Copyright and Trademarks


Version 8.0

December 2021

Copyright © 2021 by LispWorks Ltd.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of LispWorks Ltd.

The information in this publication is provided for information only, is subject to change without notice, and should not be construed as a commitment by LispWorks Ltd. LispWorks Ltd assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks and KnowledgeWorks are registered trademarks of LispWorks Ltd.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.


The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom: Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

LispWorks contains part of ICU software obtained from http://source.icu-project.org and which bears the following copyright and permission notice:

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE
Copyright © 1995-2006 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Copyright and Trademarks

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder. All trademarks and registered trademarks mentioned herein are the property of their respective owners.

US Government Restricted Rights
The LispWorks Software is a commercial computer software program developed at private expense and is provided with restricted rights. The LispWorks Software may not be used, reproduced, or disclosed by the Government except as set forth in the accompanying End User License Agreement and as provided in DFARS 227.7202-1(a), 227.7202-3(a) (1995), FAR 12.212(a)(1995), FAR 52.227-19, and/or FAR 52.227-14 Alt III, as applicable. Rights reserved under the copyright laws of the United States.

<table>
<thead>
<tr>
<th>Address</th>
<th>Telephone</th>
<th>Fax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LispWorks Ltd</td>
<td>From North America: 877 759 8839 (toll-free)</td>
<td>From North America: 617 812 8283</td>
</tr>
<tr>
<td>St. John’s Innovation Centre</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cowley Road</td>
<td>From elsewhere: +44 1223 421860</td>
<td>From elsewhere: +44 870 2206189</td>
</tr>
<tr>
<td>Cambridge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CB4 0WS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>England</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

www.lispworks.com
# Contents

**Preface** 11

1 Introduction to the FLI 12
   1.1 An example of interfacing to a foreign function 12
   1.2 Using the FLI to get the cursor position 13
   1.3 Using the FLI to set the cursor position 15
   1.4 An example of dynamic memory allocation 16
   1.5 Summary 16

2 FLI Types 17
   2.1 Immediate types 17
   2.2 Aggregate types 18
   2.3 Parameterized types 23
   2.4 Encapsulated types 23
   2.5 The void type 24
   2.6 Summary 24

3 FLI Pointers 25
   3.1 Creating and copying pointers 25
   3.2 Pointer testing functions 26
   3.3 Pointer dereferencing and coercing 27
   3.4 An example of dynamic pointer allocation 28
   3.5 More examples of allocation and pointer allocation 29
   3.6 Summary 31

4 Defining foreign functions and callables 32
   4.1 Foreign callables and foreign functions 32
   4.2 Specifying a calling convention. 34

5 Advanced Uses of the FLI 37
   5.1 Passing a string to a Windows function 37
   5.2 Passing and returning strings 38
   5.3 Lisp integers 47
   5.4 Defining new types 48
   5.5 Using DLLs within the LispWorks FLI 48
   5.6 Incorporating a foreign module into a LispWorks image 49
   5.7 Block objects in C (foreign blocks) 50
6 Self-contained examples 53

6.1 Foreign block examples 53
6.2 Miscellaneous examples 53

7 Function, Macro and Variable Reference 54

align-of 54
alloca 55
allocate-dynamic-foreign-object 55
allocate-foreign-block 56
allocate-foreign-object 57
cast-integer 59
connected-module-pathname 59
convert-from-foreign-string 61
convert-integer-to-dynamic-foreign-object 62
convert-to-dynamic-foreign-string 62
convert-to-foreign-string 64
copy-pointer 65
defc-pointer 66
define-c-enum 67
define-c-struct 69
define-c-typedef 71
define-c-union 72
define-foreign-block-callable-type 74
define-foreign-block-invoker 75
define-foreign-callable 76
define-foreign-converter 79
define-foreign-forward-reference-type 81
define-foreign-function 82
define-foreign-pointer 87
define-foreign-type 88
define-foreign-variable 89
define-opaque-pointer 92
dereference 93
disconnect-module 95
does-symbol 96
does-symbol-value 96
does-symbol-value-pairs 96
does-values 96
does-value-symbol 96
fill-foreign-object 97
## Contents

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreign-aref</td>
<td>98</td>
</tr>
<tr>
<td>foreign-array-dimensions</td>
<td>100</td>
</tr>
<tr>
<td>foreign-array-element-type</td>
<td>100</td>
</tr>
<tr>
<td>foreign-array-pointer</td>
<td>101</td>
</tr>
<tr>
<td>foreign-block-copy</td>
<td>102</td>
</tr>
<tr>
<td>foreign-block-release</td>
<td>103</td>
</tr>
<tr>
<td>foreign-function-pointer</td>
<td>104</td>
</tr>
<tr>
<td>foreign-slot-names</td>
<td>105</td>
</tr>
<tr>
<td>foreign-slot-offset</td>
<td>106</td>
</tr>
<tr>
<td>foreign-slot-pointer</td>
<td>107</td>
</tr>
<tr>
<td>foreign-slot-type</td>
<td>109</td>
</tr>
<tr>
<td>foreign-slot-value</td>
<td>110</td>
</tr>
<tr>
<td>foreign-typed-aref</td>
<td>112</td>
</tr>
<tr>
<td>foreign-type-equal-p</td>
<td>113</td>
</tr>
<tr>
<td>foreign-type-error</td>
<td>114</td>
</tr>
<tr>
<td>free</td>
<td>115</td>
</tr>
<tr>
<td>free-foreign-block</td>
<td>114</td>
</tr>
<tr>
<td>free-foreign-object</td>
<td>115</td>
</tr>
<tr>
<td>get-embedded-module</td>
<td>116</td>
</tr>
<tr>
<td>get-embedded-module-data</td>
<td>117</td>
</tr>
<tr>
<td>incf-pointer</td>
<td>118</td>
</tr>
<tr>
<td>install-embedded-module</td>
<td>119</td>
</tr>
<tr>
<td><em>install-embedded-module-delay-delete</em></td>
<td>120</td>
</tr>
<tr>
<td><em>locale-external-formats</em></td>
<td>121</td>
</tr>
<tr>
<td>make-integer-from-bytes</td>
<td>122</td>
</tr>
<tr>
<td>make-pointer</td>
<td>122</td>
</tr>
<tr>
<td>malloc</td>
<td>57</td>
</tr>
<tr>
<td>module-unresolved-symbols</td>
<td>124</td>
</tr>
<tr>
<td><em>null-pointer</em></td>
<td>125</td>
</tr>
<tr>
<td>null-pointer-p</td>
<td>125</td>
</tr>
<tr>
<td>pointer-address</td>
<td>126</td>
</tr>
<tr>
<td>pointer-element-size</td>
<td>127</td>
</tr>
<tr>
<td>pointer-element-type</td>
<td>128</td>
</tr>
<tr>
<td>pointer-element-type-p</td>
<td>129</td>
</tr>
<tr>
<td>pointer-eq</td>
<td>130</td>
</tr>
<tr>
<td>pointerp</td>
<td>131</td>
</tr>
<tr>
<td>pointer-pointer-type</td>
<td>132</td>
</tr>
<tr>
<td>print-collected-template-info</td>
<td>133</td>
</tr>
<tr>
<td>print-foreign-modules</td>
<td>133</td>
</tr>
<tr>
<td>register-module</td>
<td>134</td>
</tr>
<tr>
<td>replace-foreign-array</td>
<td>138</td>
</tr>
<tr>
<td>replace-foreign-object</td>
<td>140</td>
</tr>
<tr>
<td>set-locale</td>
<td>141</td>
</tr>
<tr>
<td>set-locale-encodings</td>
<td>142</td>
</tr>
</tbody>
</table>
8 Type Reference 158

: boolean 158
: byte 159
: c-array 159
: char 161
: const 161
: double 162
: double-complex 163
: ef-mb-string 163
: ef-wc-string 164
: enum 165
: enumeration 165
: fixnum 166
: float 166
: float-complex 167
: foreign-array 167
: foreign-block-pointer 168
: function 169
: int16 170
: int32 170
: int64 170
: int8 170
: int 171
: intmax 170
: intptr 170
: lisp-array 171
: lisp-double-float 173
: lisp-float 173
: lisp-simple-1d-array 174
: lisp-single-float 175
### Contents

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>175</td>
</tr>
<tr>
<td>long-long</td>
<td>176</td>
</tr>
<tr>
<td>one-of</td>
<td>177</td>
</tr>
<tr>
<td>pointer</td>
<td>178</td>
</tr>
<tr>
<td>ptr</td>
<td>178</td>
</tr>
<tr>
<td>ptdiff-t</td>
<td>178</td>
</tr>
<tr>
<td>reference</td>
<td>179</td>
</tr>
<tr>
<td>reference-pass</td>
<td>180</td>
</tr>
<tr>
<td>reference-return</td>
<td>181</td>
</tr>
<tr>
<td>released-foreign-block-pointer</td>
<td>181</td>
</tr>
<tr>
<td>short</td>
<td>182</td>
</tr>
<tr>
<td>signed</td>
<td>183</td>
</tr>
<tr>
<td>size-t</td>
<td>184</td>
</tr>
<tr>
<td>ssize-t</td>
<td>184</td>
</tr>
<tr>
<td>struct</td>
<td>185</td>
</tr>
<tr>
<td>time-t</td>
<td>186</td>
</tr>
<tr>
<td>uint16</td>
<td>186</td>
</tr>
<tr>
<td>uint32</td>
<td>186</td>
</tr>
<tr>
<td>uint64</td>
<td>186</td>
</tr>
<tr>
<td>uint8</td>
<td>186</td>
</tr>
<tr>
<td>uintmax</td>
<td>186</td>
</tr>
<tr>
<td>uintptr</td>
<td>186</td>
</tr>
<tr>
<td>union</td>
<td>187</td>
</tr>
<tr>
<td>unsigned</td>
<td>188</td>
</tr>
<tr>
<td>vector-char16</td>
<td>189</td>
</tr>
<tr>
<td>vector-char2</td>
<td>189</td>
</tr>
<tr>
<td>vector-char3</td>
<td>189</td>
</tr>
<tr>
<td>vector-char32</td>
<td>189</td>
</tr>
<tr>
<td>vector-char4</td>
<td>189</td>
</tr>
<tr>
<td>vector-char8</td>
<td>189</td>
</tr>
<tr>
<td>vector-double2</td>
<td>190</td>
</tr>
<tr>
<td>vector-double3</td>
<td>190</td>
</tr>
<tr>
<td>vector-double4</td>
<td>191</td>
</tr>
<tr>
<td>vector-double8</td>
<td>191</td>
</tr>
<tr>
<td>vector-float16</td>
<td>190</td>
</tr>
<tr>
<td>vector-float2</td>
<td>190</td>
</tr>
<tr>
<td>vector-float3</td>
<td>190</td>
</tr>
<tr>
<td>vector-float4</td>
<td>190</td>
</tr>
<tr>
<td>vector-float8</td>
<td>190</td>
</tr>
<tr>
<td>vector-int16</td>
<td>190</td>
</tr>
<tr>
<td>vector-int2</td>
<td>190</td>
</tr>
<tr>
<td>vector-int3</td>
<td>190</td>
</tr>
<tr>
<td>vector-int4</td>
<td>190</td>
</tr>
<tr>
<td>vector-int8</td>
<td>190</td>
</tr>
</tbody>
</table>
9 The Foreign Parser 196

9.1 Introduction 196
9.2 Loading the Foreign Parser 196
9.3 Using the Foreign Parser 196
9.4 Using the LispWorks Editor 198
9.5 Foreign Parser Reference 198
Contents

*preprocessor* 198
*preprocessor-format-string* 199
*preprocessor-include-path* 199
*preprocessor-options* 200
process-foreign-file 200

Glossary 203

Index
Preface

This manual documents the Foreign Language Interface (FLI), which provides a toolkit for the development of interfaces between Common Lisp and other programming languages, and supersedes the Foreign Function Interface (FFI).

The manual is divided into three sections: a user guide to the FLI which includes illustrative examples indicating how to use the FLI for a variety of purposes, a reference section providing complete details of the functions, macros, variables and types that make up the FLI, and a guide to the Foreign Parser.

The user guide section starts by describing the ideas behind the FLI, followed by a few simple examples presenting some of the more commonly used features of the FLI. The next chapter explains the existing type system, and includes examples showing how to define new types. This is followed by chapters explaining the FLI implementation of pointers and some of the more advanced topics. Finally, 6 Self-contained examples enumerates relevant example Lisp source files which are available in the LispWorks library.

The reference section consists of a chapter documenting the functions and macros that constitute the FLI, and a chapter documenting the FLI variables and types.

The Foreign Parser section describes a helper tool for generating FLI definitions from a C header file.

Viewing example files

This manual refers to example files in the LispWorks library via a Lisp form like this:

(example-edit-file "fli/foreign-callable-example")

These examples are Lisp source files in your LispWorks installation under lib/8-0-0-0/examples/. You can simply evaluate the given form to view the example source file.

