
 *
 * To make code portable and independent of the representation, only
 * access the objects using the common macros below (that is all
 * except ECL_CONS_PTR or ECL_PTR_CONS).
 */
```

`cl_object` [C/C++ identifier]

This is the type of a lisp object. For your C/C++ program, a cl_object can be either a fixnum, a character, or a pointer to a union of structures (See `cl_lispunion` in the header object.h). The actual interpretation of that object can be guessed with the macro `ecl_t_of`.

Example

For example, if x is of type cl_object, and it is of type fixnum, we may retrieve its value:

```
if (ecl_t_of(x) == t_fixnum)
    printf("Integer value: %d\n", ecl_fixnum(x));
```

Example

If `x` is of type `cl_object` and it does not contain an immediate datatype, you may inspect the cell associated to the lisp object using `x` as a pointer. For example:

```
if (ecl_t_of(x) == t_vector)
    printf("Vector's dimension is: %d\n", x->vector.dim);
```

You should see the following sections and the header object.h to learn how to use the different fields of a `cl_object` pointer.

`cl_type` [C/C++ identifier]

Enumeration type which distinguishes the different types of lisp objects. The most important values are:

`t_cons` `t_fixnum`, `t_character`, `t_bignum`, `t_ratio`, `t_singlefloat`, `t_doublefloat`, `t_complex`, `t_symbol`, `t_package`, `t_hashtable`, `t_array`, `t_vector`, `t_string`, `t_bitvector`, `t_stream`, `t_random`, `t_readtable`, `t_pathname`, `t_bytecodes`, `t_cfun`, `t_cclosure`, `t_gfun`, `t_instance`, `t_foreign` and `t_thread`.

`cl_type ecl_t_of` (*cl_object x*) [Function]

If *x* is a valid lisp object, `ecl_t_of(x)` returns an integer denoting the type that lisp object. That integer is one of the values of the enumeration type `cl_type`.

`bool ECL_CHARACTERP` (*cl_object o*) [Function]
`bool ECL_BASE_CHAR_P` (*cl_object o*) [Function]
`bool ECL_BASE_CHAR_CODE_P` (*ecl_character o*) [Function]
`bool ECL_NUMBER_TYPE_P` (*cl_object o*) [Function]
`bool ECL_COMPLEXP` (*cl_object o*) [Function]
`bool ECL_REAL_TYPE_P` (*cl_object o*) [Function]
`bool ECL_FIXNUMP` (*cl_object o*) [Function]
`bool ECL_BIGNUMP` (*cl_object o*) [Function]
`bool ECL_SINGLE_FLOAT_P` (*cl_object o*) [Function]
`bool ECL_DOUBLE_FLOAT_P` (*cl_object o*) [Function]
`bool ECL_LONG_FLOAT_P` (*cl_object o*) [Function]
`bool ECL_CONSP` (*cl_object o*) [Function]
`bool ECL_LISTP` (*cl_object o*) [Function]
`bool ECL_ATOM` (*cl_object o*) [Function]
`bool ECL_SYMBOLP` (*cl_object o*) [Function]
`bool ECL_ARRAYP` (*cl_object o*) [Function]
`bool ECL_VECTORP` (*cl_object o*) [Function]
`bool ECL_BIT_VECTOR_P` (*cl_object o*) [Function]
`bool ECL_STRINGP` (*cl_object o*) [Function]
`bool ECL_HASH_TABLE_P` (*cl_object o*) [Function]
`bool ECL_RANDOM_STATE_P` (*cl_object o*) [Function]
`bool ECL_PACKAGEP` (*cl_object o*) [Function]
`bool ECL_PATHNAMEP` (*cl_object o*) [Function]
`bool ECL_READTABLEP` (*cl_object o*) [Function]
`bool ECL_FOREIGN_DATA_P` (*cl_object o*) [Function]

`bool ECL_SSE_PACK_P` (*cl_object o*) [Function]
Different macros that check whether *o* belongs to the specified type. These checks have been optimized, and are preferred over several calls to `ecl_t_of`.

`bool ECL_IMMEDIATE` (*cl_object o*) [Function]
Tells whether *x* is an immediate datatype.

4.4.2 Constructing objects

On each of the following sections we will document the standard interface for building objects of different types. For some objects, though, it is too difficult to make a C interface that resembles all of the functionality in the lisp environment. In those cases you need to

1. build the objects from their textual representation, or
2. use the evaluator to build these objects.

The first way makes use of a C or Lisp string to construct an object. The two functions you need to know are the following ones.

`si::string-to-object` *string* **&optional** (*err-value* **nil**) [Function]

`cl_object si_string_to_object` (*cl_narg narg, cl_object str, ...*) [Function]
`cl_object ecl_read_from_cstring` (*const char *s*) [Function]
`ecl_read_from_cstring` builds a lisp object from a C string which contains a suitable representation of a lisp object. `si_string_to_object` performs the same task, but uses a lisp string, and therefore it is less useful.

- **DEPRECATED** `c_string_to_object` – equivalent to `ecl_read_from_cstring`

Example

Using a C string

```
cl_object array1 = ecl_read_from_cstring("#(1 2 3 4)");
```

Using a Lisp string

```
cl_object string = make_simple_base_string("#(1 2 3 4)");
cl_object array2 = si_string_to_object(string);
```

Integers

Common-Lisp distinguishes two types of integer types: `bignum`s and `fixnum`s. A fixnum is a small integer, which ideally occupies only a word of memory and which is between the values `MOST-NEGATIVE-FIXNUM` and `MOST-POSITIVE-FIXNUM`. A `bignum` is any integer which is not a `fixnum` and it is only constrained by the amount of memory available to represent it.

ECL `integer` subtypes `fixnum` and `bignum` are defined as built-in classes, so it is possible to specialize to them generic functions.

In ECL a `fixnum` is an integer that, together with the tag bits, fits in a word of memory. The size of a word, and thus the size of a `fixnum`, varies from one architecture to another, and you should refer to the types and constants in the ecl.h header to make sure that your C extensions are portable. All other integers are stored as `bignum`s, they are not immediate objects, they take up a variable amount of memory and the GNU Multiprecision Library is required to create, manipulate and calculate with them.

`cl_fixnum` [C/C++ identifier]
: This is a C signed integer type capable of holding a whole `fixnum` without any loss of precision. The opposite is not true, and you may create a `cl_fixnum` which exceeds the limits of a fixnum and should be stored as a `bignum`.

`cl_index` [C/C++ identifier]
: This is a C unsigned integer type capable of holding a non-negative `fixnum` without loss of precision. Typically, a `cl_index` is used as an index into an array, or into a proper list, etc.

`MOST_NEGATIVE_FIXNUM` [Constant]
`MOST_POSITIVE_FIXNUM` [Constant]
: These constants mark the limits of a `fixnum`.

`bool ecl_fixnum_lower` (*cl_fixnum a, cl_fixnum b*) [Function]
`bool ecl_fixnum_greater` (*cl_fixnum a, cl_fixnum b*) [Function]
`bool ecl_fixnum_leq` (*cl_fixnum a, cl_fixnum b*) [Function]
`bool ecl_fixnum_geq` (*cl_fixnum a, cl_fixnum b*) [Function]
`bool ecl_fixnum_plusp` (*cl_fixnum a*) [Function]
`bool ecl_fixnum_minusp` (*cl_fixnum a*) [Function]
: Operations on `fixnums` (comparison and predicates).

`cl_object ecl_make_fixnum` (*cl_fixnum n*) [Function]
`cl_fixnum ecl_fixnum` (*cl_object o*) [Function]
: `ecl_make_fixnum` converts from an integer to a lisp object, while the `ecl_fixnum` does the opposite (converts lisp object fixnum to integer). These functions do **not** check their arguments.

  - **DEPRECATED** `MAKE_FIXNUM` – equivalent to `ecl_make_fixnum`
  - **DEPRECATED** `fix` – equivalent to `ecl_fixnum`

`cl_fixnum fixint` (*cl_object o*) [Function]
`cl_index fixnint` (*cl_object o*) [Function]
: Safe conversion of a lisp `fixnum` to a C integer of the appropriate size. Signals an error if *o* is not of fixnum type.

  `fixnint` additionally ensure that *o* is not negative.

Floats

In ECL `float` subtypes `single-float`, `double-float` and `long-float` (on supporting platforms) are defined as built-in classes, so it is possible to specialize to them generic functions.

Characters

ECL has two types of characters – one fits in the C type char, while the other is used when ECL is built with a configure option `--enable-unicode` which defaults to 32 (characters are stored in 32bit variable and codepoints have 21-bits).

`ecl_character` [C/C++ identifier]

Immediate type `t_character`. If ECL built with Unicode support, then may be either base or extended character, which may be distinguished with the predicate `ECL_BASE_CHAR_P`.

Additionally we have `ecl_base_char` for base strings, which is an equivalent to the ordinary char.

Example