Example files contain instructions about how to use them at the start of the file.

The examples files are in a read-only directory and therefore you should compile them inside the IDE (by the Editor command Compile Buffer or the toolbar button or by choosing Buffer > Compile from the context menu), so it does not try to write a fasl file.

If you want to manipulate an example file or compile it on the disk rather than in the IDE, then you need first to copy the file elsewhere (most easily by using the Editor command Write File or by choosing File > Save As from the context menu).

11
1 Introduction to the FLI

The Foreign Language Interface (FLI) is an extension to LispWorks which allows you to call functions written in a foreign language from LispWorks, and to call Lisp functions from a foreign language. The FLI currently supports C (and therefore also the Win32 API for Microsoft Windows users).

The main problem in interfacing different languages is that they usually have different type systems, which makes it difficult to pass data from one to the other. The FLI solves the problem of interfacing Lisp with C. It consists of FLI types that have obvious parallels to the C types and structures, and FLI functions that allow LispWorks to define new FLI types and set their values. The FLI also contains functions for passing FLI objects to C, and functions for receiving data from C.

To interface to a C++ program from LispWorks, define C stubs which call your C++ entry points, as described in 5.5.2 Using C++ DLLs. Use the FLI to interface to these C stubs.

1.1 An example of interfacing to a foreign function

The following example shows how to use the FLI to call a C function. The function to interface with, FahrenheitToCelsius, takes one integer as its argument (the temperature in Fahrenheit) and returns the result as a single float (the temperature in Celsius).

The example consists of three stages: defining a foreign language interface to the C function, loading the foreign code into the Lisp image, and calling the C function to obtain the results.

1.1.1 Defining the FLI function

The FLI provides the macro define-foreign-function for creating interfaces to foreign functions. It takes the name of the function you wish to interface to, the argument types the function accepts, and the result type the function returns.

Given the following C declaration to FahrenheitToCelsius:

```c
float FahrenheitToCelsius( int );
```

The FLI interface is as follows:

```lisp
(fli:define-foreign-function
   (fahrenheit-to-celsius "FahrenheitToCelsius" :source)
   ((fahrenheit :int))
   :result-type :float
   :language :ansi-c)
```

The first argument to define-foreign-function declares that fahrenheit-to-celsius is the name of the Lisp function that is generated to interface with the C function FahrenheitToCelsius. The :source keyword is a directive to define-foreign-function that FahrenheitToCelsius is the name of the C function as seen in the source files. On some platforms the actual symbol name available in the foreign object file we are interfacing with could include character prefixes such as "." and "_", and so the :source keyword encoding allows you to write cross-platform portable foreign language interfaces.

The second argument to define-foreign-function, ((fahrenheit :int)), is the argument list for the foreign function. In this case, only one argument is required. The first part of each argument descriptor is the lambda argument
name. The rest of the argument describes the type of argument we are trying to interface to and how the conversion from Lisp to C is performed. In this case the foreign type :int specifies that we are interfacing between a Lisp integer and a C type "int".

The :result-type keyword tells us that the conversion required between the C function and Lisp uses the foreign type :float. This tells Lisp that C will return a result of type "float", which needs to be converted to a Lisp single-float.

The final keyword argument, :language, specifies which language the foreign function was written in. In this case the example uses ANSI C. This keyword determines how single-floating point values are passed to and returned from C functions as described for define-foreign-function.

### 1.1.2 Loading foreign code

Once an interface has been created, the object code defining those functions (and indeed any variables) must be made available to LispWorks.

LispWorks for Windows can load Windows Dynamic Link Libraries (.DLL files).

LispWorks for Linux, LispWorks for x86/x64 Solaris and LispWorks for FreeBSD can load shared libraries (typically .so files).

LispWorks for Macintosh can load Mach-O dynamically-linked shared libraries (typically .dylib files).

Throughout this manual we shall refer to these dynamic libraries as DLLs.

On all platforms the function register-module is the main LispWorks interface to DLL files. It is used to specify which DLLs are looked up when searching for foreign symbols. Here are example forms to register a connection to a DLL.

On Windows:

```
(fli:register-module "MYDLL.DLL")
```

On Linux:

```
(fli:register-module "mylib.so")
```

On macOS:

```
(fli:register-module "mylib.dylib")
```

**Note:** It is also possible to embed a DLL in the Lisp image. See 5.6 Incorporating a foreign module into a LispWorks image.

### 1.1.3 Calling foreign code

Calling the foreign code is the simplest part of using the FLI. The interface to the C function, defined using define-foreign-function, is called like any other Lisp function. In our example, the fahrenheit-to-celsius function takes the temperature in Fahrenheit as its only argument, and returns the temperature in Celsius.

### 1.2 Using the FLI to get the cursor position

**Note:** The rest of the examples in this chapter only work in LispWorks for Windows.

The following example shows how to use the FLI to call a C function in a Win32 library. The function we are going to call returns the screen position of the mouse pointer, or cursor. The example consists of three stages: setting up the correct data
types to pass and receive the data, defining and calling a FLI function to call the Win32 function, and collecting the values returned by the Win32 function to find where the cursor is.

1.2.1 Defining FLI types

The example uses the FLI to find the position of the cursor using the Windows function `GetCursorPos`, which has the following C prototype:

```c
BOOL GetCursorPos( LPPOINT )
```

The `LPPOINT` argument is a pointer to the `POINT` structure, which has the following C definition:

```c
typedef struct tagPOINT {
    LONG x;
    LONG y;
} POINT;
```

First we use the `define-c-typedef` macro to define a number of basic types which are needed to pass data to and from the Windows function.

```lisp
(fli:define-c-typedef bool (:boolean :int))
(fli:define-c-typedef long :long)
```

This defines two types, `BOOL` and `LONG`, which are used to associate a Lisp boolean value (`t` or `nil`) with a C boolean of type `int`, and a Lisp `bignum` with a C `long`. These are required because the Windows function `GetCursorPos` returns a boolean to indicate if it has executed successfully, and the cursor's `x` and `y` positions are specified in a `long` format in the `POINT` structure.

Next, we need to define a structure for the FLI which is used to get the coordinates of the cursor. These coordinates will consist of an `x` and `y` position. We use the `define-c-typedef` macro for this, and the resulting Lisp FLI code has obvious parallels with the C `tagPOINT` structure.

```lisp
(fli:define-c-struct tagpoint
    (x long)
    (y long))
```

The `tagPOINT` structure for the FLI, corresponding to the C structure of the same name, has been defined. This now needs to be further defined as a type for the FLI, using `define-c-typedef`.

```lisp
(fli:define-c-typedef point (:struct tagpoint))
```

Finally, a pointer type to point to the structure is required. It is this FLI pointer which will be passed to the Windows function `GetCursorPos`, so that `GetCursorPos` can change the `x` and `y` values of the structure pointed to.

```lisp
(fli:define-c-typedef lppoint (:pointer point))
```

All the required FLI types have now been defined. Although it may seem that there is a level of duplicity in the definitions of the structures, pointers and types in this section, this was necessary to match the data structures of the C functions to which the FLI will interface. We can now move on to the definition of FLI functions to perform the interfacing.
1.2.2 Defining a FLI function

This next step uses the `define-foreign-function` macro to define a FLI function, or interface function, to be used to call the `GetCursorPos` function. An interface function takes its arguments, converts them into a C format, calls the foreign function, receives the return values, and converts them into a suitable Lisp format.

```lisp
(fli:define-foreign-function (get-cursor-position "GetCursorPos")
  ((lp-point lppoint))
  :result-type bool)
```

In this example, the defined FLI function is `get-cursor-position`. It takes as its argument a pointer of type `lppoint`, converts this to a C format, and calls `GetCursorPos`. It takes the return value it receives from `GetCursorPos` and converts it into the FLI `bool` type we defined earlier.

We have now defined all the types and functions required to get the cursor position. The next step is to allocate memory for an instance of the `tagPOINT` structure using `allocate-foreign-object`. The following line of code binds `location` to a pointer that points to such an instance.

```lisp
(setq location (fli:allocate-foreign-object :type 'point))
```

Finally, we can use our interface function `get-cursor-position` to get the cursor position:

```lisp
(get-cursor-position location)
```

1.2.3 Accessing the results

The position of the cursor is now stored in a `POINT` structure in memory, and `location` is a pointer to that location. To find out what values are stored we use the `foreign-slot-value` accessor, which returns the value stored in the specified field of the structure.

```lisp
(fli:foreign-slot-value location 'x)
(fli:foreign-slot-value location 'y)
```

1.3 Using the FLI to set the cursor position

A similar Windows function, `SetCursorPos`, can be used to set the cursor position. The `SetCursorPos` function takes two `LONGs`. The following code defines an interface function to call `SetCursorPos`.

```lisp
(fli:define-foreign-function (set-cursor-position "SetCursorPos")
  ((x :long)
   (y :long))
  :result-type :boolean)
```

For example, the cursor position can now be set to be near the top left corner by simply using the following command:

```lisp
(set-cursor-position 20 20)
```

For a more extravagant example, define and execute the following function:

```lisp
(defun test-cursor ()
  (dotimes (x 10)
    (dotimes (d 300)
      (let ((r (/ (+ d (* 300 x)) 10.0)))
  
)
1 Introduction to the FLI

(set-cursor-position
 (+ 300 (floor (* x (cos (/ (* d pi) 150.0))))))
 (+ 300 (floor (* y (sin (/ (* d pi) 150.0))))))
)))))

(test-cursor)

1.4 An example of dynamic memory allocation

In the previous example our defined interface function get-cursor-position used the function allocate-foreign-object to allocate memory for an instance of a POINT structure. This memory is now reserved, with a pointer to its location bound to the variable location. More detailed information on pointers is available in 3 FLI Pointers. To free the memory associated with the foreign object requires the use of the function free-foreign-object.

(defun test-cursor (location)
  (if (get-cursor-position location)
      (values t (fli:foreign-slot-value lppoint 'x)
              (fli:foreign-slot-value lppoint 'y))
      (values nil 0 0))))

On calling current-cursor-position the following happens:

1. The macro with-dynamic-foreign-objects is called, which ensures that the lifetime of any allocated objects is within the scope of the code specified in its body.

2. The function allocate-dynamic-foreign-object is called to create an instance of the relevant data structure required to get the cursor position. Refer to it using the lppoint pointer.

3. The previously defined foreign function get-cursor-position is called with lppoint.

4. Provided the call to GetCursorPos was successful the function foreign-slot-value is called twice, once to return the value in the x slot and again to return the value in the y slot. If the call was unsuccessful then 0 0 nil is returned.

1.5 Summary

In this chapter an introduction to some of the FLI functions and types was presented. Some examples demonstrating how to interface LispWorks with Windows and C functions were presented. The first example involved defining a foreign function using define-foreign-function to call a C function that converts between Fahrenheit and Celsius. The second involved setting up foreign types, using the FLI macros define-c-typedef and define-c-struct, and defining a foreign function using the FLI macro define-foreign-function, with which to obtain data from the Windows function GetCursorPos. The third example consisted of defining a foreign function to pass data to the Windows function SetCursorPos. A further example illustrated how to manage the allocation of memory for creating instances of foreign objects more carefully using the FLI macro with-dynamic-foreign-objects.
2 FLI Types

A central aspect of the FLI is implementation of foreign language types. FLI variables, function arguments and temporary objects have predictable properties and structures which are analogous to the properties and structures of the types found in C. The FLI can translate Lisp data objects into FLI data objects, which are then passed to the foreign language, such as C. Similarly, data can be passed from C or the Windows functions to the FLI, and then translated into a suitable Lisp form. The FLI types can therefore best be seen as an intermediate stage in the passing of data between Lisp and other languages.

Here are some of the features and sorts of foreign types:

- Consistency — Foreign types behave in a consistent and predictable manner. There is only one definition for any given foreign type.
- Parameterized types — these can be created using a `deftype`-like syntax. The macro `define-foreign-type` provides a simple mechanism for creating parameterized types.
- Encapsulated types — the ability to define a new foreign type as an extension to an existing type definition is provided. All types are converters between Lisp and the foreign language. New types can be defined to add an extra level of conversion around an existing type. The macro `define-foreign-converter` and the foreign type `:wrapper` provide this functionality.
- Generalized accessors — the FLI does not create named accessors. Instead, several generalized accessors use information stored within the foreign type in order to access the foreign object. These accessors are `foreign-slot-value`, `foreign-aref` and `dereference`. This makes it possible to handle type definitions corresponding to C types defined using unnamed structures, as we do not rely on specialized accessors for the given type. Also, there is `foreign-typed-aref` for efficient access in compiled code.
- Documentation for types — foreign type definitions can include documentation strings.
- Specialized type constructors — to make the definition of the Lisp to C interfaces even easier several type constructor macros are provided to mimic the C type constructors `typedef`, `enum`, `struct`, and `union`. The new FLI constructors are `define-c-typedef`, `define-c-enum`, `define-c-struct` and `define-c-union`. Note that the equivalent foreign types for most standard C types are already available within the FLI.
- Querying and testing functions — to get the byte size of a foreign type, use `size-of`. To test for equivalence of foreign types, use `foreign-type-equal-p`.

There are two fundamental sorts of FLI types: immediate and aggregate. Immediate types, which correspond to the C fundamental types, are so called because they are basic data types such as integers, booleans and bytes which have a direct representation in the computer memory. Aggregate types, which correspond to the C derived types, consist of a combination of immediate types, and possibly of smaller aggregate types. Examples of aggregate types are arrays and structures. Any user-defined type is an aggregate type.

2.1 Immediate types

The immediate types are the basic types used by the FLI to convert between Lisp and a foreign language.

2 FLI Types

2.1 Integral types

Integral types are the FLI types that represent integers. They consist of the following: `:int`, `:byte`, `:long`, `:short`, `:signed`, `:unsigned` and `:enum`, along with integer types converting to types with particular sizes defined by ISO C99 such as `:int8`, `:uint64` and `:intmax`.

Integral types can be combined in a list for readability and compatibility purposes with the foreign language, although when translated to Lisp such combinations are usually returned as a Lisp `integer`, or a `fixnum` for byte sized combinations. For example, a C `unsigned long` can be represented in the FLI as an `(:unsigned :long)`.

2.1.2 Floating point types

The FLI provides the several different immediate types for the representation of floating point numbers. They consist of the following: `:float`, `:double`, `:lisp-double-float`, `:lisp-float`, and `:lisp-single-float`. The floating types all associate equivalent Lisp and C types, except the `:lisp-float`, which can take a modifier to cause an association between different floating types. A `:lisp-float` associates a Lisp `float` with a C `float` by default, but a declaration of `(:lisp-float :double)` corresponds to a C `double`, for example.

Note: be sure to use `:language :ansi-c` when passing float arguments to and from C using `define-foreign-function` and so on.

2.1.3 Complex number types

The FLI provides two immediate types for the representation of complex numbers, named `:float-complex` and `:double-complex`, which correspond to the C types `float complex` and `double complex` respectively.

2.1.4 Character types

The FLI provides the `:char` type to interface a Lisp `character` with a C `char`.

2.1.5 Boolean types

The FLI provides the `:boolean` type to interface a Lisp boolean value (t or `nil`) with a C `int` (0 corresponding to `nil`, and any other value corresponding to `t`). The `:boolean` type can be modified to make it correspond with other C types. For example, `(:boolean :byte)` would associate a Lisp boolean with a C `byte`, and `(:boolean :long)` would associate a Lisp boolean with a C `long`. `(:boolean :standard)` would associate a Lisp boolean with a C99 _Bool.

2.1.6 Pointer types

Pointers are discussed in detail in 3 FLI Pointers. Further details can also be found in the reference entry for `:pointer`.

2.2 Aggregate types

Aggregate types are types such as arrays, strings and structures. The internal structure of an aggregate type is not transparent in the way that immediate types are. For example, two structures may have the same size of 8 bytes, but one might partition its bytes into two integers, whereas the other might be partitioned into a byte, an integer, and another byte. The FLI provides a number of functions to manipulate aggregate types. A feature of aggregate types is that they are usually accessed through the use of pointers, rather than directly.
2 FLI Types

2.2.1 Arrays
The FLI has two predefined array types: the :c-array type, which corresponds to C arrays, and the :foreign-array type. The two types are the same in all aspects but one: if you attempt to pass a :c-array by value through a foreign function, the starting address of the array is what is actually passed, whereas if you attempt to pass a :foreign-array in this manner, an error is raised.

For examples on the use of FLI arrays refer to :c-array and :foreign-array in 8 Type Reference.

2.2.2 Strings
The FLI provides two foreign types to interface Lisp and C strings, :ef-wc-string and :ef-mb-string.

The :ef-mb-string converts between a Lisp string and an external format C multi-byte string. A maximum number of bytes must be given as a limit for the string size.

The :ef-wc-string converts between a Lisp string and an external format C wide character string. A maximum number of characters must be given as a limit for the string size.

For more information on converting Lisp strings to foreign language strings see the string types :ef-mb-string, :ef-wc-string, and the string functions convert-from-foreign-string, convert-to-foreign-string, and with-foreign-string.

2.2.3 Structures and unions
The FLI provides the :struct and :union types to interface Lisp objects with the C struct and union types.

To define types to interface with C structures, the FLI macro define-c-struct is provided. In the next example it is used to define a FLI structure, tagpoint:

\[
\begin{align*}
(fli:define-c-struct & \text{tagpoint} \\
& (x : \text{long}) \\
& (y : \text{long}) \\
& (\text{visible} (:\text{boolean} :\text{byte}))
\end{align*}
\]

This structure would interface with the following C structure:

```c
typedef struct tagPOINT {
    LONG x;
    LONG y;
    BYTE visible;
} POINT;
```

The various elements of a structure are known as slots, and can be accessed using the FLI foreign slot functions foreign-slot-names, foreign-slot-type and foreign-slot-value, and the macro with-foreign-slots. For example, the next commands set point equal to an instance of tagPOINT, and set the Lisp variable names equal to a list of the names of the slots of tagPOINT.

\[
\begin{align*}
\text{(setq point (fli:allocate-foreign-object :type 'tagpoint))}
\end{align*}
\]

\[
\begin{align*}
\text{(setq names (fli:foreign-slot-names point))}
\end{align*}
\]

The next command finds the type of the first element in the list names, and sets the variable name-type equal to it.

\[
\begin{align*}
\text{(setq name-type (fli:foreign-slot-type point (car names)))}
\end{align*}
\]
2 FLI Types

Finally, the following command sets `point-to` equal to a pointer to the first element of `point`, with the correct type.

```
(setq point-to (fli:foreign-slot-pointer point (car names) :type name-type))
```

The above example demonstrates some of the functions used to manipulate FLI structures. The FLI `:union` type is similar to the `:struct` type, in that the FLI slot functions can be used to access instances of a union. The convenience FLI function `define-c-union` is also provided for the definition of specific union types.

2.2.4 Vector types

Vector types are types that correspond to C vector types. These are handled by the C compiler in a special way, and therefore when you pass or return them to/from foreign code by value you must declare them correctly.

2.2.4.1 Vector type names

The names of the FLI types are designed to best match the types that are defined by Clang, which is used on macOS, iOS and FreeBSD and is optionally available on other operating systems. For every C/Objective-C type of the form `vector_<type><count>`, there is an FLI type of the form `fli:vector-<scalar fli type><count>`. For example, the C/Objective-C type `vector_double8` is matched by the FLI type `fli:vector-double8`.

The scalar fli types and their matching Common Lisp types are:

- `char` (signed-byte 8)
- `uchar` (unsigned-byte 8)
- `short` (signed-byte 16)
- `ushort` (unsigned-byte 16)
- `int` (signed-byte 32)
- `uint` (unsigned-byte 32)
- `long` (signed-byte 64)
- `ulong` (unsigned-byte 64)
- `float` `single-float`
- `double` `double-float`

The count can be 2, 3, 4, 8, 16 (for elements of 32 bits or less) or 32 (for elements of 16 bits or less). The restrictions mean that the maximum size of a vector is 64 bytes and the maximum count is 32.

Note that `long` and `ulong` are always 64 bits in this context, even on 32-bit where the C type `long` is 32 bits.

The full list of types:

- `vector-char2` `vector-char3` `vector-char4` `vector-char8` `vector-char16` `vector-char32`
- `vector-uchar2` `vector-uchar3` `vector-uchar4` `vector-uchar8` `vector-uchar16` `vector-uchar32`
- `vector-ushort2` `vector-ushort3` `vector-ushort4` `vector-ushort8` `vector-ushort16` `vector-ushort32`
- `vector-int2` `vector-int3` `vector-int4` `vector-int8` `vector-int16`
- `vector-ulong2` `vector-ulong3` `vector-ulong4` `vector-ulong8` `vector-ulong16`
In addition, `vector-long1` and `vector-ulong1` are defined as immediate 64-bit signed and unsigned integers, because Clang defines them like that.

### 2.2.4.2 Vector type values

When passing an argument that is declared as any of the FLI vector types, the value needs to be a Lisp vector of the correct length or a foreign pointer to the FLI vector type.

- For `vector-double<count>` and `vector-float<count>`, the Lisp vector must either have element type `double-float` or `single-float`, or have element type `t` and contain elements of type `float`.
- For the integer vector types, the Lisp vector must either have an element type that is subtype of the element type of the FLI vector type, or have element type `t` and contain elements that fit into the FLI vector.
- If a foreign pointer is passed for an argument that is declared as a FLI vector type, it must point to an object of the FLI vector type, which must be an exact match, including being correctly signed. The vector is passed by value, not as a pointer.

When a FLI vector type is passed into Lisp, either because it is a returned value from a foreign function or an argument to a foreign callable, it is automatically converted to a Lisp vector of the correct length and element type. This also occurs when accessing a value using `foreign-slot-value`, `foreign-aref` and `dereference`.

### 2.2.4.3 Using a foreign pointer to a vector type

When you have a foreign pointer to a vector type, you can access individual elements using `foreign-aref`, or convert the vector into a Lisp vector using `dereference`. The reverse operations can be performed using the `setf` form or `foreign-aref` and `dereference`. For example:

```lisp
(let ((d4-poi (fli:allocate-foreign-object
               :type 'fli:vector-double4)))
  (setf (fli:dereference d4-poi) #(0d0 1d0 2d0 3d0))
  (format t "Collected values: ~s~%"
         (loop for x below 4
               collect (fli:foreign-aref d4-poi x))))
  (setf (fli:foreign-aref d4-poi 3) -3d0)
  (format t "Dereference after setf: ~s~%"
         (fli:dereference d4-poi)))
=>
Collected values: (0.0D0 1.0D0 2.0D0 3.0D0)
Dereference after setf: #(0.0D0 1.0D0 2.0D0 -3.0D0)
```

Normally there is no reason to allocate a foreign object for a vector type as in the example above. You would, however, encounter such a pointer if you have foreign code that calls into Lisp passing it an argument that is a pointer to a vector type, and your Lisp code needs to set the values in it. In this case, you will need to declare the argument type as `(:pointer vector-double4)` and then set it like this:

```lisp
(fli:define-foreign-callable my-callable
  ((d4-poi (:pointer fli:vector-double4)))
  (let ((lisp-v4 (my-compute-d4-values)))
    (setf (fli:dereference d4-poi) lisp-v4)))
```
Note that if you call a function that takes a pointer to a vector type, you can use the FLI types :reference, :reference-pass and :reference-return to pass and return values without having to explicitly allocate a foreign pointer. For example, if the C function my_function takes a pointer to vector_double2 and fills it like this:

```c
void my_function (vector_double2* d2_poi) {
    (*d2_poi)[0] = 3.0;
    (*d2_poi)[1] = 4.0;
}
```

then in Lisp you can call it by:

```lisp
(defun my-compute-d4-values ()
     (vector 3.5d0 7d0 9d23 0.1d0)))
```

```lisp
(fli:define-foreign-function my-function
     ((d2-poi (:reference-return fli:vector-double2))))

(my-function) ; returns #(3D0 4D0)
```

2.2.4.4 Notes on foreign vector types

C compilers other than Clang can also define vector types in various ways:

- In GCC, they can be defined using the `vector_size` attribute, for example, `vector_double4` would be defined by:

  ```c
typedef double vector_double4 __attribute__ ((vector_size (32)));
  ```

  Note that the size is in bytes, rather than an element count.

- The compiler supplied by ARM has "vector data types", so for example the type `float32x4_t` matches `vector-float4`.

- In Clang, it is possible to define vector types using the GCC syntax, OpenCL syntax, Altivec syntax and Neon syntax.

On 32-bit x86, vector types can be passed either with or without using SSE2. The Lisp FLI definitions must pass/receive arguments in the same way as the C compiler that was used to compile the foreign code. On macOS, this is always with SSE2, so this is not an issue, but on other platforms (Linux, FreeBSD, Solaris) the situation is not clear. What the Lisp definitions do is controlled by `*use-sse2-for-ext-vector-type*`.

When using `vector-char2` and `vector-uchar2` on x86_64 platforms and the C compiler is Clang or a derivative, you need to check that you have the latest version of the C compiler, because earlier versions of Clang compiled these types differently from later versions. This affects macOS too because the Xcode C compiler is based on Clang. You can check the version of the C compiler by executing `cc -v` in a shell. On macOS, you need to check that you have LLVM 8.0 or later. If you have Clang, you need to check that you have version 3.9 or later.

On macOS x86_64, the treatment of `vector_char2` and `vector_uchar2` changed between LLVM 6.0 and 8.0. LispWorks is compatible with LLVM 8.0. You can check which version of LLVM you have by executing `cc -v` in a shell.

When a structure is passed by value and it contains one or more fields whose types are vector types, it is also important to declare the type correctly in Lisp, otherwise the wrong data may be passed. That is because the machine registers that are used to pass such structures may be different from the registers that are used to pass seemingly equivalent structures that are defined without vector types. Such structures are commonly used to represent matrices.
2 FLI Types

2.3 Parameterized types

The `define-foreign-type` and `define-foreign-converter` macros allow the definition of parameterized types. For example, assume you want to create a foreign type that matches the Lisp type `unsigned-byte` when supplied with an argument of one of 8, 16, or 32. The following code achieves this:

```lisp
(fli:define-foreign-type unsigned-byte (&optional (bitsize '*))
  (case bitsize
    (8 ':unsigned :byte)
    (16 ':unsigned :short)
    (32 ':unsigned :int)
    (otherwise (error "Illegal foreign type (~s ~s)" 'unsigned-byte bitsize))))
```

This defines the new foreign type `unsigned-byte` that can be used anywhere within the FLI as one of:

- `(unsigned-byte 8)`
- `(unsigned-byte 16)`
- `(unsigned-byte 32)`

Specifying anything else returns an error.

2.4 Encapsulated types

With earlier version of the foreign function interface it was not possible to create new foreign types that encapsulated the functionality of existing types. The only way in which types could be abstracted was to create "wrapper" functions that filtered the uses of a given type. The FLI contains the ability to encapsulate foreign types, along with the ability to create parameterized types. This enables you to easily create more advanced and powerful type definitions.

2.4.1 Passing Lisp objects to C

There are occasions when it is necessary to pass Lisp object references through to C and then back into Lisp again. An example of this is the need to specify Lisp arguments for a GUI action callback.

Using either the foreign type `:wrapper` or the macro `define-foreign-converter` a new foreign type can be created that wraps an extra level of conversion around the Lisp to C or C to Lisp process.

2.4.2 An example

For example, let us assume that we want to pass Lisp object handles through to C and then back to Lisp again. Passing C a pointer to the Lisp object is not sufficient, as the Lisp object might be moved at any time, for example due to garbage collection. Instead, we could assign each Lisp object to be passed to C a unique `int` handle. Callbacks into Lisp could then convert the handle back into the Lisp object. This example is implemented in two ways: using the `:wrapper` type and using `define-foreign-converter`.

The `:wrapper` foreign type allows the specification of automatic conversion functions between Lisp and an instance of a FLI type. Its signature is:

```
:wrapper fli-type &key lisp-to-foreign foreign-to-lisp
```

Using `:wrapper` we can wrap Lisp to C and C to Lisp converters around the converters of an existing type:

```lisp
(fli:define-foreign-type lisp-object-wrapper ()
  "A mechanism for passing a Lisp object handle to C.
```

23
2 FLI Types

Underlying C type is Lint"
`(:wrapper :int
 :lisp-to-foreign find-index-for-object
 :foreign-to-lisp find-object-from-index))

If the :lisp-to-foreign and :foreign-to-lisp keyword arguments are not specified, no extra conversion is applied to the underlying foreign type, causing it to behave like a standard :int type.

See the reference entry for :wrapper for more examples.

A second method uses define-foreign-converter, which is specifically designed for the creation of new converter types (that is, types which wrap extra levels of conversion around existing types). A simple use of define-foreign-converter is to only wrap extra levels of conversion around existing Lisp to foreign and foreign to Lisp converters.

(fli:define-foreign-converter lisp-object-wrapper () object
 :foreign-type :int
 :lisp-to-foreign `(find-index-for-object ,object)
 ;; object will be the Lisp Object
 :foreign-to-lisp `(find-object-from-index ,object)
 ;; object will be the :int object
 :documentation "Foreign type for converting from Lisp objects to integers handles to Lisp objects which can then be manipulated in C. Underlying foreign type : 'C' int")

The definition of lisp-object-wrapper using define-foreign-converter is very similar to the definition using :wrapper, and indeed the :wrapper type could be defined using define-foreign-converter.

See the reference entry for define-foreign-converter for more information.

2.5 The void type

The FLI provides the :void type for interfacing with the C void type. In accordance with ANSI C, it behaves like an unsigned char. In practice you will probably want to interface with a C void *, for which you should use the FLI construction (:pointer :void).

For an example of interfacing to a void **, see 3.5.2 Allocating a pointer to a pointer to a void.

2.6 Summary

In this chapter the various FLI data types have been examined. FLI types perform a translation on data passed between Lisp objects and C objects, and there are two main sorts of FLI types: immediate and aggregate. Immediate types have a simple representation in computer memory, and represent objects such as integers, floating point number and bytes. Aggregate types have a more complicated structure in memory, and consist of structures, arrays, strings, and unions. Parameterized and encapsulated types were also discussed. Finally, a number of FLI types that perform specific functions, such as the :void type and the :wrapper type, were examined.
3 FLI Pointers

Pointers are a central part of the C type system, and because Lisp does not provide them directly, one of the core features of the FLI is a special pointer type that is used to represent C pointers in Lisp. This chapter discusses how to use FLI pointers by examining some of the functions and macros which allow you to create and manipulate them.

A FLI pointer is a FLI object containing a memory address and a type specification. The implication is that the pointer points to an object of the type specified at the memory address, although a pointer can point to a memory location not containing an allocated FLI object, or an object that was allocated with a different type. Pointers can also point to other pointers, and even to functions.

3.1 Creating and copying pointers

This section discusses how to create a FLI pointer, how to copy it, and where the memory is actually allocated.

3.1.1 Creating pointers

Many FLI functions when called return a pointer to the object created. For example, a form such as:

```lisp
(fli:allocate-foreign-object :type :int)
```

will return something similar to the following:

```lisp
#<Pointer to type :INT = #x007608A0>
```

This is a FLI pointer object, pointing to an object at address #x007608A0 of type :int. Note that the memory address is printed in hexadecimal format, but when you use the FLI pointer functions and macros discussed in this chapter, numeric values are interpreted as base 10 unless you use Lisp reader syntax such as #x.

To use the pointer in the future it needs to be bound to a Lisp variable. This can be done by using `setq`.

```lisp
(setq point1 (fli:allocate-foreign-object :type :int))
```

A pointer can be explicitly created, rather than being returned during the allocation of memory for a FLI object, by using `make-pointer`. In the next example a pointer is made pointing to an :int type at the address 100, and is bound to the Lisp variable `point2`.

```lisp
(setq point2 (fli:make-pointer :address 100 :type :int))
```

For convenience you may wish to define your own pointer types, for example:

```lisp
(fli:define-foreign-pointer my-pointer-type :int)

(setq point3
     (fli:make-pointer :address 100
                        :pointer-type 'my-pointer-type))
```

point3 contains the same type and address information as point2.

A pointer which holds the address of a foreign symbol, either one which is defined in foreign code or one that is defined in
Lisp using `define-foreign-callable`, can be created either by `make-pointer` with :symbol-name or `foreign-function-pointer`.

### 3.1.2 Copying pointers

Suppose the Lisp variable `point3` is bound to a FLI pointer as in 3.1.1 Creating pointers. To make a copy of the pointer it is not sufficient to do the following:

```
(setq point4 point3)
```

This simply sets `point4` to contain the same pointer object as `point3`. Thus if the pointer is changed using `point3`, a similar change is observed when looking in `point4`. To create a distinct copy of the pointer object you should use `copy-pointer`, which returns a new pointer object with the same address and type as the old one, as the following example shows.

```
(setq point5 (fli:copy-pointer point3))
```

### 3.1.3 Allocation of FLI memory

Foreign objects do take up memory. If a foreign object is no longer needed, it should be deallocated using `free-foreign-object`. This should be done only once for each foreign object, regardless of the number of pointer objects that contain its address. After freeing a foreign object, any pointers or copies of pointers containing its address will give unpredictable results if the memory is accessed.

FLI memory is allocated using `malloc()` so it comes from the C heap.

The FLI pointer object itself is a Lisp object, but the memory it points to does not show up in the output of `room`. Therefore you must use Operating System tools to see the virtual address size of the program.

### 3.2 Pointer testing functions

A number of functions are provided for testing various properties of pointers. The most basic, `pointerp`, tests whether an object is a pointer. In the following examples the first expression returns `nil`, because 7 is a number, and not a pointer. The second returns `t` because `point4` is a pointer.

```
(fli:pointerp 7)
(fli:pointerp point4)
```

The address pointed to by a pointer is obtained using `pointer-address`. For example, the following expression returns the address pointed to by `point4`, which was defined to be `100`.

```
(fli:pointer-address point4)
```

Pointers which point to address 0 are known as null pointers. Passing the Lisp object `nil` instead of a pointer results in `nil` being treated as a null pointer. The function `null-pointer-p` tests whether a pointer is a null pointer or not. If the pointer is a null pointer the value `t` is returned. We know that `point4` points to address `100` and is therefore not a null pointer. As a result, the following expression returns `nil`.

```
(fli:null-pointer-p point4)
```

Another testing function is `pointer-eq` which returns `t` if two pointers point to the same address, and `nil` if they do not. In

---

26
the previous section we created point3 by making a copy of point1, and so both point to the same address. Therefore the following expression returns t.

\[(\text{fli:} \text{pointer-eq point1 point3})\]

Two functions are provided to return information about the object pointed to by a pointer, `pointer-element-type` and `pointer-element-size`. In practice, it is the pointer which holds the information as to the type of the object at a given memory location—the memory location itself only contains data in the form of bytes. Recall that point1 was defined in the previous section as a pointer to an `:int`. As a result the following two lines of code return 4 (the size of an `:int`) and `:int`.

\[(\text{fli:} \text{pointer-element-size point1})\]

\[(\text{fli:} \text{pointer-element-type point1})\]

The question of pointer types is discussed further in the next section.

### 3.3 Pointer dereferencing and coercing

The `dereference` function returns the value stored at the location held by a pointer, provided the type of the object is an immediate type and not a structure or an aggregate type. For now, you can consider immediate data types to be the simple types such as `:int`, `:byte`, and `:char`, and aggregate types to consist of structures defined using `:struct`. Full details about types are given in 2 FLI Types, and the use of the `dereference` function with aggregate types is discussed further in 5 Advanced Uses of the FLI.

The `dereference` function supports the `setf` function which can therefore be used to set values at the address pointed to by the pointer. In the following example an integer is allocated and a pointer to the integer is returned. Then `dereference` and `setf` are used to set the value of the integer to 12. Finally, the value of the integer is returned using `dereference`.

\[(\text{setq point5 (fli:} \text{allocate-foreign-object :type :int))}\]

\[(\text{setf (fli:} \text{dereference point5) 12)}\]

\[(\text{fli:} \text{dereference point5)}\]

The function `dereference` has an optional `:type` keyword which can be used to return the value pointed to by a pointer as a different type. This is known as coercing a pointer. The default value for `:type` is the type the pointer is specified as pointing to. In the next example the value at `point5` is returned as a Lisp boolean even thought it was set as an `:int`. Because the value at `point5` is not 0, it is returned as `t`.

\[(\text{fli:} \text{dereference point5 :type '(boolean :int))}\]

Recall that at the end of the previous section the function `pointer-element-type` was demonstrated. What follows is an example which uses this function to clarify the issue of pointers and types.

The first action consists of allocating an integer, and setting up a pointer to this integer:

\[(\text{setq pointer-a (fli:} \text{allocate-foreign-object :type :int))}\]

Now we use `fli:copy-pointer` to make a copy of `pointer-a`, but with the type of the new pointer changed to be a `:byte`. We call this pointer `pointer-b`.

\[(\text{setq pointer-b (fli:copy-pointer pointer-a :type :byte))}\]
3 FLI Pointers

We now have two pointers which point to the same memory location, but one thinks it is pointing to an :int, and the other thinks it is pointing to a :byte. Test this by using the following two commands:

(fli:pointer-element-type pointer-a)

(fli:pointer-element-type pointer-b)

Similar commands using pointer-element-size show that pointer-a is pointing to an element of size 4, and pointer-b to an element of size 1.

So far we have seen the use of the :type keyword to specify how to set up or dereference a pointer to obtain values in the format we want. There is, however, a further level of abstraction in pointer typing which uses the :pointer-type keyword instead of the :type keyword.

The following two commands produce identical pointers, but one uses the :type keyword, and the other uses the :pointer-type keyword:

(fli:make-pointer :address 0 :type :int)

(fli:make-pointer :address 0 :pointer-type '(:pointer :int))

In the instance above there is no advantage in using the :pointer-type option. However, :pointer-type can be very useful when used in combination with a defined type, as the next example shows.

Imagine you are writing a program with many statements creating pointers to a certain type, for example :byte, and this is done using the :type keyword. If half way through coding the type to be pointed to was changed to a :char, every individual statement would need to be changed. However, if a general pointer type had been defined at the start and all the statements had used the :pointer-type keyword to refer to that particular type, only one statement would need to be changed: the initial definition of the pointer type. The following code illustrates this:

(fli:define-c-typedef my-pointer-type (:pointer :byte))

(fli:make-pointer :address 0 :pointer-type 'my-pointer-type)

...  
(fli:make-pointer :address 100 :pointer-type 'my-pointer-type)

The above code consists of a definition of a new pointer type, called my-pointer-type, which points to a :byte. Following it are one hundred lines of code using my-pointer-type. If you decide that all the pointers made should actually point to a :char, only the first line needs to be changed, as shown below:

(fli:define-c-typedef my-pointer-type (:pointer :char))

The program can now be re-compiled. The use of :pointer-type with pointers is thus analogous to the use of constants instead of absolute numbers in programming.

The function pointer-pointer-type returns the pointer type of a foreign pointer.

3.4 An example of dynamic pointer allocation

When a pointer is created, using make-pointer, or due to the allocation of a foreign object, memory is put aside to store the details of the pointer. However, if a pointer is only needed within the scope of a particular section of code, there is a FLI macro, with-coerced-pointer, which can be used to create a temporary pointer which is automatically deallocated at the end of the code. The next example illustrates the use of this macro.
To start with, we need an object to use the temporary pointer on. The following code allocates ten consecutive integers, and sets their initial values.