```
if (ECL_CHARACTERP(o) && ECL_BASE_CHAR_P(o))
    printf("Base character: %c\n", ECL_CHAR_CODE(o));
```

`ECL_CHAR_CODE_LIMIT` [Constant]

Each character is assigned an integer code which ranges from 0 to (ECL_CHAR_CODE_LIMIT-1).

- **DEPRECATED** `CHAR_CODE_LIMIT` – equivalent to `ECL_CHAR_CODE_LIMIT`

`cl_object ECL_CODE_CHAR` (*ecl_character o*) [Function]
`ecl_character ECL_CHAR_CODE` (*cl_object o*) [Function]
`ecl_character ecl_char_code` (*cl_object o*) [Function]
`ecl_base_char ecl_base_char_code` (*cl_object o*) [Function]

`ECL_CHAR_CODE`, `ecl_char_code` and `ecl_base_char_code` return the integer code associated to a lisp character. `ecl_char_code` and `ecl_base_char_code` perform a safe conversion, while `ECL_CHAR_CODE` doesn't check its argument.

`ECL_CODE_CHAR` returns the lisp character associated to an integer code. It does not check its arguments.

- **DEPRECATED** `CHAR_CODE` – equivalent to `ECL_CHAR_CODE`
- **DEPRECATED** `CODE_CHAR` – equivalent to `ECL_CODE_CHAR`

`bool ecl_char_eq` (*cl_object x, cl_object y*) [Function]
`bool ecl_char_equal` (*cl_object x, cl_object y*) [Function]

Compare two characters for equality. char_eq take case into account and char_equal ignores it.

`int ecl_char_cmp` (*cl_object x, cl_object y*) [Function]
`int ecl_char_compare` (*cl_object x, cl_object y*) [Function]

Compare the relative order of two characters. `ecl_char_cmp` takes care of case and `ecl_char_compare` converts all characters to uppercase before comparing them.

Arrays

An array is an aggregate of data of a common type, which can be accessed with one or more non-negative indices. ECL stores arrays as a C structure with a pointer to the region of memory which contains the actual data. The cell of an array datatype varies depending on whether it is a vector, a bit-vector, a multidimensional array or a string.

`bool ECL_ADJUSTABLE_ARRAY_P` (*cl_object x*) [Function]
`bool ECL_ARRAY_HAS_FILL_POINTER_P` (*cl_object x*) [Function]

All arrays (arrays, strings and bit-vectors) may be tested for being adjustable and whenever they have a fill pointer with this two macros. They don't check the type of their arguments.

`ecl_vector` [C/C++ identifier]

If `x` contains a vector, you can access the following fields:

`x->vector.elttype`
: The type of the elements of the vector.

`x->vector.displaced`
: List storing the vectors that `x` is displaced from and that x displaces to.

`x->vector.dim`
: The maximum number of elements.

`x->vector.fillp`
: Actual number of elements in the vector or `fill pointer`.

`x->vector.self`
: Union of pointers of different types. You should choose the right pointer depending on *x->vector.elttype*.

`ecl_array` [C/C++ identifier]

If `x` contains a multidimensional array, you can access the following fields:

`x->array.elttype`
: The type of the elements of the array.

`x->array.rank`
: The number of array dimensions.

`x->array.displaced`
: List storing the arrays that `x` is displaced from and that x displaces to.

`x->array.dim`
: The maximum number of elements.

`x->array.dims[]`
: Array with the dimensions of the array. The elements range from `x->array.dim[0]` to `x->array.dim[x->array.rank-1]`.

`x->array.fillp`
: Actual number of elements in the array or `fill pointer`.

`x->array.self`
: Union of pointers of different types. You should choose the right pointer depending on *x->array.elttype*.

`cl_elttype` *ecl_aet_object ecl_aet_sf ecl_aet_df ecl_aet_lf ecl_aet_csf ecl_aet_cdf ecl_aet_clf ecl_aet_bit ecl_aet_fix ecl_aet_index ecl_aet_b8 ecl_aet_i8 ecl_aet_b16 ecl_aet_i16 ecl_aet_b32 ecl_aet_i32 ecl_aet_b64 ecl_aet_i64 ecl_aet_ch ecl_aet_bc* [C/C++ identifier]

Each array is of an specialized type which is the type of the elements of the array. ECL has arrays only a few following specialized types, and for each of these

types there is a C integer which is the corresponding value of `x->array.elttype` or `x->vector.elttype`. We list some of those types together with the C constant that denotes that type:

t `ecl_aet_object`

single-float
`ecl_aet_sf`

double-float
`ecl_aet_df`

long-float `ecl_aet_lf`

(COMPLEX SINGLE-FLOAT)
`ecl_aet_csf`

(COMPLEX DOUBLE-FLOAT)
`ecl_aet_cdf`

(COMPLEX LONG-FLOAT)
`ecl_aet_clf`

BIT `ecl_aet_bit`

FIXNUM `ecl_aet_fix`

INDEX `ecl_aet_index`

CHARACTER
`ecl_aet_ch`

BASE-CHAR
`ecl_aet_bc`

`cl_elttype ecl_array_elttype` (*cl_object array*) [Function]
Returns the element type of the array `o`, which can be a string, a bit-vector, vector, or a multidimensional array.

Example

For example, the code

```
ecl_array_elttype(ecl_read_from_cstring("\"AAA\""));  /* returns ecl_aet_ch */
ecl_array_elttype(ecl_read_from_cstring("#(A B C)")); /* returns ecl_aet_object *
```

`cl_object ecl_aref` (*cl_object x, cl_index index*) [Function]
`cl_object ecl_aset` (*cl_object x, cl_index index, cl_object value*) [Function]
These functions are used to retrieve and set the elements of an array. The elements are accessed with one index, index, as in the lisp function ROW-MAJOR-AREF.

Example

```
cl_object array = ecl_read_from_cstring("#2A((1 2) (3 4))");
cl_object x = ecl_aref(array, 3);
cl_print(1, x); /* Outputs 4 */
ecl_aset(array, 3, ecl_make_fixnum(5));
cl_print(1, array); /* Outputs #2A((1 2) (3 5)) */
```

`cl_object ecl_aref1` (*cl_object x, cl_index index*) [Function]
`cl_object ecl_aset1` (*cl_object x, cl_index index, cl_object value*) [Function]
These functions are similar to `aref` and `aset`, but they operate on vectors.

Example

```
cl_object array = ecl_read_from_cstring("#(1 2 3 4)");
cl_object x = ecl_aref1(array, 3);
cl_print(1, x);     /* Outputs 4 */
ecl_aset1(array, 3, ecl_make_fixnum(5));
cl_print(1, array); /* Outputs #(1 2 3 5) */
```

Strings

A string, both in Common-Lisp and in ECL is nothing but a vector of characters. Therefore, almost everything mentioned in the section of arrays remains valid here.

The only important difference is that ECL stores the base-strings (non-Unicode version of a string) as a lisp object with a pointer to a zero terminated C string. Thus, if a string has n characters, ECL will reserve n+1 bytes for the base-string. This allows us to pass the base-string self pointer to any C routine.

`ecl_string` [C/C++ identifier]
`ecl_base_string` [C/C++ identifier]
If `x` is a lisp object of type string or a base-string, we can access the following fields:

`x->string.dim` `x->base_string.dim`
: Actual number of characters in the string.

`x->string.fillp` `x->base_string.fillp`
: Actual number of characters in the string.

`x->string.self` `x->base_string.self`
: Pointer to the characters (appropriately `ecl_character`'s and `ecl_base_char`'s).

`bool ECL_EXTENDED_STRING_P` (*cl_object object*) [Function]
`bool ECL_BASE_STRING_P` (*cl_object object*) [Function]
Verifies if an objects is an extended or base string. If Unicode isn't supported, then `ECL_EXTENDED_STRING_P` always returns 0.

Bit-vectors

Bit-vector operations are implemented in file `src/c/array.d`. Bit-vector shares the structure with a vector, therefore, almost everything mentioned in the section of arrays remains valid here.

Streams

Streams implementation is a broad topic. Most of the implementation is done in the file `src/c/file.d`. Stream handling may have different implementations referred by a member pointer `ops`.

Additionally on top of that we have implemented *Gray Streams* (in portable Common Lisp) in file `src/clos/streams.lsp`, which may be somewhat slower (we need to benchmark it!).

This implementation is in a separate package *GRAY*. We may redefine functions in the *COMMON-LISP* package with a function `redefine-cl-functions` at run-time.

`ecl_file_ops` *write_* * *read_* * *unread_* * *peek_* * *listen* *clear_input clear_output finish_output force_output input_p output_p interactive_p element_type length get_position set_position column close* [C/C++ identifier]

`ecl_stream` [C/C++ identifier]

`ecl_smmode mode`
: Stream mode (in example `ecl_smm_string_input`).

`int closed`
: Whenever stream is closed or not.

`ecl_file_ops *ops`
: Pointer to the structure containing operation implementations (dispatch table).

`union file`
: Union of ANSI C streams (FILE *stream) and POSIX files interface (cl_fixnum descriptor).

`cl_object object0, object1`
: Some objects (may be used for a specific implementation purposes).

`cl_object byte_stack`
: Buffer for unread bytes.

`cl_index column`
: File column.

`cl_fixnum last_char`
: Last character read.

`cl_fixnum int0 int1`
: Some integers (may be used for a specific implementation purposes).

`cl_index byte_size`
: Size of byte in binary streams.

`cl_fixnum last_op`
: 0: unknown, 1: reading, -1: writing

`char *buffer`
: Buffer for FILE

`cl_object format`
: external format

`cl_eformat_encoder encoder`
`cl_eformat_encoder decoder`
`cl_object format_table`
`in flags` Character table, flags, etc

`ecl_character eof_character`

`bool ECL_ANSI_STREAM_P` (*cl_object o*) [Function]
Predicate determining if `o` is a first-class stream object.

`bool ECL_ANSI_STREAM_TYPE_P` (*cl_object o, ecl_smmode m*) [Function]
Predicate determining if `o` is a first-class stream object of type `m`.

Structures

Structures and instances share the same datatype `t_instance` (with a few exceptions. Structure implementation details are the file `src/c/structure.d`.

`cl_object ECL_STRUCT_TYPE` (*cl_object x*) [Function]
`cl_object ECL_STRUCT_SLOTS` (*cl_object x*) [Function]
`cl_object ECL_STRUCT_LENGTH` (*cl_object x*) [Function]
`cl_object ECL_STRUCT_SLOT` (*cl_object x, cl_index i*) [Function]
`cl_object ECL_STRUCT_NAME` (*cl_object x*) [Function]
Convenience functions for the structures.

Instances

`cl_object ECL_CLASS_OF` (*cl_object x*) [Function]
`cl_object ECL_SPEC_FLAG` (*cl_object x*) [Function]
`cl_object ECL_SPEC_OBJECT` (*cl_object x*) [Function]
`cl_object ECL_CLASS_NAME` (*cl_object x*) [Function]
`cl_object ECL_CLASS_SUPERIORS` (*cl_object x*) [Function]
`cl_object ECL_CLASS_INFERIORS` (*cl_object x*) [Function]
`cl_object ECL_CLASS_SLOTS` (*cl_object x*) [Function]
`cl_object ECL_CLASS_CPL` (*cl_object x*) [Function]
`bool ECL_INSTANCEP` (*cl_object x*) [Function]
Convenience functions for the structures.

Bytecodes

A bytecodes object is a lisp object with a piece of code that can be interpreted. The objects of type `t_bytecodes` are implicitly constructed by a call to eval, but can also be explicitly constructed with the `si_make_lambda` function.

`si:safe-eval` *form env* **&optional** *err-value* [Function]

`cl_object si_safe_eval` (*cl_narg narg, cl_object form, cl_object env, ...*) [Function]
`si_safe_eval` evaluates `form` in the lexical environment[1] `env`, which can be `ECL_NIL`. Before evaluating it, the expression form is bytecompiled. If the form signals an error, or tries to jump to an outer point, the function has two choices: by default, it will invoke a debugger, but if a third value is supplied, then `si_safe_eval` will not use a debugger but rather return that value.

- **DEPRECATED** `cl_object cl_eval (cl_object form)` - `cl_eval` is the equivalent of `si_safe_eval` but without environment and with no `err-value` supplied. It exists only for compatibility with previous versions.

[1] Note that `env` must be a lexical environment as used in the interpreter, See Section 4.6.3 [The lexical environment], page 177

- **DEPRECATED** `cl_object cl_safe_eval (cl_object form, cl_object env, cl_object err_value)` - Equivalent of `si_safe_eval`.

Example