```
(setf array-obj
     (fli:allocate-foreign-object :type :int
      :nelems 10
      :initial-contents
      '(0 1 2 3 4 5 6 7 8 9)))
```

When the ten integers are created, `allocate-foreign-object` returns a pointer to the first one. The next piece of code uses `with-coerced-pointer` to create a copy of the pointer, which is then used to print out the contents of the ten integers. At the end of the printing, the temporary pointer is automatically deallocated.

```
(fli:with-coerced-pointer (temp) array-obj
  (dotimes (x 10)
    (print (fli:dereference temp))
    (fli:incf-pointer temp)))
```

The above example also illustrates the use of the `incf-pointer`, which increases the address stored in a pointer by the size of the object pointed to. There is a similar function called `decf-pointer`, which decreases the address held by a pointer in a similar fashion.

### 3.5 More examples of allocation and pointer allocation

The functions `allocate-dynamic-foreign-object`, `allocate-foreign-object`, `alloca`, and `malloc` can take the keyword arguments `:type` and `:pointer-type`. It is important to understand the difference between these two arguments.

The `:type` argument is used to specify the name of the FLI type to allocate. Once such an object has been allocated a foreign pointer of type `(:pointer type)` is returned, which points to the allocated type. Without this pointer it would not be possible to refer to the object.

The `:pointer-type` argument is used to specify a FLI pointer type. If it is used then the value `pointer-type` should be of the form `(:pointer type)` or be defined as a FLI pointer type. The function then allocates an object of type `type`, and a pointer to the object of type `type` is returned.

#### 3.5.1 Allocating an integer

To allocate an integer in C:

```
(int *)malloc(sizeof(int))
```

You can allocate the integer from LispWorks using the `:type` argument:

```
(fli:allocate-foreign-object :type :int)
=> #<Pointer to type :INT = #x007E1A60>
```

Alternatively you can allocate the integer from LispWorks using the `:pointer-type` argument:

```
(fli:allocate-foreign-object
  :pointer-type '(:pointer :int))
=> #<Pointer to type :INT = #x007E1A60>
```
3.5.2 Allocating a pointer to a pointer to a void

Suppose you need to call a C function that takes a `void **` argument, defined as follows:

```c
struct arg_struct
{
    int val;
};

void func_handle_init(void **h)
{
    struct arg_struct *handle = NULL;
    handle = (struct arg_struct *)malloc(sizeof(struct arg_struct));
    memset(handle, 0, sizeof(struct arg_struct));
    handle->val = 12;
    *h = handle;
}
```

With this foreign function definition:

```lisp
(fli:define-foreign-function
 (func-handle-init "func_handle_init"
 :source)
   ((handle (:pointer (:pointer :void))))
 :result-type :void
 :language :ansi-c)
```

you could simply do:

```lisp
(setq handle
      (fli:allocate-foreign-object :type :pointer))
(fli:free-foreign-object handle)
```

but do not forget to also free the pointer:

```lisp
(fli:free-foreign-object handle)
```

Another approach is to allocate the pointer on the stack. In this case you do not need to free it explicitly:

```lisp
(fli:with-dynamic-foreign-objects ((handle :pointer))
  (func-handle-init handle))
```

Yet another approach is to define the foreign function like this:

```lisp
(fli:define-foreign-function
 (func-handle-init "func_handle_init"
 :source)
   ([:ignore (:reference-return (:pointer :void))])
 :result-type :void
 :language :ansi-c)
```

Then call the function like this:

```lisp
(func-handle-init)
```

and it will return the handle. This works because the `:reference-return` type allocates the temporary `void **` within the function and returns its contents.
3.6 Summary

In this chapter the use of FLI pointers was examined. A number of FLI functions useful for copying, creating and testing the properties of a pointer were presented. The use of the `dereference` function for obtaining the value pointed to by a pointer was examined, as was the coercing of a pointer—namely dereferencing a pointer to an object in a manner which returns the value found there as a different type. Finally, an example of the use of the `with-coerced-pointer` macro was given to illustrate the use of temporary pointers for efficient memory management.

In the next chapter some advanced topics of the FLI are examined in greater detail.
4 Defining foreign functions and callables

This chapter discusses how to define foreign functions and callables.

4.1 Foreign callables and foreign functions

The two main macros for interfacing LispWorks with a foreign language are `define-foreign-callable` which defines Lisp functions that can be called from the foreign language, and `define-foreign-function` which defines a short linking function that can call functions in a foreign language.

In 1 Introduction to the FLI we defined a foreign function for calling the Win32 function `SetCursorPos`. The code for this example is repeated here.

```lisp
(fli:define-foreign-function (set-cursor-position "SetCursorPos")
  ((x :long) (y :long))
  :result-type :boolean)
```

A FLI foreign function calling some C code. is an illustration of `set-cursor-position`, represented by a square, calling the C code which constitutes `SetCursorPos`.

A FLI foreign function calling some C code.

The next diagram, C calling a callable function in Lisp., illustrates a callable function. Whereas a foreign function consists of a Lisp function name calling some code in C, a callable function consists of Lisp code, represented by an oval in the diagram, which can be called from C.

C calling a callable function in Lisp.

Callable functions are defined using `fli:define-foreign-callable`, which takes as its arguments, amongst other things, the name of the C function that will call Lisp, the arguments for the callable function, and a body of code which makes up the callable function.

To call a Lisp function from C or C++ you need to define it using `fli:define-foreign-callable`. Then call `fli:make-pointer` with the `:symbol-name` argument and pass the result to C or C++ as a function pointer.
For the purpose of creating a self-contained illustration in Lisp, the following Lisp code defines a foreign callable function that takes the place of the Windows function `SetCursorPos`.

```lisp
(fli:define-foreign-callable ("SetCursorPos"
    :result-type :boolean)
  ((x :long) (y :long))
  (capi:display-message
   "The cursor position can no longer be set"))
```

Supposing you had the above foreign callable defined in a real application, you would use:

```lisp
(make-pointer :symbol-name "SetCursorPos")
```

to create a foreign pointer which you pass to foreign code so that it can call the Lisp definition of `SetCursorPos`.

**A FLI foreign function calling a callable function.** illustrates what happens when `set-cursor-position` is called. The foreign function `set-cursor-position` (represented by the square) calls what it believes to be the Windows function `SetCursorPos`, but the callable function (represented by the oval), also called `SetCursorPos`, is called instead. It pops up a CAPI pane displaying the message "The cursor position can no longer be set".

A FLI foreign function calling a callable function.

For more information on calling foreign code see `define-foreign-function`.

For more information on defining foreign callable functions see 4.1.1 Strings and foreign callables and `define-foreign-callable`.

For information on how to create a LispWorks DLL, see "Creating a dynamic library" in the *LispWorks® User Guide and Reference Manual*.

For some complete examples of building a LispWorks DLL, then loading and calling it from foreign code, see "Delivering a dynamic library" in the *Delivery User Guide*.

### 4.1.1 Strings and foreign callables

To interface to a C function which takes a pointer to a string `form` and puts a string in the memory pointed to by `result`, declared like this:

```c
void evalx(const char *form, char *result);
```

you would define in Lisp:

```lisp
(fli:define-foreign-function evalx
  ((form (:reference-pass :ef-mb-string))
   (:ignore (:reference-return
     (:ef-mb-string :limit 1000))))
```

and call:
Now suppose instead that you want your C program to call a similar routine in a LispWorks for Windows DLL named "evaluator", like this:

```c
typedef void (_stdcall *evalx_func_type)(const char *form, char *result);
HINSTANCE dll = LoadLibrary("evaluator");
evalx_func_type evalx = (evalx_func_type) GetProcAddress(dll, "evalx");
char result[1000];
evalx("(+ 2 3)", result);
printf("%s\n", result);
```

You would put this foreign callable in your DLL built with LispWorks:

```lisp
(fli:define-foreign-callable
 ("evalx" :calling-convention :stdcall)
 ((form (:reference :ef-mb-string
 :lisp-to-foreign-p nil
 :foreign-to-lisp-p t))
   :limit 1000)
  :multiple-value-bind (res err)
    (ignore-errors (read-from-string form)))
 (setq result
   (if (not (fixnum err))
     (format nil "Error reading: ~a" err)
     (multiple-value-bind (res err)
       (ignore-errors (eval res))
       (if (and (not res) err)
         (format nil "Error evaluating: ~a" err)
         (princ-to-string res)))))))
```

Note: you could use :reference-return and :reference-pass in the foreign callable definition, but we have shown :reference with explicit lisp-to-foreign-p and foreign-to-lisp-p arguments to emphasise the direction of each conversion.

### 4.2 Specifying a calling convention.

The FLI macros such as `define-foreign-function` and `define-foreign-callable` take a keyword :calling-convention. Apart from on 32-bit Windows and on the ARM architectures, there is only one calling convention and in most cases you do not need to specify it.

The common case when you need to specify the calling convention is on 32-bit Windows where the default LispWorks calling convention is __stdcall. This matches the Win32 API functions, but compilers typically produce __cdecl by default (which is the same as the non-Windows x86 systems).

ARM (both 32-bit and 64-bit) also has more than one calling convention, but it should be rare (in 32-bit) or extremely rare (in 64-bit) that you need to specify the convention. Note however that, on ARM, failing to specify that a function is variadic (by the keyword :variadic-num-of-fixed) is more likely to cause crashes than on the other architectures.
4.2.1 Windows 32-bit calling conventions

The Win32 API functions in 32-bit Windows applications are compiled using the __stdcall calling convention, but compilers normally use __cdecl by default. Thus if you call functions that are not part of the Win32 API from 32-bit LispWorks then you need to check the calling convention and in most cases you need to specify it as __cdecl by passing :calling-convention :cdecl. To specify __stdcall, pass :calling-convention :stdcall, which is the default so is not really needed.

Note that all the other LispWorks architectures, including 64-bit Windows, interpret both :cdecl and :stdcall to mean the default.

Since whole libraries are normally compiled with the same calling convention, it is usually convenient to define your own defining macro that expands to the FLI defining macro and passes it the calling convention. For example, LispWorks itself uses the following defining macro to define foreign calls to the MySQL library:

```
(defmacro def-mysql-function (&body x)
  `(dspec:def (def-mysql-function ,(car x))
   (define-foreign-function ,@x
     :module 'mysql-library
     :calling-convention :cdecl)))
```

4.2.2 ARM 32-bit calling conventions

32-bit ARM systems have two calling conventions: hard float and soft float. These calling conventions are binary incompatible, and operating systems generally support only one or the other. Currently, Android and iOS are both soft float but Android is now starting to support hard float code, while ARM Linux distributions are now almost always hard float, but used to be soft float. Moreover, iOS has a calling convention which is soft, and somewhat different from the Android/old-Linux soft float, so these are also binary incompatible.

Thus LispWorks supports 3 calling conventions:

Soft float conventions:

iOS The calling convention that is used by iOS.
soft Linux The calling convention that is used by Android, and was used by old Linux systems.

Hard float convention:

hard float The calling convention used by newer Linux systems.

When LispWorks compiles a foreign call or callable function, it (by default) generates "tri-compatible" code that can interface with either hard float, soft Linux or iOS foreign code. At run time, the code checks an internal flag and uses the appropriate calling convention. The internal flag is set to the correct value on start-up. The tri-compatible code is needed only for functions where the calling conventions differ, and when 2 or more of the conventions need the same code LispWorks avoids duplicating code, while remaining compatible with all 3 conventions.

Because of the tri-compatible code, LispWorks binaries (fasl files) are compatible with all the conventions. The compiled Lisp code is also compatible with all conventions. However, LispWorks executables (including LispWorks as a shared library) have a small C program that starts Lisp (the "xstarter"), and this is either hard float, soft Linux or iOS. Therefore, a LispWorks executable can run only on one calling convention, but the code that LispWorks compiles can run on all of them.

In particular, that means that it is possible to compile and build runtimes for Android and iOS on either soft float or hard float systems, because the runtime is created using the appropriate xstarter for the target OS.

It is possible to tell LispWorks to compile a foreign call or callable function for only one calling convention, by supplying the keyword :calling-convention with one of these values:
4 Defining foreign functions and callables

:ios
iOS.

:hard-float
hard float.

:soft-linux
soft Linux.

:android
Android. Currently that is an alias to :soft-linux.

:soft-float
Code that selects between :soft-linux and :ios.

All other values generate tri-compatible code.

You are only required to pass :calling-convention when you use a library with a calling convention that does not match the calling convention of the OS. That should be rare.

Passing :calling-convention also makes the code smaller and slightly faster, but the difference is unlikely to be significant.

Note that variadic functions (for example printf and scanf) are always soft float, which means that when compiling calls to such functions it is essential to specify that they are variadic (by passing :variadic-num-of-fixed) to ensure that LispWorks does not try to pass the arguments as hard float.

Compatibility note: in LispWorks 7.0, you had to pass :calling-convention :soft-float for variadic functions. This still works, but passing :variadic-num-of-fixed is more correct and will make it work properly on other architectures, (in particular 64-bit ARM).

4.2.3 ARM 64-bit calling conventions

There is a standard calling convention for 64-bit ARM (documented by ARM), but iOS uses a slightly different one. Therefore, there are effectively two calling conventions: the standard one and iOS.

By default, LispWorks compiles code that selects which convention to use at run time. However, the difference between the conventions is quite minor and affects only a small number of functions, so the code is the same for most functions. Thus the overhead is quite small and you will not normally have a reason to pass :calling-convention for 64-bit ARM.

You can use the following values with :calling-convention to tell LispWorks to compile for a specific convention:

:ios
Compile only the iOS convention.

:standard
Compile only the standard convention.

Other values are treated as the default.

Note that all the keywords used for 32-bit ARM (see 4.2.2 ARM 32-bit calling conventions), with the exception of :ios, are treated as the default on 64-bit ARM.

4.2.4 Fastcall on 32-bit x86 platforms

On 32-bit x86 platforms, the C compilers have a fastcall calling convention. In Visual C and the GNU C compiler, this it is specified by the __fastcall qualifier. If you call a foreign function that is compiled as a fastcall, you must specify the calling convention :fastcall.

On other architectures, the calling convention :fastcall is quietly ignored, and the code produced is the same as would be produced without it.

The calling convention :fastcall cannot be used in foreign callables (calls from foreign code into LispWorks).
5 Advanced Uses of the FLI

Note: Some of the examples in this chapter only work for LispWorks for Windows.

This is the final chapter of the user guide section of this manual. It provides a selection of examples which demonstrate some of the more advanced uses of the FLI.

5.1 Passing a string to a Windows function

The following example shows how to define a Lisp function which calls a Win32 API function to change the title of the active window. It demonstrates the use of define-foreign-function and with-foreign-string to pass a Lisp string to a Windows function.

The first step involves defining a FLI type to correspond to the Windows hwnd type, which is the window handle type.

(fli:define-c-typedef fli-hwnd
 (:unsigned :long))

The next step consists of the foreign function definitions. The first foreign function returns the window handle of the active window, by calling the Windows function GetActiveWindow. It takes no arguments.

(fli:define-foreign-function (get-act-window "GetActiveWindow")
 ()
 :result-type fli-hwnd
 :documentation "Returns the window handle of the active window for the current thread. If no active window is associated with the current thread then it returns 0.")

The next foreign function uses the Windows function SetWindowText to set the text of the active window titlebar. It takes a window handle and a pointer to a FLI string as its arguments.

(fli:define-foreign-function (set-win-text "SetWindowText" :dbcs)
 ((hwnd fli-hwnd)
  (lpstring :pointer))
 :result-type :boolean
 :documentation "Sets the text of the window titlebar.")

The foreign function set-win-text returns a boolean to indicate whether it has successfully changed the title bar.

The required FLI data types and foreign functions have been defined. What is now required is a Lisp function which uses them to change the titlebar of the active window. The next function does this:

(defun set-active-window-text (new-text)
 (let ((active-window (get-act-window))
   (external-format (if (string= (software-type) "Windows NT")
                      :unicode
                      :ascii)))
  (unless (zerop active-window)
    (fli:with-foreign-string (new-ptr element-count byte-count
                             :external-format external-format)
      new-text
      (declare (ignore element-count byte-count)))
The function `set-active-window-text` takes a Lisp string as its argument, and does the following:

1. It calls the foreign function `get-act-window` to set the variable `active-window` to be the handle of the active window. If no window is active, this will be zero.

2. The variable `external-format` is set to be `:unicode` if the operating system is Windows NT or a later system based on it (which expects strings to be passed to it in Unicode format), otherwise it is set to be `:ascii`.

3. If `active-window` is zero, then there is no active window, and the function terminates, returning `nil`.

4. If `active-window` is not zero, then it contains a window handle, and the following happens:
   The function uses `with-foreign-string` to convert the Lisp string argument of the function into a FLI string, and a pointer to the FLI string is allocated, ready to be handed to the foreign function `set-win-text` that we defined earlier. The encoding of the string is `external-format`, which is the encoding suitable for the operating system running on the computer. Once the window title has been set, `with-foreign-string` automatically deallocates the memory that was allocated for the FLI string and the pointer. The function then terminates, returning `t`.

You can test that this is what happens by entering the command:

```
(set-active-window-text "A new title for the active window")
```

See `with-foreign-string`, for more details on the use of foreign strings.

### 5.2 Passing and returning strings

#### 5.2.1 Use of Reference Arguments

Lisp and C cannot in general share memory so the FLI needs to make a copied of strings, either temporarily when passing them to C or as new Lisp objects when returning them.

#### 5.2.2 Passing a string

Use of the `:reference-pass` type in this example converts the Lisp string to a foreign string on calling, but does not convert the string back again on return.

Here is the C code for the example. It uses the argument string but returns an integer.

Windows version:

```c
#include <string.h>
#include <ctype.h>

__declspec(dllexport) int __cdecl count_upper(const char *string)
{
    int count;
    int len;
    int ii;
    count = 0;
    len = strlen(string);
    for (ii = 0; ii < len ; ii++)
        if (isupper(string[ii])
            count++;
    return count;
}
```
Non-Windows version:

```c
#include <string.h>
#include <ctype.h>

int count_upper(const char *string)
{
    int count;
    int len;
    int ii;
    count = 0;
    len = strlen(string);
    for (ii = 0; ii < len ; ii++)
        if (isupper(string[ii]))
            count++;  
    return count;
}
```

Here is the foreign function definition using `:reference-pass`:

```lisp
(fli:define-foreign-function (count-upper "count_upper" :source)
    ((string (:reference-pass :ef-mb-string)))
    :result-type :int
    :language :c
    :calling-convention :cdecl)
```

(count-upper "ABCdef")
=>
3

5.2.3 Returning a string via a buffer

In this example no Lisp string is needed when calling. The `:reference-return` type converts a foreign string of lowercase ASCII characters to a Lisp string on return. Here is the C code for the example.

Windows version:

```c
#include <string.h>
#include <stdlib.h>
__declspec(dllexport) void __cdecl random_string(int length, char *string)
{
    int ii;
    for (ii = 0; ii < length ; ii++)
        string[ii] = 97 + rand() % 26;
    string[length] = 0;
}
```

Non-Windows version:

```c
#include <string.h>
#include <stdlib.h>

void random_string(int length, char *string)
{
    int ii;
    for (ii = 0; ii < length ; ii++)
        string[ii] = 97 + rand() % 26;
    string[length] = 0;
}
In this foreign function definition the :reference-return type must specify a size, since memory is allocated for it before calling the C function. Note also the use of :lambda-list so that the caller does not have to pass a dummy argument for the returned string, and :result-type nil corresponding to the void declaration of the C function.

(fli:define-foreign-function (random-string
  "random_string"
  :source)
  ((length :int)
   (return-string (:reference-return
     (:ef-mb-string
      :limit 256))))
   :result-type nil
  :lambda-list (length &aux return-string)
  :calling-convention :cdecl)

(random-string 3)
=>
"uxw"

(random-string 6)
=>
"fnfozv"

5.2.4 Modifying a string in a C function

Here is the C code for the example. On return, the argument string has been modified (the code assumes there is enough space after the string for the extra characters).

Windows version:

```c
#include <stdio.h>
#include <string.h>
__declspec(dllexport) void __cdecl modify(char *string) {
  char temp[256];
  sprintf(temp, "'\%s' modified in a C function", string);
  strcpy(string, temp);
}
```

Non-Windows version:

```c
#include <stdio.h>
#include <string.h>

void modify(char *string) {
  char temp[256];
  sprintf(temp, "'\%s' modified in a C function", string);
  strcpy(string, temp);
}
```

Here are three approaches to calling modify from Lisp:

1. Use a fixed size buffer in define-foreign-function. This uses the :reference type, which automatically allocates a temporary foreign object, fills it with data converted from the Lisp object, passes a pointer to C and converts the data in the foreign object back into a new Lisp object on return. Note that the Lisp object passed to the function is not modified. This is the neatest way, provided you can bound the size of the result string at compile-time.

(fli:define-foreign-function (dff-modify "modify" :source)
  ((string (:reference (:ef-mb-string :limit 256))))
  :calling-convention :cdecl)
5 Advanced Uses of the FLI

(dff-modify "Lisp String")
=>
"'Lisp String' modified in a C function"

2. Use a fixed size buffer from with-dynamic-foreign-objects. In this case, we do most of the conversion explicitly and define the foreign function as taking a :pointer argument. This is a good approach if you don't know the maximum length when the function is defined, but will know it at compile-time for each call to the function.

(fli:define-foreign-function (wdfo-modify "modify" :source)
  ((string :pointer))
  :calling-convention :cdecl)

(fli:with-dynamic-foreign-objects
  ((c-string (:ef-mb-string :limit 256)
    :initial-element "Lisp String")
   (wdfo-modify c-string)
   (fli:convert-from-foreign-string c-string))
=>
"'Lisp String' modified in a C function"

3. With a variable size buffer from allocate-dynamic-foreign-object. In this case, we do all of the conversion explicitly because we need to make an array of the right size, which is only known after the foreign string has been created (the extra 100 bytes are to allow for what the C function inserts into the string). Note that, in order to support arbitrary external formats, the code makes no assumptions about the length of the temporary array being the same as the length of the Lisp string: it does the conversion first using with-foreign-string, which works out the required number of bytes. The use of with-dynamic-foreign-objects provides a dynamic scope for call to allocate-dynamic-foreign-object - on exit, the foreign object will be freed automatically.

(fli:with-foreign-string (temp element-count byte-count)
  "Lisp String"
  (fli:with-dynamic-foreign-objects ()
    (let ((c-string (fli:allocate-dynamic-foreign-object
      :type '(:unsigned :byte)
      :nelems (+ byte-count 100))))
      (fli:replace-foreign-object c-string temp :nelems byte-count)
      (wdfo-modify c-string)
      (fli:convert-from-foreign-string c-string)))))

5.2.5 Calling a C function that takes an array of strings

Suppose you have a C function declared like this:

extern "C" void foo( const char** StringArray);

To call this from Lisp you need to first allocate the foreign memory for each piece of data, that is the array itself and each string. Assuming that foo does not capture any of the pointers, you can give this memory dynamic extent as follows:

(defun convert-to-dynamic-foreign-array (strings)
  (let* ((count (length strings))
    (array
      (fli:allocate-foreign-object
       :nelems (+ count) ; assume NULL terminated
       :type '(:pointer :char))))
    (dotimes (index count)
      (setf (fli:dereference array :index index)
        (fli:convert-to-dynamic-foreign-string
         (elt strings index))))
    (setf (fli:dereference array :index count) nil))
Here is a similar example converting Lisp strings to **char or *char[] which by default allocates using malloc (the value :static for the allocation argument):

```lisp
(defun convert-strings-to-foreign-array (strings &key
  (allocation :static))
  (let* ((count (length strings))
     (array (fli:allocate-foreign-object
       :type '(:pointer (:unsigned :char))
       :nelems (1+ count)
       :initial-element nil
       :allocation allocation)))
    (loop for index from 0
         for string in strings
         do (setf (fli:dereference array :index index)
                   (fli:convert-to-foreign-string
                    string
                    :external-format :utf-8
                    :allocation allocation)))
  array))
```

If you call it frequently, then you will probably want to free the array (and the strings inside it). Alternatively, you can give the array and its strings dynamic scope if the foreign side does not keep a pointer to the data, like this:

```lisp
(fli:with-dynamic-foreign-objects ()
  (let ((array (convert-strings-to-foreign-array
                strings :allocation :dynamic)))
    (%foo array)))
```

### 5.2.6 Foreign string encodings

The :ef-mb-string type is capable of converting between the internal encoding of LispWorks strings (Unicode) and various encodings that may be expected by the foreign code. The encoding on the foreign side is specified by the :external-format argument, which takes an External Format specification. See the LispWorks® User Guide and Reference Manual for a more detailed description of external formats.