```
cl_object form = ecl_read_from_cstring("(print 1)");
si_safe_eval(2, form, ECL_NIL);
si_safe_eval(3, form, ECL_NIL, ecl_make_fixnum(3)); /* on error function will ret
```

`cl_object si_make_lambda` (*cl_object name*, *cl_object def*) [Function]

Builds an interpreted lisp function with name given by the symbol name and body given by `def`.

Example

For instance, we would achieve the equivalent of

```
(funcall #'(lambda (x y)
             (block foo (+ x y)))
         1 2)
```

with the following code

```
cl_object def = ecl_read_from_cstring("((x y) (+ x y))");
cl_object name = ecl_make_symbol("FOO", "COMMON-LISP-USER");
cl_object fun = si_make_lambda(name, def);
return cl_funcall(3, fun, ecl_make_fixnum(1), ecl_make_fixnum(2));
```

Notice that `si_make_lambda` performs a bytecodes compilation of the definition and thus it may signal some errors. Such errors are not handled by the routine itself so you might consider using `si_safe_eval` instead.

4.5 Environment implementation

4.6 The interpreter

4.6.1 ECL stacks

ECL uses the following stacks:

Frame Stack	consisting of catch, block, tagbody frames
Bind Stack	for shallow binding of dynamic variables
Interpreter Stack	acts as a Forth data stack, keeping intermediate arguments to interpreted functions, plus a history of called functions.
C Control Stack	used for arguments/values passing, typed lexical variables, temporary values, and function invocation.

4.6.2 Procedure Call Conventions

ECL employs standard C calling conventions to achieve efficiency and interoperability with other languages. Each Lisp function is implemented as a C function which takes as many arguments as the Lisp original. If the function takes optional or keyword arguments, the corresponding C function takes one additional integer argument which holds the number of actual arguments. The function sets `nvalues` in the thread local environment to the

number of Lisp values produced, it returns the first one and the remaining ones are kept in a global (per thread) array (`values`).

To show the argument/value passing mechanism, here we list the actual code for the Common-Lisp function last.

```
cl_object
cl_last(cl_narg narg, cl_object l, ...)
{
        const cl_env_ptr the_env = ecl_process_env();
        cl_object k;
        va_list ARGS;
        va_start(ARGS, l);
        if (ecl_unlikely(narg < 1|| narg > 2)) FEwrong_num_arguments(/* ... */);
        if (narg > 1) {
                k = va_arg(ARGS,cl_object);
        } else {
                k = ecl_make_fixnum(1);
        }
        cl_object __value0 = ecl_last(l, ecl_to_size(k));
        the_env->nvalues = 1;
        the_env->values[0] = __value0;
        va_end(ARGS);
        return __value0;
}
```

ECL adopts the convention that the name of a function that implements a Common-Lisp function begins with a short package name (`cl` for `COMMON-LISP`, `si` for `SYSTEM`, etc), followed by `L`, and followed by the name of the Common-Lisp function. (Strictly speaking, '`-`' and '`*`' in the Common-Lisp function name are replaced by '`_`' and '`A`', respectively, to obey the syntax of C.)

The code for the function `last` first checks that the right number of arguments are supplied to `cl_last`. That is, it checks that `narg` is 1 or 2, and otherwise, it causes an error. Following that, the optional variable `k` is initialized and the return value `__value0` is computed. The number assigned to `nvalues` set by the function (1 in this case) represents the number of values of the function. The return value of the function is copied in the `values` array as well as returned directly.

In general, if one is to play with the C kernel of ECL there is no need to know about all these conventions. There is a preprocessor (see Section 4.3 [Defun preprocessor], page 163) that takes care of the details, by using a lisp representation of the statements that output values, and of the function definitions. For instance, the actual source code for `cl_last` in `src/c/list.d` is

```
@(defun last (l &optional (k ecl_make_fixnum(1)))
@
  @(return ecl_last(l, ecl_to_size(k)));
@)
```

4.6.3 The lexical environment

The ECL interpreter uses a list containing local functions and macros, variables, tags and blocks to represent the lexical environment. When a function closure is created, the current lexical environment is saved in the closure along with the lambda expression. Later, when the closure is invoked, this list is used to recover the lexical environment.

Note that this list is different from what the Common Lisp standard calls a lexical environment, which is the content of a `&environment` parameter to `defmacro`. For the differences between this two environments see the comments in `src/c/compiler.d` and `src/c/interpreter.d`.

4.6.4 The interpreter stack

The bytecodes interpreter uses a stack of its own to save and restore values from intermediate calculations. This Forth-like data stack is also used in other parts of the C kernel for various purposes, such as saving compiled code, keeping arguments to `format`, etc.

However, one of the most important roles of the Interpreter Stack is to keep a log of the functions which are called during the execution of bytecodes. For each function invoked, the interpreter keeps three lisp objects on the stack:

```
+----------+---------------------------------------------+
| function | lexical environment | index to previous record |
+----------+---------------------+--------------------------+
```

The first item is the object which is funcalled. It can be a bytecodes object, a compiled function or a generic function. In the last two cases the lexical environment is just `nil`. In the first case, the second item on the stack is the lexical environment on which the code is executed. Each of these records are popped out of the stack after function invocation.

Let us see how these invocation records are used for debugging.

```
> (defun fact (x)               ;;;  Wrong definition of the
    (if (= x 0)                 ;;;  factorial function.
        one                     ;;;  one  should be  1.
        (* x (fact (1- x)))))
FACT

> (fact 3)                      ;;;  Tries  3!
Error: The variable ONE is unbound.
Error signalled by IF.
Broken at IF.
>> :b                           ;;;  Backtrace.
Backtrace: eval > fact > if > fact > if > fact > if > fact > IF
;;;  Currently at the last  IF.
>> :h                           ;;;  Help.

Break commands:
:q(uit)         Return to some previous break level.
:pop            Pop to previous break level.
:c(ontinue)     Continue execution.
:b(acktrace)    Print backtrace.
```

```
:f(unction)     Show current function.
:p(revious)     Go to previous function.
:n(ext)         Go to next function.
:g(o)           Go to next function.
:fs             Search forward for function.
:bs             Search backward for function.
:v(ariables)    Show local variables, functions, blocks, and tags.
:l(ocal)        Return the nth local value on the stack.
:hide           Hide function.
:unhide         Unhide function.
:hp             Hide package.
:unhp           Unhide package.
:unhide-all     Unhide all variables and packages.
:bds            Show binding stack.
:m(essage)      Show error message.
:hs             Help stack.
Top level commands:
:cf             Compile file.
:exit or ^D     Exit Lisp.
:ld             Load file.
:step           Single step form.
:tr(ace)        Trace function.
:untr(ace)      Untrace function.

Help commands:
:apropos        Apropos.
:doc(ument)     Document.
:h(elp) or ?    Help.  Type ":help help" for more information.

>> :p                       ;;;  Move to the last call of  FACT.
Broken at IF.

>> :b
Backtrace: eval > fact > if > fact > if > fact > if > FACT > if
;;;  Now at the last  FACT.
>> :v                       ;;;  The environment at the last call
Local variables:            ;;;  to  FACT  is recovered.
  X: 0                      ;;;  X  is the only bound variable.
Block names: FACT.          ;;;  The block  FACT  is established.

>> x
0                           ;;;  The value of  x  is  0.

>>(return-from fact 1)      ;;;  Return from the last call of
6                           ;;;  FACT  with the value of  0.
                            ;;;  The execution is resumed and
>                           ;;;  the value  6  is returned.
```

```
;;;  Again at the top-level loop.
```

4.7 The compiler

4.7.1 The compiler translates to C

The ECL compiler is essentially a translator from Common-Lisp to C. Given a Lisp source file, the compiler first generates three intermediate files:

- a C-file which consists of the C version of the Lisp program
- an H-file which consists of declarations referenced in the C-file
- a Data-file which consists of Lisp data to be used at load time

The ECL compiler then invokes the C compiler to compile the C-file into an object file. Finally, the contents of the Data-file is appended to the object file to make a *Fasl-file*. The generated Fasl-file can be loaded into the ECL system by the Common-Lisp function `load`. By default, the three intermediate files are deleted after the compilation, but, if asked, the compiler leaves them.

The merits of the use of C as the intermediate language are:

- The ECL compiler is highly portable.
- Cross compilation is possible, because the contents of the intermediate files are common to all versions of ECL. For example, one can compile his or her Lisp program by the ECL compiler on a Sun, bring the intermediate files to DOS, compile the C-file with the gcc compiler under DOS, and then append the Data-file to the object file. This procedure generates the Fasl-file for the ECL system on DOS. This kind of cross compilation makes it easier to port ECL.
- Hardware-dependent optimizations such as register allocations are done by the C compiler.

The demerits are:

- At those sites where no C compiler is available, the users cannot compile their Lisp programs.
- The compilation time is long. 70% to 80% of the compilation time is used by the C compiler. The ECL compiler is therefore slower than compiler generating machine code directly.

4.7.2 The compiler mimics human C programmer

The format of the intermediate C code generated by the ECL compiler is the same as the hand-coded C code of the ECL source programs. For example, supposing that the Lisp source file contains the following function definition:

```
(defvar *delta* 2)
(defun add1 (x) (+ *delta* x))
```

The compiler generates the following intermediate C code.