Consider a variant of the last example where the returned string contains characters beyond the ASCII range.

Windows version:

```c
#include <string.h>
#include <stdlib.h>
__declspec(dllexport) void __cdecl random_string2(int length, char *string)
{
  int ii;
  for (ii = 0; ii < length ; ii++)
    string[ii] = 225 + rand() % 26;
  string[length] = 0;
}
```

Non-Windows version:
```c
#include <string.h>
#include <stdlib.h>

void random_string2(int length, char *string) {
    int ii;
    for (ii = 0; ii < length; ii++)
        string[ii] = 225 + rand() % 26;
    string[length] = 0;
}

A foreign function defined like `random-string` above is inadequate by itself here because the default external format is that for the default C locale, ASCII. This will signal error when it encounters a non-ASCII character code. There are two approaches to handling non-ASCII characters.

1. Pass an appropriate external format, in this case it is Latin-1:

   ```scheme
   (fli:define-foreign-function (random-string2
        "random_string2"
        :source)
        ((length :int)
         (return-string (:reference-return
                         (:ef-mb-string
                          :limit 256
                          :external-format :latin-1))))
        :result-type nil
        :lambda-list (length &aux return-string)
        :calling-convention :cdecl)

   (random-string2 3)
   =>
   "ðâã"

   (random-string2 6)
   =>
   "ðâæççä"
   ```

2. Set the locale, using `set-locale`. This sets the C locale and switches the FLI to use an appropriate default wherever an external-format argument is accepted.

   ```scheme
   (fli:define-foreign-function (random-string
        "random_string2"
        :source)
        ((length :int)
         (return-string (:reference-return
                         (:ef-mb-string
                          :limit 256))))
        :result-type nil
        :lambda-list (length &aux return-string)
        :calling-convention :cdecl)

   On a Windows system with current Code Page for Western European languages:

   (fli:set-locale)
   =>
   (win32:code-page :id 1252)

   On a Non-Windows system with a Latin-1/ISO8859-1 default locale:

   (fli:set-locale)
   =>
   :latin-1
   ```
After the default external-format has been switched:

(random-string 6)
=>
"ðèñçèõ"

If you do not actually wish to set the C locale, you can call set-locale-encodings which merely switches the FLI to use the specified external formats where an external-format argument is accepted.

5.2.7 Foreign string line terminators

You can specify the line terminator in foreign string conversions via the :eol-style parameter in the external-format argument.

By default foreign strings are assumed to have lines terminated according to platform conventions: Linefeed on Non-Windows systems, and Carriage-Return followed by Linefeed on Windows. That is, eol-style defaults to :lf and :crlf respectively. This means that unless you take care to specify the external format :eol-style parameter, you may get unexpected string length when returning a Lisp string.

Consider the following C code example on Windows:

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
__declspec(dllexport) int __cdecl crlf_string(int length, char *string)
{
  int ii;
  int jj;
  for (ii = 0; ii < length ; ii++)
    if (ii % 3 == 1) {
      string[ii] = 10;
      printf("%d\n", ii);
    } else
    if (((ii > 0) && (ii % 3 == 0)) {
      string[ii] = 13;
      printf("%d\n", ii);
    } else
    if (ii % 3 == 2) {
      string[ii] = 97 + rand() % 26;
      printf("%d\n", ii);
    }
  string[length] = 0;
  return length;
}
```

Call this C function from Lisp:

```
(fli:define-foreign-function (crlf-string  
"crlf_string"  
:source)
((length :int)  
(return-string (:reference-return  
(:ef-mb-string  
:limit 256  
:external-format :latin-1))))  
:lambda-list (length &aux return-string)  
:calling-convention :cdecl  
:result-type :int)

(multiple-value-bind (length string)
```
Each two character CR LF sequence in the foreign string has been mapped to a single LF character in the Lisp string. If you want to return a Lisp string and not do line terminator conversion, then you must specify the eol-style as in this example:

```lisp
(fli:define-foreign-function (crlf-string "crlf_string" :source)
  ((length :int) (return-string (:reference-return (:ef-mb-string :limit 256 :external-format (:latin-1 :eol-style :lf))))
   :lambda-list (length &aux return-string) :calling-convention :cdecl :result-type :int)

(multiple-value-bind (length string) (crlf-string 99) (format t "~C length ~D, Lisp string length ~D~%" length (length string)))
=>
C length 99, Lisp string length 67
```

5.2.8 Win32 API functions that handle strings

Functions in the Win32 API that handle strings come in two flavors, one for ANSI strings and one for Unicode strings. Supported versions of Microsoft Windows support both flavors. The functions are named with a single letter suffix, an A for the ANSI functions and a W for the Unicode functions. So for example both `CreateFileA` and `CreateFileW` exist. In C, this is finessed by the use of `#define` in the header files.

There are three ways to handle this:

- Use the A function explicitly, for example:

  ```lisp
  (define-foreign-function (create-file "CreateFileA")
    ((lpFileName win32:lpcstr) ...))
  
  This will prevent the use of Unicode strings but this is typically only a problem if you are handling mixed language data. Be sure to use the correct FLI types `win32:str, win32:lpcstr` and so on when explicitly interfacing to an ANSI Win32 function.

- Use the W function explicitly, for example:

  ```lisp
  (define-foreign-function (create-file "CreateFileW")
    ((lpFileName win32:lpcwstr) ...))
  
  This will allow use of Unicode strings. Be sure to use the correct FLI types `win32:wstr, win32:lpcwstr` and so on when explicitly interfacing to a Unicode Win32 function.

- Use encoding :dbcs in `define-foreign-function` and omit the single letter suffix, for example:

  ```lisp
  (fli:define-foreign-function (create-file "CreateFile" :dbcs)
    ((lpFileName win32:lpctstr) ...))
  ```
This will cause it to use the Unicode \texttt{W} function implicitly in supported versions of Windows. (In some older operating systems such as Windows ME, this mechanism would implicitly use the ANSI \texttt{A} function.)

In all cases, as well as calling the correct function, you must encode/decode any string arguments and results correctly, to match the \texttt{A} or \texttt{W} in the function name. The foreign types \texttt{win32:tstr}, \texttt{win32:lpctstr} and \texttt{win32:lpctstr} automatically switch between ANSI and Unicode strings and correspond to the typical ones found in the Win32 API. For more information about these foreign types, see their manual pages in the \textit{LispWorks User Guide and Reference Manual}.

### 5.2.9 Mapping \texttt{nil} to a Null Pointer

If you wish a string argument to accept \texttt{nil} and pass it as a null pointer, or to return a null pointer as Lisp value \texttt{nil}, use the \texttt{:allow-null} argument to the \texttt{:reference} types.

The C function \texttt{strcap} in the following example modifies a string, but also accepts and returns a null pointer if passed.

**Windows version:**

```c
#include <string.h>
#include <ctype.h>

__declspec(dllexport) void __cdecl strcap(char *string)
{
    int len;
    int ii;
    if (string) {
        len = strlen(string);
        if (len > 0) {
            for (ii = len - 1; ii > 0; ii--)
                if (isupper(string[ii]))
                    string[ii] = tolower(string[ii]);
            if (islower(string[0]))
                string[0] = toupper(string[0]);
        }
    }
}
```

**Non-Windows version:**

```c
#include <string.h>
#include <ctype.h>

void strcap(char *string)
{
    int len;
    int ii;
    if (string) {
        len = strlen(string);
        if (len > 0) {
            for (ii = len - 1; ii > 0; ii--)
                if (isupper(string[ii]))
                    string[ii] = tolower(string[ii]);
            if (islower(string[0]))
                string[0] = toupper(string[0]);
        }
    }
}
```

With this following foreign function definition:

```lisp
(fli:define-foreign-function (strcap "strcap" :source)
  ((string (:reference :ef-mb-string)))
)
```

46
5 Advanced Uses of the FLI

```lisp
(defun c-string-capitalize (string)
  (fli:with-foreign-string (ptr elts bytes :allow-null t)
    string
    (declare (ignore elts bytes))
    (strcap ptr)
    (fli:convert-from-foreign-string ptr :allow-null t)))
```

```lisp
(c-string-capitalize "abC")
=>
"Abc"
```

However, `(strcap nil)` signals error because the `:ef-mb-string` type expects a string.

Using `:allow-null` allows `nil` to be passed:

```lisp
(defun c-string-capitalize (string)
  (fli:with-foreign-string (ptr elts bytes :allow-null t)
    string
    (declare (ignore elts bytes))
    (strcap ptr)
    (fli:convert-from-foreign-string ptr :allow-null t)))
```

```lisp
(c-string-capitalize nil)
=>
nil
```

Note that `with-foreign-string`, `convert-to-foreign-string` and `convert-from-foreign-string` also accept an `:allow-null` argument. So another way to call `strcap` and allow the null pointer is:

```lisp
(defun c-string-capitalize (string)
  (fli:with-foreign-string (ptr elts bytes :allow-null t)
    string
    (declare (ignore elts bytes))
    (strcap ptr)
    (fli:convert-from-foreign-string ptr :allow-null t)))
```

```lisp
(c-string-capitalize "abC")
=>
"Abc"
```

```lisp
(c-string-capitalize nil)
=>
nil
```

5.3 Lisp integers

Lisp integers cannot be used directly in the FLI unless they are known to be of certain sizes that match foreign types such as `:int`.

However, the FLI provides a mechanism to convert any Lisp integer into a foreign array of bytes and to convert that array back to an equivalent Lisp integer. This would allow the integer to be stored in an database for example and then retrieved later.

The macro `with-integer-bytes` and the function `convert-integer-to-dynamic-foreign-object` generates the array of bytes and also to determine its length. The function `make-integer-from-bytes` converts the foreign array back
5 Advanced Uses of the FLI

to an integer. The layout of the bytes is unspecified, so these operations must be used for all such conversions.

5.4 Defining new types

The FLI provides the `define-foreign-type` macro for defining new FLI types, using the basic FLI types that you have seen in 2 FLI Types. The next example shows you how to define a new array type that only takes an odd number of dimensions.

```
(fli:define-foreign-type odd-array (element &rest dimensions)
  (unless (oddp (length dimensions))
    (error "Can't define an odd array with even dimensions - try
      adding an extra dimension!"))
  `(:c-array ,element ,@dimensions))
```

The new array type is called `odd-array`, and takes a FLI type and a sequence of numbers as its arguments. When trying to allocate an `odd-array`, if there are an even number of items in the sequence then an error is raised. If there are an odd number of items then an instance of the array is allocated. The next command raises an error, because a 2 by 3 array has an even dimension.

```
(fli:allocate-foreign-object :type '(odd-array :int 2 3))
```

However, adding an extra dimension and defining a 2 by 3 by 4 array works:

```
(fli:allocate-foreign-object :type `(odd-array :int 2 3 4))
```

For more information on defining types see `define-foreign-type`.

5.5 Using DLLs within the LispWorks FLI

In order to use functions defined in a dynamically linked library (DLL) within the LispWorks FLI, the functions need to be exported from the DLL.

5.5.1 Using C DLLs

You can export C functions in three ways:

1. Use a `__declspec(dllexport)` declaration in the C file.

   In this case you should also make the functions use the `cdecl` calling convention, which removes another level of name mangling.

2. Use an `/export` directive in the link command.

3. Use a `.def` file.

An example of method 2 follows. Let us assume you have the following C code in a file called `example.c`.

```c
int multiply (int i1, int i2)
{
  int result;
  result = i1 * i2 * 500;
  return result;
}
```

Then you can create a DLL by, for example, using a 32 bit C compiler such as `cl.exe`. 

Finally, you should use the LispWorks FLI to define your C function in your Lisp code. This definition should look something like:

```lisp
(fli:define-foreign-function (multiply "multiply")
  ((x :int)
   (y :int))
  :result-type :int
  :module :my-dll
  :calling-convention :cdecl)
```

Note that the `define-foreign-function` also includes a `:calling-convention` keyword to specify that the function we are interfacing to is defined as using the `__cdecl` calling convention (the default for cl.exe).

5.5.1.1 Testing whether a function is defined

Having loaded your DLLs (with `register-module`) you may wish to test whether certain functions are now available.

To detect when a C function `name` is defined, call:

```lisp
(not (fli:null-pointer-p
     (fli:make-pointer :symbol-name name
                      :errorp nil)))
```

You can also return a list of unresolved foreign symbol names by calling `module-unresolved-symbols`.