```
/*      function definition for ADD1                                */
/*      optimize speed 3, debug 0, space 0, safety 2                */
static cl_object L1add1(cl_object v1x)
{
```

```
 cl_object env0 = ECL_NIL;
 const cl_env_ptr cl_env_copy = ecl_process_env();
 cl_object value0;
 ecl_cs_check(cl_env_copy,value0);
 {
TTL:
  value0 = ecl_plus(ecl_symbol_value(VV[0]),v1x);
  cl_env_copy->nvalues = 1;
  return value0;
 }
}

/*      initialization of this module                                   */
ECL_DLLEXPORT void init_fas_CODE(cl_object flag)
{
 const cl_env_ptr cl_env_copy = ecl_process_env();
 cl_object value0;
 cl_object *VVtemp;
 if (flag != OBJNULL){
 Cblock = flag;
 #ifndef ECL_DYNAMIC_VV
 flag->cblock.data = VV;
 #endif
 flag->cblock.data_size = VM;
 flag->cblock.temp_data_size = VMtemp;
 flag->cblock.data_text = compiler_data_text;
 flag->cblock.cfuns_size = compiler_cfuns_size;
 flag->cblock.cfuns = compiler_cfuns;
 flag->cblock.source = make_constant_base_string("test.lisp");
 return;}
 #ifdef ECL_DYNAMIC_VV
 VV = Cblock->cblock.data;
 #endif
 Cblock->cblock.data_text = (const cl_object *)"@EcLtAg:init_fas_CODE@";
 VVtemp = Cblock->cblock.temp_data;
 ECL_DEFINE_SETF_FUNCTIONS
  si_Xmake_special(VV[0]);
  if (ecl_boundp(cl_env_copy,VV[0])) { goto L2; }
  cl_set(VV[0],ecl_make_fixnum(2));
L2:;
  ecl_cmp_defun(VV[2]);                          /*  ADD1            */
}
```

The C function `L1add1` implements the Lisp function `add1`. This relation is established by `ecl_cmp_defun` in the initialization function `init_fas_CODE`, which is invoked at load time. There, the vector `VV` consists of Lisp objects; `VV[0]` and `VV[1]` in this example hold the Lisp symbols `*delta*` and `add1`, while `VV[2]` holds the function object for `add1`, which

is created during initialization of the module. `VM` in the definition of `L1add1` is a C macro declared in the corresponding H-file. The actual value of `VM` is the number of value stack locations used by this module, i.e., 3 in this example. Thus the following macro definition is found in the H-file.

```
#define VM 3
```

4.7.3 Implementation of Compiled Closures

The ECL compiler takes two passes before it invokes the C compiler. The major role of the first pass is to detect function closures and to detect, for each function closure, those lexical objects (i.e., lexical variable, local function definitions, tags, and block-names) to be enclosed within the closure. This check must be done before the C code generation in the second pass, because lexical objects to be enclosed in function closures are treated in a different way from those not enclosed.

Ordinarily, lexical variables in a compiled function `f` are allocated on the C stack. However, if a lexical variable is to be enclosed in function closures, it is allocated on a list, called the "environment list", which is local to `f`. In addition, a local variable is created which points to the lexical variable's location (within the environment list), so that the variable may be accessed through an indirection rather than by list traversal.

The environment list is a pushdown list: It is empty when `f` is called. An element is pushed on the environment list when a variable to be enclosed in closures is bound, and is popped when the binding is no more in effect. That is, at any moment during execution of `f`, the environment list contains those lexical variables whose binding is still in effect and which should be enclosed in closures. When a compiled closure is created during execution of `f`, the compiled code for the closure is coupled with the environment list at that moment to form the compiled closure.

Later, when the compiled closure is invoked, a pointer is set up to each lexical variable in the environment list, so that each object may be referenced through a memory indirection.

Let us see an example. Suppose the following function has been compiled.

```
(defun foo (x)
  (let ((a #'(lambda () (incf x)))
        (y x))
    (values a #'(lambda () (incf x y)))))
```

`foo` returns two compiled closures. The first closure increments x by one, whereas the second closure increments x by the initial value of x. Both closures return the incremented value of x.

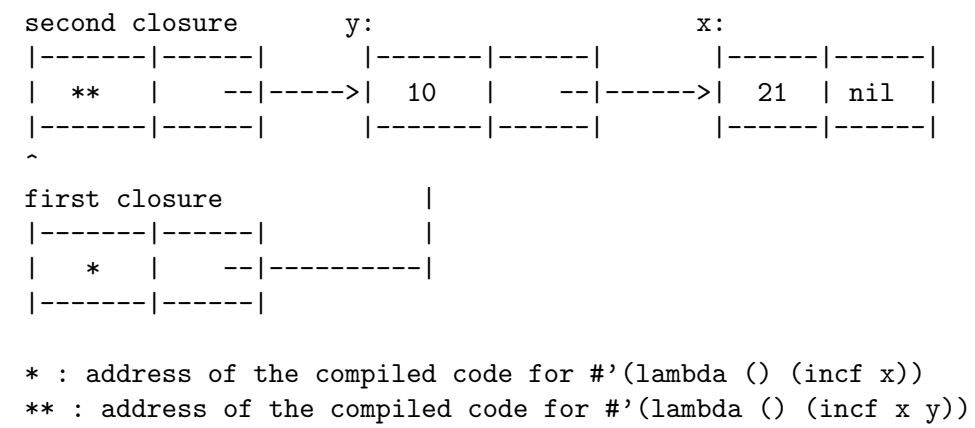
```
>(multiple-value-setq (f g) (foo 10))
#<compiled-closure nil>

>(funcall f)
11

>(funcall g)
21

>
```

After this, the two compiled closures look like:




```
* : address of the compiled code for #'(lambda () (incf x))
** : address of the compiled code for #'(lambda () (incf x y))
```

4.7.4 Use of Declarations to Improve Efficiency

Declarations, especially type and function declarations, increase the efficiency of the compiled code. For example, for the following Lisp source file, with two Common-Lisp declarations added,

```
(eval-when (:compile-toplevel)
  (proclaim '(ftype (function (fixnum fixnum) fixnum) tak))
  (proclaim '(optimize (speed 3) (debug 0) (safety 0))))

(defun tak (x y)
  (declare (fixnum x y))
  (if (not (< y x))
    y
    (tak (tak (1- x) y)
         (tak (1- y) x))))
```

The compiler generates the following C code (Note that the tail-recursive call of `tak` was replaced by iteration):

```
/*      function definition for TAK                                     */
/*      optimize speed 3, debug 0, space 0, safety 0                    */
static cl_object L1tak(cl_object v1x, cl_object v2y)
{
 cl_object env0 = ECL_NIL;
 const cl_env_ptr cl_env_copy = ecl_process_env();
 cl_object value0;
 cl_fixnum v3x;
 cl_fixnum v4y;
 v3x = ecl_fixnum(v1x);
 v4y = ecl_fixnum(v2y);
TTL:
 if ((v4y)<(v3x)) { goto L1; }
 value0 = ecl_make_fixnum(v4y);
 cl_env_copy->nvalues = 1;
 return value0;
```

```
L1:;
 {
  cl_fixnum v5;
  {
   cl_fixnum v6;
   v6 = (v3x)-1;
   v5 = ecl_fixnum(L1tak(ecl_make_fixnum(v6), ecl_make_fixnum(v4y)));
  }
  {
   cl_fixnum v6;
   v6 = (v4y)-1;
   v4y = ecl_fixnum(L1tak(ecl_make_fixnum(v6), ecl_make_fixnum(v3x)));
  }
  v3x = v5;
 }
 goto TTL;
}
```

4.7.5 Inspecting generated C code

Common-Lisp defines a function disassemble, which is supposed to disassemble a compiled function and to display the assembler code. According to *Common-Lisp: The Language*,

> This is primary useful for debugging the compiler, ..\\

This is, however, *useless* in our case, because we are not concerned with assembly language. Rather, we are interested in the C code generated by the ECL compiler. Thus the disassemble function in ECL accepts not-yet-compiled functions only and displays the translated C code.

```
> (defun add1 (x) (1+ x))

ADD1
> (disassemble *)

/*      function definition for ADD1                                    */
/*      optimize speed 3, debug 0, space 0, safety 2                    */
static cl_object L1add1(cl_object v1x)
{
 cl_object env0 = ECL_NIL;
 const cl_env_ptr cl_env_copy = ecl_process_env();
 cl_object value0;
 ecl_cs_check(cl_env_copy,value0);
 {
TTL:
  value0 = ecl_one_plus(v1x);
  cl_env_copy->nvalues = 1;
  return value0;
 }
}
```

4.8 Porting ECL

To port ECL to a new architecture, the following steps are required:

1. Ensure that the GNU Multiprecision library supports this machine.
2. Ensure that the Boehm-Weiser garbage collector is supported by that architecture. Alternatively, port ECL's own garbage collector `src/c/alloc.d` and `src/c/gbc.d` to that platform.
3. Fix `src/aclocal.in`, `src/h/config.h.in` and `src/h/ecl.h` so that they supply flags for the new host machine.
4. Fix the machine dependent code in `src/c/`. The most critical parts are in the `unix*.d` and `thread*.d` files.
5. Compile as in any other platform.
6. Run the tests and compare to the results of other platforms.

4.9 Removed features

In-house DFFI

Commit `10bd3b613fd389da7640902c2b88a6e36088c920`. Native DFFI was replaced by a libffi (`https://sourceware.org/libffi/`) long time ago, but we have maintained the code as a fallback. Due to small number of supported platforms and no real use it has been removed in 2016.

In-house GC

Commit `61500316b7ea17d0e42f5ca127f2f9fa3e6596a8`. Broken GC is replaced by BoehmGC library. This may be added back as a fallback in the near future.

`3bd9799a2fef21cc309472e604a46be236b155c7` removes a leftover (apparently gbc.d wasn't bdwgc glue).

Green threads

Commit `41923d5927f31f4dd702f546b9caee74e98a2080`. Green threads (aka light weight processes) has been replaced with native threads implementation. There is an ongoing effort to bring them back as an alternative interface.

Compiler newcmp

Commit `9b8258388487df8243e2ced9c784e569c0b34c4f` This was abandoned effort of changing the compiler architecture. Some clever ideas and a compiler package hierarchy. Some of these things should be incorporated during the evolution of the primary compiler.

Old MIT loop

Commit `5042589043a7be853b7f85fd7a996747412de6b4`. This old loop implementation has got superseded by the one incorporated from Symbolics LOOP in 2001.

Support for bignum arithmetic (earith.d)

Commit `edfc2ba785d6a64667e89c869ef0a872d7b9704b`. Removes pre-gmp bignum code. Name comes probably from "extended arithmetic", contains multiplication and division routines (assembler and a portable implementation).

Unification module

Commit `6ff5d20417a21a76846c4b28e532aac097f03109`. Old unification module (logic programming) from EcoLisp times.

Hierarchical packages

Commit `72e422f1b3c4b3c52fa273b961517db943749a8f`. Partially broken. Tests left in package-extensions.lsp.

8-bit opcodes in bytecodes interpreter

Commit `c3244b0148ed352808779b07b25c3edddf9d7349`. Works fine but provides no real gain and is limited to intel.

thread local variables

Commit `618f6e92e8144f7b95bc36b42a337c212bacc8e7`. Disabled by default, practically not tested, works on limited number of platforms.

Indexes

Concept index

(

(complex float) internal representation 42

A

ANSI Dictionary 25
Arrays 57

B

Barriers (synchronization) 139
Bytecodes eager compilation 28

C

C/C++ code inlining 111
Command line processing 101
Common Lisp functions limits 29
Compiler declarations 25
Creating executables and libraries 89
cstring and foreign string differences 124

D

Defun preprocessor 163
`disassemble` and `compile` on defined functions 29
Dynamic foreign function interface 115

E

Eager compilation implications 28
Environment implementation 175
External processes 103

F

FIFO files (named pipes) 105
Foreign aggregate types 117
Foreign function interface 106
Foreign functions and libraries 128
Foreign objects 120
Foreign primitive types 115
Foreign strings 124

H

Hash table generic test 69
Hash table serialization 68

L

long-float internal representation 42

M

Memory management 147

N

Native FASL 91
Native threads 129

O

Object file internal layout 92
One type for everything: `cl_object` 23
Only in Common Lisp 25

P

Package local nicknames 154
Package locks 155
Parsing arguments in standalone executable ... 102
Portable FASL 90

R

Readers-writer locks 135

S

Semaphores (synchronization) 137
Shadowed bindings in let, flet, labels and lambda-list 28
Shared-exclusive locks 135
Static foreign function interface 111
Synchronized hash tables 68
System building 89

T

Thread-safe hash tables 68
Two kinds of FFI 106

U

Universal foreign function interface 115

W

Weak hash tables 68

Configure option index

`--enable-c99complex [auto]` 45
`--enable-shared [YES|no]` 91
`--enable-small-cons [YES|no]` 165
`--enable-threads [yes|no|AUTO]` 129
`--enable-unicode [32|16|no]` 50, 168
`--with-dffi [system|included|AUTO|no]` 106
`--with-fpe [YES|no]` 42
`--with-ieee-fp [YES|no]` 42
`--with-libffi-prefix=path` 106

Feature index

C

`COMPLEX-FLOAT` 42, 115

D

`DFFI` 107
`DLOPEN` 91

E

`ecl-read-write-lock` 135
`ECL-WEAK-HASH` 68

F

`FFI` 106

L

`LONG-FLOAT` 42, 115
`LONG-LONG` 115

P

`PACKAGE-LOCAL-NICKNAMES` 154
`PACKAGE-LOCKS` 155

T

`THREADS` 129

U

`UINT16-T` 115
`UINT32-T` 115
`UINT64-T` 115

Example index

A

Accessing underlying `cl_object` structure 166
Atomic update of a structure slot 141

B

Building executable 93
Building native FASL 91
Building Portable FASL file 90
Building shared library 93
Building static library 92

C

CFFI usage 110
`cl_object` checking the type with `ecl_t_of` ... 165
Conversion between foreign
string and cstring 125
cstring used to send and return a value 124

D

Define a compare-and-swap expansion 142
Defpackage `:lock` option 156
defpackage and package local nicknames 154
distinguishing between base and
Unicode character 169
dpp: function definition 163

E

Eager compilation impact on macros 28
`ecl_aref` and `ecl_aset` accessing arrays 171
`ecl_aref1` and `ecl_aset1` accessing vectors ... 172
`ecl_array_elttype` different types of objects 171
`ecl_read_from_cstring` constructing Lisp objects in C 167
`ext:with-backend` use different code for c and bytecodes compiler 114

F

`ffi:allocate-foreign-object` allocating structure object 120
`ffi:c-inline` inlining c code 112
`ffi:c-inline` returning multiple values 113
`ffi:c-progn` interleaving c and lisp code 113
`ffi:clines` adding c toplevel declarations 111
`ffi:def-array-pointer` usage 119
`ffi:def-constant` defining constants 116
`ffi:def-enum` sample enumerations 118
`ffi:def-foreign-type` examples 117
`ffi:def-foreign-var` places in foreign world 123
`ffi:def-function` 128
`ffi:def-struct` defining C structure 118
`ffi:def-union` union definition and usage 119
`ffi:deref-array` retrieving array element 119
`ffi:deref-pointer` 121
`ffi:ensure-char-character` 122
`ffi:ensure-char-integer` 122
`ffi:find-foreign-library` 129
`ffi:get-slot-value` manipulating a struct field 118
`ffi:get-slot-value` usage 119
`ffi:load-foreign-library` 129
`ffi:null-char-p` example 117
`ffi:size-of-foreign-type` 121
`ffi:with-cast-pointer` 123
`ffi:with-foreign-object` macro usage 121
foreign string used to send and return a value 124

H

Hash table extensions example 69

I

Initializing static/shared library in C/C++ 92

K

Keeping lambda definitions with `si:*keep-definitions*` 29
Killing process 132

L

LS implementation 102

M

mp:process-run-function usage 133

P

Possible implementation of `mp:process-run-function`: 131
Process interruption 131

S

Safely executing Lisp code with floating point exceptions in embedding program 21
Setting a signal handler 147
SFFI usage 111
`si::make-lambda` usage (bytecodes compilation) 28
`si_make_lambda` building functions 175
`si_safe_eval` 175
Suspend and resume process 132

T

`trace` usage 86

U

UFFI usage 109
Using sequence streams 78

W

`with-cstring` 126

Function index

*

*ecl_base_string_pointer_safe(cl_object 109
*ecl_foreign_data_pointer_safe(cl_object 108

_

_ecl_caar 53
_ecl_cadr 53
_ecl_car 53
_ecl_cdr 53

C

cl:defpackage 154, 156
cl_boot 16
cl_shutdown 16
close 152
compile-file 83
compile-file-pathname 84

D

directory 72
disassemble 84

E

ecl_aet_to_symbol 58
ecl_alloc_adjustable_base_string 63
ecl_alloc_simple_base_string 63
ecl_alloc_simple_vector 59
ecl_alpha_char_p 52
ecl_alphanumericp 52
ecl_aref 60, 171
ecl_aref1 ..... 60, 172
ecl_array_dimension 61
ecl_array_elttype 61, 171
ecl_array_rank 61
ecl_aset 60, 171
ecl_aset1 ..... 60, 172
ecl_atomic_get 140
ecl_atomic_incf 140
ecl_atomic_incf_by_fixnum 140
ecl_atomic_index_incf 140
ecl_atomic_pop 140
ecl_atomic_push 140
ecl_base_char_code 52, 169
ecl_base_char_p 52
ecl_bds_bind 30
ecl_bds_push 30
ecl_bds_unwind_n 31
ecl_bds_unwind1 31
ecl_both_case_p 52
ecl_cdfloat 46
ecl_char 64
ecl_char_cmp 169
ecl_char_code 52, 169
ecl_char_compare 169
ecl_char_downcase 52
ecl_char_eq 169
ecl_char_equal 169
ecl_char_set 64
ecl_char_upcase 52
ecl_clear_interrupts 21
ecl_clfloat 46
ecl_compare_and_swap 139
ecl_csfloat 46
ecl_decode_from_cstring 64
ecl_decode_from_unicode_wstring 65
ecl_digitp 52
ecl_disable_interrupts 21
ecl_double_float 46
ecl_enable_interrupts 21
ecl_encode_to_cstring 65
ecl_encode_to_unicode_wstring 65
ecl_fixnum 46, 168
ecl_fixnum_geq 168
ecl_fixnum_greater 168
ecl_fixnum_leq 168
ecl_fixnum_lower 168
ecl_fixnum_minusp 168
ecl_fixnum_plusp 168
ecl_get_option 19
ecl_graphic_char_p 52
ecl_import_current_thread 19
ecl_long_float 46
ecl_lower_case_p 52
ecl_make_barrier 139
ecl_make_cdfloat 45
ecl_make_clfloat 45
ecl_make_complex 46
ecl_make_constant_base_string 63
ecl_make_csfloat 45
ecl_make_double_float 45
ecl_make_fixnum 45, 168
ecl_make_foreign_data 108
ecl_make_int 45
ecl_make_int16_t 45
ecl_make_int32_t 45
ecl_make_int64_t 45
ecl_make_int8_t 45
ecl_make_integer 45
ecl_make_keyword 38
ecl_make_lock 134
ecl_make_long 46
ecl_make_long_float 45
ecl_make_long_long 46
ecl_make_ratio 46
ecl_make_rwlock 136
ecl_make_semaphore 138
ecl_make_short_t 45
ecl_make_simple_base_string 63
ecl_make_single_float 45
ecl_make_symbol 39

ecl_make_uint 46
ecl_make_uint16_t 45
ecl_make_uint32_t 45
ecl_make_uint64_t 45
ecl_make_uint8_t 45
ecl_make_ulong 46
ecl_make_ulong_long 46
ecl_make_unsigned_integer 45
ecl_make_ushort_t 45
ecl_nth_value 32
ecl_null_terminated_base_
  string(cl_object 109
ecl_nvalues 32
ecl_process_env 27
ecl_read_from_cstring 167
ecl_release_current_thread 19
ecl_return0 33
ecl_return1 33
ecl_return2 33
ecl_return3 33
ecl_set_option 16
ecl_setq 31
ecl_single_float 46
ecl_standard_char_p 52
ecl_symbol_to_aet 58
ecl_symbol_value 31
ecl_t_of 166
ecl_to_cdfloat 47
ecl_to_clfloat 47
ecl_to_csfloat 47
ecl_to_double 47
ecl_to_fixnum 46
ecl_to_float 46
ecl_to_int 47
ecl_to_int16_t 47
ecl_to_int32_t 47
ecl_to_int64_t 47
ecl_to_int8_t 47
ecl_to_long 47
ecl_to_long_double 47
ecl_to_long_long 47
ecl_to_short 47
ecl_to_uint 47
ecl_to_uint16_t 47
ecl_to_uint32_t 47
ecl_to_uint64_t 47
ecl_to_uint8_t 47
ecl_to_ulong 47
ecl_to_ulong_long 47
ecl_to_unsigned_integer 46
ecl_to_ushort 47
ecl_upper_case_p 52
ecl_va_arg 31
ecl_va_end 31
ecl_va_start 31
ECL_ADJUSTABLE_ARRAY_P 170
ECL_ANSI_STREAM_P 174
ECL_ANSI_STREAM_TYPE_P 174
ECL_ARRAY_HAS_FILL_POINTER_P 170
ECL_ARRAYP 166
ECL_ATOM 166
ECL_BASE_CHAR_CODE_P 166
ECL_BASE_CHAR_P 166
ECL_BASE_STRING_P 172
ECL_BIGNUMP 166
ECL_BIT_VECTOR_P 166
ECL_BLOCK_BEGIN 33
ECL_CATCH_ALL 19
ECL_CATCH_BEGIN 33
ECL_CHAR_CODE 52, 169
ECL_CHARACTERP 166
ECL_CLASS_CPL 174
ECL_CLASS_INFERIORS 174
ECL_CLASS_NAME 174
ECL_CLASS_OF 174
ECL_CLASS_SLOTS 174
ECL_CLASS_SUPERIORS 174
ECL_CODE_CHAR 52, 169
ECL_COMPLEXP 166
ECL_CONS_CAR 53
ECL_CONS_CDR 53
ECL_CONSP 166
ECL_DOUBLE_FLOAT_P 166
ECL_EXTENDED_STRING_P 172
ECL_FIXNUMP 166
ECL_FOREIGN_DATA_P 166
ECL_HANDLER_CASE 36
ECL_HASH_TABLE_P 166
ECL_IMMEDIATE 167
ECL_INSTANCEP 174
ECL_LISTP 166
ECL_LONG_FLOAT_P 166
ECL_NUMBER_TYPE_P 166
ECL_PACKAGEP 166
ECL_PATHNAMEP 166
ECL_RANDOM_STATE_P 166
ECL_READTABLEP 166
ECL_REAL_TYPE_P 166
ECL_RESTART_CASE 37
ECL_RPLACA 53
ECL_RPLACD 53
ECL_SINGLE_FLOAT_P 166
ECL_SPEC_FLAG 174
ECL_SPEC_OBJECT 174
ECL_SSE_PACK_P 166
ECL_STRINGP 166
ECL_STRUCT_LENGTH 174
ECL_STRUCT_NAME 174
ECL_STRUCT_SLOT 174
ECL_STRUCT_SLOTS 174
ECL_STRUCT_TYPE 174
ECL_SYMBOLP 166
ECL_UNWIND_PROTECT 20
ECL_UNWIND_PROTECT_BEGIN 33
ECL_VECTORP 166
ECL_WITH_LISP_FPE 21

ed 85
ext:add-package-local-nickname 155
ext:all-encodings 76
ext:catch-signal 147
ext:character-coding-error-external-format 77
ext:character-decoding-error-octects 77
ext:character-encoding-error-code 77
ext:chdir 105
ext:chmod 105
ext:command-args 101
ext:copy-file 105
ext:decoding-error 78
ext:encoding-error 78
ext:environ 105
ext:external-process-error-stream 103
ext:external-process-input 103
ext:external-process-output 103
ext:external-process-pid 103
ext:external-process-status 103
ext:external-process-wait 103
ext:file-kind 105
ext:file-stream-fd 76
ext:float-infinity-p 44
ext:float-nan-p 44
ext:get-finalizer 150
ext:get-limit 151
ext:get-signal-handler 146
ext:getcwd 105
ext:getenv 105
ext:getpid 105
ext:getuid 105
ext:hash-table-content 68
ext:hash-table-fill 68
ext:hash-table-synchronized-p 68
ext:hash-table-weakness 68
ext:lock-package 156
ext:make-pipe 105
ext:make-sequence-input-stream 78
ext:make-sequence-output-stream 78
ext:nan 44
ext:octets-to-string 64
ext:package-local-nicknames 154
ext:package-locally-nicknamed-by-list 154
ext:package-locked-p 156
ext:peek-byte 76
ext:process-command-args 102
ext:quit 105
ext:remove-package-local-nickname 155
ext:run-program 104
ext:set-buffering-mode 76
ext:set-finalizer 151
ext:set-limit 151
ext:set-signal-handler 146
ext:setenv 105
ext:stack-overflow-size 150
ext:stack-overflow-type 150
ext:string-to-octets 64
ext:system 105
ext:terminate-process 103
ext:trap-fpe 42
ext:unix-signal-received-code 146
ext:unlock-package 156
ext:unread-byte 76
ext:with-backend 114
ext:with-unlocked-packages 156
ext:without-package-locks 156

F

ffi:allocate-foreign-object 120
ffi:allocate-foreign-string 127
ffi:c-inline 112
ffi:c-progn 113
ffi:clines 111
ffi:convert-from-cstring 125, 126
ffi:convert-from-foreign-string 126
ffi:convert-to-cstring 125
ffi:convert-to-foreign-string 127
ffi:def-array-pointer 119
ffi:def-constant 116
ffi:def-enum 117
ffi:def-foreign-type 116
ffi:def-foreign-var 123
ffi:def-function 128
ffi:def-struct 118
ffi:def-union 119
ffi:defcallback 114
ffi:defcbody 114
ffi:defentry 114
ffi:defla 115
ffi:deref-array 119
ffi:deref-pointer 121
ffi:ensure-char-character 122
ffi:ensure-char-integer 122
ffi:find-foreign-library 129
ffi:free-foreign-object 120
ffi:get-slot-pointer 118
ffi:get-slot-value 118
ffi:load-foreign-library 128
ffi:make-null-pointer 122
ffi:null-char-p 117
ffi:null-pointer-p 122
ffi:pointer-address 121
ffi:size-of-foreign-type 121
ffi:with-cast-pointer 123
ffi:with-cstring 126
ffi:with-cstrings 126
ffi:with-foreign-object 120
ffi:with-foreign-string 127
ffi:with-foreign-strings 127
fixint 168
fixnint 168

M

mp:all-processes 130
mp:atomic-decf 140
mp:atomic-incf 140
mp:atomic-pop 141
mp:atomic-push 141
mp:atomic-update 141
mp:barrier-arrivers-count 139
mp:barrier-count 139
mp:barrier-name 139
mp:barrier-unblock 139
mp:barrier-wait 139
mp:block-signals 133
mp:compare-and-swap 140
mp:condition-variable-broadcast 137
mp:condition-variable-signal 137
mp:condition-variable-timedwait 137
mp:condition-variable-wait 137
mp:defcas 142
mp:define-cas-expander 141
mp:exit-process 130
mp:get-cas-expansion 143
mp:get-lock 135
mp:get-rwlock-read 136
mp:get-rwlock-write 136
mp:giveup-lock 135
mp:giveup-rwlock-read 136
mp:giveup-rwlock-write 136
mp:holding-lock-p 135
mp:interrupt-process 130
mp:lock-count 135
mp:lock-owner 135
mp:lock_name 135
mp:make-barrier 139
mp:make-condition-variable 137
mp:make-lock 134
mp:make-process 131
mp:make-rwlock 136
mp:make-semaphore 138
mp:process-active-p 131
mp:process-enable 131
mp:process-join 132
mp:process-kill 132
mp:process-name 132
mp:process-preset 133
mp:process-resume 132
mp:process-run-function 133
mp:process-suspend 132
mp:process-yield 131
mp:recursive-lock-p 135
mp:remcas 143
mp:restore-signals 133
mp:rwlock-name 136
mp:semaphore-count 138
mp:semaphore-name 138
mp:semaphore-wait 138
mp:semaphore-wait-count 138
mp:signal-semaphore 138
mp:try-get-semaphore 138
mp:wait-on-semaphore 138
mp:with-interrupts 134, 146
mp:with-lock 135
mp:with-rwlock 136
mp:without-interrupts 133, 146
mp_all_processes 130
mp_block_signals 133
mp_condition_variable_broadcast 137
mp_condition_variable_signal 137
mp_condition_variable_timedwait 137
mp_condition_variable_wait 137
mp_current_process 133
mp_exit_process 130
mp_get_lock_nowait 135
mp_get_lock_wait 135
mp_get_rwlock_read_nowait 136
mp_get_rwlock_read_wait 136
mp_get_rwlock_write_nowait 136
mp_get_rwlock_write_wait 136
mp_giveup_lock 135
mp_giveup_rwlock_read 136
mp_giveup_rwlock_write 136
mp_holding_lock_p 135
mp_interrupt_process 130
mp_lock_count 135
mp_lock_name 135
mp_lock_owner 135
mp_make_condition_variable 137
mp_make_process 131
mp_process_active_p 131
mp_process_enable 131
mp_process_join 132
mp_process_kill 132
mp_process_name 132
mp_process_preset 133
mp_process_resume 132
mp_process_run_function 133
mp_process_suspend 132
mp_process_yield 131
mp_recursive_lock_p 135
mp_restore_signals 133
mp_rwlock_name 136
mp_semaphore_count 138
mp_semaphore_name 138
mp_semaphore_wait_count 138
mp_sempahore_wait(cl_object 138
mp_signal_semaphore 138
mp_try_get_semaphore 138
mp_wait_on_semaphore 138

O

open 76

R

rename-file 73

S

si::string-to-object 167
si:safe-eval 174
si_add_package_local_nickname 155
si_adjust_vector 59
si_make_array 59
si_make_lambda 175
si_make_vector 59
si_package_local_nicknames 154
si_package_locally_nicknamed_by_list 154
si_remove_package_local_nickname 155
si_safe_eval 174
si_string_to_object 167
stream-file-length 153
stream-file-position 152
stream-interactive-p 153
stream-line-length 153
stream-peek-byte 153
stream-read-sequence 153
stream-unread-byte 153
stream-write-sequence 153

T

trace 85
typedef struct { ... } ecl_va_list[1]; 31

W

with-compilation-unit 84

Variable index

C

c:*cc-optimize* 100
c:*user-cc-flags* 100
c:*user-ld-flags* 100
c:*user-linker-flags* 100
c:*user-linker-libs* 100
c::*ar* 101
c::*cc* 101
c::*ecl-include-directory* 101
c::*ecl-library-directory* 101
c::*ld* 101
c::*ranlib* 101

E

ECL_ARRAY_DIMENSION_LIMIT 58
ECL_ARRAY_RANK_LIMIT 58
ECL_ARRAY_TOTAL_LIMIT 58
ECL_CHAR_CODE_LIMIT 169
ECLDIR 19
ext:*default-external-format* 76
ext:*help-message* 101
ext:*lisp-init-file-list* 101
ext:+default-command-arg-rules+ 101
ext:{short,single,double,long}-float-
{positive,negative}-infinity 44

F

ffi:*use-dffi* 115
ffi:+null-cstring-pointer+ 122

M

MOST_NEGATIVE_FIXNUM 168
MOST_POSITIVE_FIXNUM 168
mp:*current-process* 133

S

si:*keep-definitions* 29

Type index

C

cl_elttype 58, 170
cl_fixnum 168
cl_index 168
cl_lispunion 164
cl_object 165
cl_type 166

E

ecl_array 170
ecl_base_string 172
ecl_character 169
ecl_file_ops 173
ecl_stream 173
ecl_string 172
ecl_vector 170
ext:character-coding-error 77
ext:character-decoding-error 77
ext:character-encoding-error 77
ext:sequence-stream 78

ext:stack-overflow 150
ext:storage-exhausted 150
ext:stream-decoding-error 78
ext:stream-encoding-error 77
ext:unix-signal-received 146

Common Lisp symbols

:

:* 115
:byte 115
:cdfloat 115
:char 115
:clfloat 115
:csfloat 115
:cstring 115
:double 115
:float 115
:int 115
:int16-t 115
:int32-t 115
:int64-t 115
:long 115
:long-double 115
:object 115
:pointer-void 115
:short 115
:uint16-t 115
:uint32-t 115
:uint64-t 115
:unsigned-byte 115
:unsigned-char 115
:unsigned-int 115
:unsigned-long 115
:unsigned-short 115
:void 115

C

call-arguments-limit 29
compile-file 83
compile-file-pathname 84

D

debug 25
directory 72
disassemble 84
double-float 27

E

ed 84
ext:*default-external-format* 76
ext:*help-message* 101
ext:*lisp-init-file-list* 101
ext:+default-command-arg-rules+ 101
ext:add-package-local-nickname 155
ext:all-encodings 76
ext:binding-stack 148
ext:c-stack 148
ext:catch-signal 147
ext:character-coding-error 77
ext:character-coding-error-external-format 77
ext:character-decoding-error 77
ext:character-decoding-error-octects 77
ext:character-encoding-error 77
ext:character-encoding-error-code 77
ext:chdir 105
ext:chmod 105
ext:command-args 101
ext:copy-file 105
ext:decoding-error 78
ext:double-float-negative-infinity 44
ext:double-float-positive-infinity 44
ext:encoding-error 78
ext:environ 105
ext:external-process-error-stream 103
ext:external-process-input 103
ext:external-process-output 103
ext:external-process-pid 103
ext:external-process-status 103
ext:external-process-wait 103
ext:file-kind 105
ext:file-stream-fd 76
ext:float-infinity-p 44
ext:float-nan-p 44
ext:frame-stack 148
ext:get-finalizer 150
ext:get-limit 151
ext:get-signal-handler 146
ext:getcwd 105
ext:getenv 105
ext:getpid 105
ext:getuid 105
ext:hash-table-fill 68
ext:hash-table-synchronized-p 68
ext:hash-table-weakness 68
ext:heap-size 148
ext:lisp-stack 148

ext:lock-package 156
ext:long-float-negative-infinity 44
ext:long-float-positive-infinity 44
ext:make-pipe 105
ext:make-sequence-input-stream 78
ext:make-sequence-output-stream 78
ext:nan 44
ext:octets-to-string 64
ext:package-local-nicknames 154
ext:package-locally-nicknamed-by-list 154
ext:package-locked-p 156
ext:peek-byte 76
ext:process-command-args 102
ext:quit 105
ext:remove-package-local-nickname 155
ext:run-program 104
ext:sequence-stream 78
ext:set-buffering-mode 76
ext:set-finalizer 151
ext:set-limit 151
ext:set-signal-handler 146
ext:setenv 105
ext:short-float-negative-infinity 44
ext:short-float-positive-infinity 44
ext:single-float-negative-infinity 44
ext:single-float-positive-infinity 44
ext:stack-overflow 150
ext:stack-overflow-size 150
ext:stack-overflow-type 150
ext:storage-exhausted 150
ext:stream-decoding-error 78
ext:stream-encoding-error 77
ext:string-to-octets 64
ext:system 105
ext:terminate-process 103
ext:trap-fpe 42
ext:unix-signal-received 146
ext:unix-signal-received-code 146
ext:unlock-package 156
ext:unread-byte 76
ext:with-backend 114
ext:with-unlocked-packages 156
ext:without-package-locks 156
extended-char 50

F

ffi:*use-dffi* 115
ffi:+null-cstring-pointer+ 122
ffi:allocate-foreign-object 120
ffi:allocate-foreign-string 127
ffi:c-inline 112
ffi:c-progn 113
ffi:clines 111
ffi:convert-from-cstring 125
ffi:convert-from-foreign-string 126
ffi:convert-to-cstring 125
ffi:convert-to-foreign-string 127
ffi:def-array-pointer 119
ffi:def-constant 116
ffi:def-enum 117
ffi:def-foreign-type 116
ffi:def-foreign-var 123
ffi:def-function 128
ffi:def-struct 118
ffi:def-union 119
ffi:defcallback 114
ffi:defcbody 114
ffi:defentry 114
ffi:defla 115
ffi:deref-array 119
ffi:deref-pointer 121
ffi:ensure-char-character 122
ffi:ensure-char-integer 122
ffi:find-foreign-library 129
ffi:free-cstring 126
ffi:free-foreign-object 120
ffi:get-slot-pointer 118
ffi:get-slot-value 118
ffi:load-foreign-library 128
ffi:make-null-pointer 122
ffi:null-char-p 117
ffi:null-pointer-p 122
ffi:pointer-address 121
ffi:size-of-foreign-type 121
ffi:with-cast-pointer 123
ffi:with-cstring 126
ffi:with-cstrings 126
ffi:with-foreign-object 120
ffi:with-foreign-string 127
ffi:with-foreign-strings 127

H

hash-table-content 68

L

lambda-list-keywords 29
lambda-parameters-limit 29

M

MOST-NEGATIVE-FIXNUM 168
MOST-POSITIVE-FIXNUM 168
mp:*current-process* 133
mp:all-processes 130
mp:allow-with-interrupts 133
mp:atomic-decf 140
mp:atomic-incf 140
mp:atomic-pop 141
mp:atomic-push 141
mp:atomic-update 141
mp:barrier-arrivers-count 139
mp:barrier-count 139
mp:barrier-name 139

mp:barrier-unblock 139
mp:barrier-wait 139
mp:block-signals 133
mp:compare-and-swap 140
mp:condition-variable-broadcast 137
mp:condition-variable-signal 137
mp:condition-variable-timedwait 137
mp:condition-variable-wait 137
mp:defcas 142
mp:define-cas-expander 141
mp:exit-process 130
mp:get-cas-expansion 143
mp:get-lock 135
mp:get-rwlock-read 136
mp:get-rwlock-write 136
mp:giveup-lock 135
mp:giveup-rwlock-read 136
mp:giveup-rwlock-write 136
mp:holding-lock-p 135
mp:interrupt-process 130
mp:lock-count 135
mp:lock-name 135
mp:lock-owner 135
mp:make-barrier 139
mp:make-condition-variable 137
mp:make-lock 134
mp:make-process 131
mp:make-rwlock 136
mp:make-semaphore 138
mp:process-active-p 131
mp:process-enable 131
mp:process-join 132
mp:process-kill 132
mp:process-name 132
mp:process-preset 133
mp:process-resume 132
mp:process-run-function 133
mp:process-suspend 132
mp:process-yield 131
mp:recursive-lock-p 135
mp:remcas 143
mp:restore-signals 133
mp:rwlock-name 136
mp:semaphore-count 138
mp:semaphore-name 138
mp:semaphore-wait 138
mp:semaphore-wait-count 138
mp:signal-semaphore 138
mp:try-get-semaphore 138
mp:wait-on-semaphore 138
mp:with-interrupts 134
mp:with-local-interrupts 133
mp:with-lock 135
mp:with-restored-interrupts 133
mp:with-rwlock 136
mp:without-interrupts 133
mp_lock_owner 135
multiple-values-limit 29

O

open 76
optimize 25

R

read-char 74

S

safety 25
si:*keep-definitions* 29
si::make-lambda 28
single-float 27
space 25
speed 25
standard-char 50

T

trace 84

W

with-compilation-unit 84
write-char 74
write-sequence 74

C/C++ index

_

_ecl_caar 53
_ecl_cadr 53
_ecl_car 53
_ecl_cdr 53

C

c_string_to_object 167
CHAR_CODE 169
CHAR_CODE_LIMIT 169
cl_boot 16
cl_elttype 58, 170
cl_env_ptr 175
cl_env_struct 175
cl_eval 174
cl_fixnum 167, 168
cl_index 168
cl_lispunion 164
cl_object 165
cl_safe_eval 174
cl_shutdown 16
CODE_CHAR 169

E

ecl_aet_to_symbol 58
ecl_alloc_adjustable_base_string 63
ecl_alloc_simple_base_string 63
ecl_alloc_simple_vector 59
ecl_alpha_char_p 52
ecl_alphanumeric_p 52
ecl_aref 60, 171
ecl_aref1 ..... 60, 172
ecl_array 170
ecl_array_dimension 60
ecl_array_elttype 60, 171
ecl_array_rank 60
ecl_aset 60, 171
ecl_aset1 ..... 60, 172
ecl_atomic_get 140
ecl_atomic_incf 140
ecl_atomic_incf_by_fixnum 140
ecl_atomic_index_incf 140
ecl_atomic_pop 140
ecl_atomic_push 140
ecl_base_char 51, 168
ecl_base_char_code 51, 169
ecl_base_char_p 52
ecl_base_string 172
ecl_base_string_pointer_safe 109
ecl_bds_bind 30
ecl_bds_push 30
ecl_bds_unwind_n 31
ecl_bds_unwind1 31
ecl_both_case_p 52
ecl_cdfloat 46
ecl_char 64
ecl_char_cmp 169
ecl_char_code 51, 169
ecl_char_compare 169
ecl_char_downcase 52
ecl_char_eq 169
ecl_char_equal 169
ecl_char_set 64
ecl_char_upcase 52
ecl_character 51, 168
ecl_clear_interrupts 21
ecl_clfloat 46
ecl_compare_and_swap 139
ecl_csfloat 46
ecl_decode_from_cstring 64
ecl_decode_from_unicode_wstring 65
ecl_digit_p 52
ecl_disable_interrupts 21
ecl_double_float 46
ecl_enable_interrupts 21
ecl_encode_to_cstring 65
ecl_encode_to_unicode_wstring 65
ecl_file_pos 173
ecl_fixnum 46, 168
ecl_fixnum_geq 168
ecl_fixnum_greater 168
ecl_fixnum_leq 168
ecl_fixnum_lower 168
ecl_fixnum_minusp 168
ecl_fixnum_plusp 168
ecl_foreign_data_pointer_safe 108
ecl_get_option 19
ecl_graphic_char_p 52
ecl_import_current_thread 19
ecl_init_module 92
ecl_long_float 46
ecl_lower_case_p 52
ecl_make_barrier 139
ecl_make_cdfloat 45
ecl_make_clfloat 45
ecl_make_complex 45
ecl_make_constant_base_string 63
ecl_make_csfloat 45
ecl_make_double_float 45
ecl_make_fixnum 45, 168
ecl_make_foreign_data 108
ecl_make_int 45
ecl_make_int16_t 45
ecl_make_int32_t 45
ecl_make_int64_t 45
ecl_make_int8_t 45
ecl_make_integer 45
ecl_make_keyword 38
ecl_make_lock 134
ecl_make_long 45
ecl_make_long_float 45
ecl_make_long_long 45
ecl_make_pointer 108
ecl_make_ratio 45

ecl_make_rwlock 136
ecl_make_semaphore 138
ecl_make_short_t 45
ecl_make_simple_base_string 63
ecl_make_single_float 45
ecl_make_symbol 39
ecl_make_uint 45
ecl_make_uint16_t 45
ecl_make_uint32_t 45
ecl_make_uint64_t 45
ecl_make_uint8_t 45
ecl_make_ulong 45
ecl_make_ulong_long 45
ecl_make_unsigned_integer 45
ecl_make_ushort_t 45
ecl_nth_value 32
ecl_null_terminated_base_string 109
ecl_nvalues 32
ecl_process_env 27
ecl_read_from_cstring 167
ecl_release_current_thread 19
ecl_return0 33
ecl_return1 33
ecl_return2 33
ecl_return3 33
ecl_set_option 16
ecl_setq 31
ecl_single_float 46
ecl_standard_char_p 52
ecl_stream 173
ecl_string 172
ecl_symbol_to_aet 58
ecl_symbol_value 31
ecl_t_of 166
ecl_to_cdfloat 46
ecl_to_clfloat 46
ecl_to_csfloat 46
ecl_to_double 46
ecl_to_fixnum 46
ecl_to_float 46
ecl_to_int 46
ecl_to_int16_t 46
ecl_to_int32_t 46
ecl_to_int64_t 46
ecl_to_int8_t 46
ecl_to_long 46
ecl_to_long_double 46
ecl_to_long_long 46
ecl_to_pointer 108
ecl_to_short 46
ecl_to_uint 46
ecl_to_uint16_t 46
ecl_to_uint32_t 46
ecl_to_uint64_t 46
ecl_to_uint8_t 46
ecl_to_ulong 46
ecl_to_ulong_long 46
ecl_to_unsigned_integer 46
ecl_to_ushort 46
ecl_upper_case_p 52
ecl_va_arg 31
ecl_va_end 31
ecl_va_list 31
ecl_va_start 31
ecl_vector 170
ECL_ADJUSTABLE_ARRAY_P 169
ECL_ANSI_STREAM_P 174
ECL_ANSI_STREAM_TYPE_P 174
ECL_ARRAY_DIMENSION_LIMIT 58
ECL_ARRAY_HAS_FILL_POINTER_P 169
ECL_ARRAY_RANK_LIMIT 58
ECL_ARRAY_TOTAL_LIMIT 58
ECL_ARRAYP 166
ECL_ATOM 166
ECL_BASE_CHAR_CODE_P 166
ECL_BASE_CHAR_P 166
ECL_BASE_STRING_P 172
ECL_BIGNUMP 166
ECL_BIT_VECTOR_P 166
ECL_BLOCK_BEGIN 33
ECL_CATCH_ALL 19
ECL_CATCH_BEGIN 33
ECL_CHAR_CODE 51, 169
ECL_CHAR_CODE_LIMIT 169
ECL_CHARACTERP 166
ECL_CLASS_CPL 174
ECL_CLASS_INFERIORS 174
ECL_CLASS_NAME 174
ECL_CLASS_OF 174
ECL_CLASS_SLOTS 174
ECL_CLASS_SUPERIORS 174
ECL_CODE_CHAR 51, 169
ECL_COMPLEXP 166
ECL_CONS_CAR 53
ECL_CONS_CDR 53
ECL_CONSP 166
ECL_DOUBLE_FLOAT_P 166
ECL_EXTENDED_STRING_P 172
ECL_FIXNUMP 166
ECL_FOREIGN_DATA_P 166
ECL_HANDLER_CASE 36
ECL_HASH_TABLE_P 166
ECL_IMMEDIATE 167
ECL_INSTANCEP 174
ECL_LISTP 166
ECL_LONG_FLOAT_P 166
ECL_NUMBER_TYPE_P 166
ECL_PACKAGEP 166
ECL_PATHNAMEP 166
ECL_RANDOM_STATE_P 166
ECL_READTABLEP 166
ECL_REAL_TYPE_P 166
ECL_RESTART_CASE 37
ECL_RPLACA 53
ECL_RPLACD 53
ECL_SINGLE_FLOAT_P 166

ECL_SPEC_FLAG 174
ECL_SPEC_OBJECT 174
ECL_SSE_PACK_P 166
ECL_STRINGP 166
ECL_STRUCT_LENGTH 174
ECL_STRUCT_NAME 174
ECL_STRUCT_SLOT 174
ECL_STRUCT_SLOTS 174
ECL_STRUCT_TYPE 174
ECL_SYMBOLP 166
ECL_UNWIND_PROTECT 20
ECL_UNWIND_PROTECT_BEGIN 33
ECL_VECTORP 166
ECL_WITH_LISP_FPE 21

F

fix 168
fixint 168
fixnint 168

M

MAKE_FIXNUM 168
MOST_NEGATIVE_FIXNUM 168
MOST_POSITIVE_FIXNUM 168
mp_all_processes 130
mp_barrier_arrivers_count 139
mp_barrier_count 139
mp_barrier_name 139
mp_barrier_unblock 139
mp_barrier_wait 139
mp_block_signals 133
mp_condition_variable-broadcast 137
mp_condition_variable_signal 137
mp_condition_variable_timedwait 137
mp_condition_variable_wait 137
mp_current_process 133
mp_exit_process 130
mp_get_lock_nowait 135
mp_get_lock_wait 135
mp_get_rwlock_read_nowait 136
mp_get_rwlock_read_wait 136
mp_get_rwlock_write_nowait 136
mp_get_rwlock_write_wait 136
mp_giveup_lock 135
mp_giveup_rwlock_read 136
mp_giveup_rwlock_write 136
mp_holding_lock_p 135
mp_interrupt_process 130
mp_lock_count 135
mp_lock_name 135
mp_make_barrier 139
mp_make_condition_variable 137
mp_make_process 131
mp_make_semaphore 138
mp_process-join 132
mp_process_active_p 131
mp_process_enable 131
mp_process_kill 132
mp_process_name 132
mp_process_preset 133
mp_process_resume 132
mp_process_run_function 133
mp_process_suspend 132
mp_process_yield 131
mp_recursive_lock_p 135
mp_restore_signals 133
mp_rwlock_name 136
mp_semaphore_count 138
mp_semaphore_name 138
mp_semaphore_wait 138
mp_semaphore_wait_count 138
mp_signal_semaphore 138
mp_try_get_semaphore 138
mp_wait_on_semaphore 138

S

si_add_package_local_nickname 155
si_adjust_vector 59
si_make_array 59
si_make_lambda 175
si_make_vector 59
si_octets_to_string 64
si_package_local_nicknames 154
si_package_locally_nicknamed_by_list 154
si_remove_package_local_nickname 155
si_safe_eval 174
si_string_to_object 167
si_string_to_octets 64

T

t_array 166
t_barrier 166
t_base_string 166
t_bclosure 166
t_bignum 166
t_bitvector 166
t_bytecodes 166
t_cclosure 166
t_cdfloat 166
t_cfun 166
t_cfunfixed 166
t_character 166
t_clfloat 166
t_codeblock 166
t_complex 166
t_condition_variable 166
t_contiguous -- contiguous block 166
t_csfloat 166
t_end 166
t_fixnum 166
t_foreign 166
t_frame 166

t_hashtable ... 166
t_instance ... 166
t_list ... 166
t_lock ... 166
t_longfloat ... 166
t_mailbox ... 166
t_other ... 166
t_package ... 166
t_pathname ... 166
t_process ... 166
t_random ... 166
t_ratio ... 166
t_readtable ... 166
t_rwlock ... 166
t_semaphore ... 166
t_singlefloat ... 166
t_sse_pack ... 166
t_start ... 166
t_stream ... 166
t_string ... 166
t_structure = t_instance ... 166
t_symbol ... 166
t_vector ... 166
t_weak_pointer ... 166

Bibliography

ANSI ANSI Common-Lisp Specification, 1986.

AMOP Gregor Kickzales et al. "The Art of the Metaobject Protocol" The M.I.T. Press, Massachussets Institute of Technology, 1999

LISP1.5 John McCarthy et al. "Lisp 1.5 Programmer's Manual 2nd ed" The M.I.T. Press, Massachussets Institute of Technology, 1985

Steele:84 Guy L. Steele Jr. et al. "Common Lisp: the Language", Digital Press, 1984.

Steele:90 Guy L. Steele Jr. at al. "Common Lisp: the Language II", second edition, Digital Press, 1990.

Yasa:85 Taiichi Yuasa and Masami Hagiya "Kyoto Common-Lisp Report", Research Institute for Mathematical Sciences, Kyoto University, 1988.

Attardi:95 Giusseppe Attardi "Embeddable Common-Lisp", ACM Lisp Pointers, 8(1), 30-41, 1995

Smith:84 B.C. Smith and J. des Rivieres "The Implementation of Procedurally Reflective Languages", *Proc. of the 1984 ACM Symposium on LISP and Functional Programming*, 1984.