5.5.2 Using C++ DLLs

You must make the exported names match the FLI definitions. To do this:

- If you can alter the C++ code, wrap `extern "C"` {} around the C++ function definitions, or:
- Create a second DLL with C functions that wrap around each C++ function, and make those C functions accessible as described in 5.5.1 Using C DLLs.

Note: watch out for the calling convention of the exported function, which must match the `:calling-convention` in the FLI definitions.

5.6 Incorporating a foreign module into a LispWorks image

Embedded dynamic modules are dynamically loaded foreign modules which are embedded (that is, the data is stored inside the LispWorks image). They can then be used at run time.

The formats supported include DLL on Windows, dylib on macOS, and shared object or shared library on other platforms. See 1.1.2 Loading foreign code for details of the types of modules supported.

You use an embedded dynamic module when you want to integrate foreign code, and that foreign code is not expected to be available on the end-user’s computer. In principle this could also be achieved by supplying the foreign module as a separate file together with the Lisp image, locating it at run time and loading it with `register-module`. The embedded dynamic modules mechanism simplifies this.

The main interface is `get-embedded-module`, which is called at load time to "intern" the module, and `install-embedded-module` which needs to be called at run time to make the foreign code available. It is possible to incorporate in a fasl file by using `get-embedded-module-data` and `setup-embedded-module` instead of `get-embedded-module`. 

49
Another way to "intern" the module is to define a \texttt{lw:deFS} system containing a C source file member with the \texttt{:embedded-module} keyword. When the system is loaded, the value associated with \texttt{:embedded-module} is used to create the embedded module. You would then call \texttt{install-embedded-module} at run time to make the foreign code available.

### 5.7 Block objects in C (foreign blocks)

This section applies to LispWorks for Macintosh, only.

Foreign blocks are objects that correspond to the opaque "Block" object in C and derived languages that are introduced in CLANG and used by Apple Computer, Inc.

A "Block" in C is similar to a closure in Lisp. It encloses a piece of code, and potentially some variables (which may be local), and allows invocation of this code.

LispWorks foreign blocks allows your Lisp program to call into and get called by code that uses blocks.

A foreign block is represented in LispWorks by a foreign pointer with pointer type \texttt{foreign-block-pointer}. Even though these are foreign pointers, these objects should be regarded as opaque, and should not be manipulated or used except as described below.

You use a foreign block by passing it to a foreign function that is defined to take a block as an argument, or by invoking a block that is received from a foreign function. The argument type needs to be specified as \texttt{foreign-block-pointer}.

When a foreign function invokes a block which was created in Lisp (or a copy of it), this invocation calls a Lisp function which the programmer supplied to the creating function or macro. When Lisp invokes a block that came from foreign code, it invokes some (unknown) foreign code.

Blocks can be used to run code via the Grand Central Dispatch mechanism (GCD) in macOS (see Apple documentation). There is a simple example in:

\begin{verbatim}
  (example-edit-file "fli/grand-central-dispatch")
\end{verbatim}

#### 5.7.1 Calling foreign code that receives a block as argument

To call foreign code that needs a block as an argument, the Lisp program needs to create the blocks. You do this in two steps:

1. At load time, define a "type" by using the macro \texttt{define-foreign-block-callable-type}. This "type" corresponds to the "signature" in C.

2. At run time, generate the block, for example by calling \texttt{allocate-foreign-block} with the "type". Alternatively use one of the macros \texttt{with-foreign-block} and \texttt{with-local-foreign-block}. When generating the block, you also pass an arbitrary Lisp function that gets called when the block (or a copy of it) is invoked.

Foreign blocks created by \texttt{allocate-foreign-block} are released when appropriate by \texttt{free-foreign-block}.

Foreign block pointers created by \texttt{allocate-foreign-block} are of type \texttt{foreign-block-pointer} and print with "\texttt{lisp-foreign-block-pointer}".

For examples see:

\begin{verbatim}
  (example-edit-file "fli/foreign-blocks")
\end{verbatim}

and:

\begin{verbatim}
  (example-edit-file "fli/grand-central-dispatch")
\end{verbatim}
5 Advanced Uses of the FLI

5.7.2 Operations on foreign blocks

You might obtain a foreign pointer of type **foreign-block-pointer** that was passed as an argument to another foreign block, to a callable defined by **define-foreign-callable** or returned by a foreign function.

The foreign block can be invoked by defining an invoker (at load time) using **define-foreign-block-invoker**, and calling the invoker. If you need to keep the block after returning to the caller, you normally need to copy it by **foreign-block-copy**. If you copy a block, once you are finished with it, you should release it by **foreign-block-release**.

For examples of this see:

```lisp
(example-edit-file "fli/invoke-foreign-block")
```

5.7.3 Scope of invocation

In principle, in the general case each of these is not defined:

- The time at which the code that the block encapsulates is invoked. In particular, even after a block is released (freed), the same code may be invoked by a copy of the block.
- In which thread the code is invoked.
- How many invocations can occur in parallel. In other words, whether it is invoked serially or concurrently.

The implementation of foreign blocks copes with all of these, that is it can work concurrently on any thread and after the block was released/freed, as long as there are live copies of it (except with blocks created by **with-local-foreign-block**). However, whether the code inside the block can cope with it is dependent on the code. This needs to be considered when creating blocks.

Specific foreign functions that take blocks as argument should be documented to state the scope of invocation. Apple's documentation commonly states whether the code is invoked concurrently or serially. In some functions the caller can decide when it calls the function whether the code can be executed concurrently or not. If you pass the block to a function that is documented to execute it serially, or you can tell it to do it, then you can assume that function that you made the block with is not going to be called concurrently from the block. Otherwise it must be able to cope with concurrent calls from the blocks.

Whether the code may be invoked on another thread or after the function that took the block returned is not normally documented. In many cases it can be deduced with confidence: when you dispatch a block to a queue (for example **dispatch_after** and similar functions, see the Apple documentation) it clearly can be invoked from another threads after the function returns. In the case of **qsart_b** (see Apple documentation and the example in **(example-edit-file "fli/foreign-blocks")**) we can be sure that the code will not be invoked after **qsart_b** returned, because the arguments to the block are based on the data (first argument to **qsart_b**), and **qsart_b** and its callees cannot be guaranteed that the data is still valid once **qsart_b** returned. On the other hand, we cannot be sure that the block is not invoked on another thread(s) before **qsart_b** returns. Currently it is probably always called in the same thread where **qsart_b** was called, but the interface does not guarantee it.

Thus when you create a foreign block in Lisp, the following considerations apply to the Lisp function **function** that you supply:

- In most cases, **function** needs to cope with being called in any thread, and hence cannot rely on the dynamic environment. Normally it is impossible to deduce that function will not be called on another thread, so it can be guaranteed only when the function to which the block is passed is documented to guarantee it.

  **Note:** that is the only situation in which it is really valid to use **with-local-foreign-block**.
• *function* may need to be able to cope with being called at any time, unless it is documented or deducible from the
  interface that it can be called only within the scope of the caller. It may be possible to deduce the time limit on a call
  from the way the block is used.

• The function needs to be able to cope with being called concurrently, unless the documentation of the user of the blocks
  says that it does not, or you can tell that it is going to be called only on one thread.

### 5.8 Interfacing to graphics functions

This section applies to LispWorks for Windows, only.

If you use graphics functionality via the FLI on Microsoft Windows be aware that you may need to call the function

This condition does not apply on non-Windows platforms.

### 5.9 Summary

In this chapter a number of more advanced examples have been presented to illustrate various features of the FLI. The use of
the FLI to pass strings dynamically to Win32 API functions was examined, as was the definition of new FLI types and the use
of callable functions and foreign functions, including code using blocks.

The next two chapters form the reference section of this manual. They provide reference entries for the functions, macros,
and types which make up the FLI.
6 Self-contained examples

This chapter enumerates the set of examples in the LispWorks library relevant to the content of this manual. Each example file contains complete, self-contained code and detailed comments, which include one or more entry points near the start of the file which you can run to start the program.

To run the example code:

1. Open the file in the Editor tool in the LispWorks IDE. Evaluating the call to `example-edit-file` shown below will achieve this.
2. Compile the example code, by `Ctrl+Shift+B`.
3. Place the cursor at the end of the entry point form and press `Ctrl+X Ctrl+E` to run it.
4. Read the comment at the top of the file, which may contain further instructions on how to interact with the example.

### 6.1 Foreign block examples

This section lists the examples illustrating the use of foreign blocks, which is described in [5.7 Block objects in C (foreign blocks)](57).

These examples apply to LispWorks for Macintosh only:

```
(ex example-edit-file "fli/foreign-blocks")
```

```
(ex example-edit-file "fli/grand-central-dispatch")
```

```
(ex example-edit-file "fli/invoke-foreign-block")
```

### 6.2 Miscellaneous examples

```
(ex example-edit-file "fli/foreign-callable-example")
```
align-of

Summary
Returns the alignment in bytes of a foreign type.

Package
fli

Signature
align-of type-name => alignment

Arguments
type-name↓ A foreign type whose alignment is to be determined.

Values
alignment The alignment of the foreign type type-name in bytes.

Description
The function \texttt{align-of} returns the alignment in bytes of the foreign language type named by \texttt{type-name}.

Examples
The following example shows types with various alignments.

\begin{verbatim}
(fli:align-of :char) => 1
(fli:align-of :int) => 4
(fli:align-of :double) => 8
(fli:align-of :pointer) => 4
\end{verbatim}
allocate-dynamic-foreign-object

Summary

Allocates memory for an instance of a foreign object within the scope of a \texttt{with-dynamic-foreign-objects} macro.

Package

\texttt{fli}

Signatures

\texttt{allocate-dynamic-foreign-object} \texttt{&key type pointer-type initial-element initial-contents fill nelems size-slot} \texttt{=>} \texttt{pointer}

\texttt{alloca} \texttt{&key type pointer-type initial-element initial-contents fill nelems size-slot} \texttt{=>} \texttt{pointer}

Arguments

\texttt{type} \textdownarrow \;
\begin{itemize}
  \item A FLI type specifying the type of the object to be allocated. If \texttt{type} is supplied, \texttt{pointer-type} must not be supplied.
\end{itemize}

\texttt{pointer-type} \textdownarrow \;
\begin{itemize}
  \item A FLI pointer type specifying the type of the pointer object to be allocated. If \texttt{pointer-type} is supplied, \texttt{type} must not be supplied.
\end{itemize}

\texttt{initial-element} \textdownarrow \;
\begin{itemize}
  \item The initial value of the newly allocated objects.
\end{itemize}

\texttt{initial-contents} \textdownarrow \;
\begin{itemize}
  \item A list of values to initialize the contents of the newly allocated objects.
\end{itemize}

\texttt{fill} \textdownarrow \;
\begin{itemize}
  \item An integer between 0 to 255.
\end{itemize}

\texttt{nelems} \textdownarrow \;
\begin{itemize}
  \item An integer specifying how many copies of the object should be allocated. The default value is 1.
\end{itemize}

\texttt{size-slot} \textdownarrow \;
\begin{itemize}
  \item A symbol naming a slot in the object.
\end{itemize}

Values

\texttt{pointer} \;
\begin{itemize}
  \item A pointer to the specified \texttt{type} or \texttt{pointer-type}.
\end{itemize}

Description

The function \texttt{allocate-dynamic-foreign-object} allocates memory for a new instance of an object of type \texttt{type} or an instance of a pointer object of type \texttt{pointer-type} within the scope of the body of the macro \texttt{with-dynamic-foreign-objects}, \texttt{initial-element}, \texttt{initial-contents}, \texttt{fill}, \texttt{nelems} and \texttt{size-slot} initialize the allocated instance as if by \texttt{allocate-foreign-object}.

Once this macro has executed, the memory allocated using \texttt{allocate-dynamic-foreign-object} is therefore automatically freed for other uses.
The function `alloca` is a synonym for `allocate-dynamic-foreign-object`.

Examples

A full example using `with-dynamic-foreign-objects` and `allocate-dynamic-foreign-object` is given in [1.4 An example of dynamic memory allocation](#).

See also

- `allocate-foreign-object`
- `with-dynamic-foreign-objects`
- [1.4 An example of dynamic memory allocation](#)
- [3.5 More examples of allocation and pointer allocation](#)
- [5.2.4 Modifying a string in a C function](#)

### allocate-foreign-block

**Function**

**Summary**

Allocates a foreign block, in LispWorks for Macintosh.

**Package**

`fli`

**Signature**

\[
\text{allocate-foreign-block} \quad \text{type} \quad \text{function} \quad \&\text{rest} \quad \text{extra-arguments} \quad \Rightarrow \quad \text{foreign-block}
\]

**Arguments**

- `type`\[\]
  
  A symbol.

- `function`\[\]
  
  A Lisp function.

- `extra-arguments`\[\]
  
  Arguments.

**Values**

- `foreign-block`\[\]
  
  A Lisp-allocated `foreign-block-pointer`.

**Description**

The function `allocate-foreign-block` allocates a foreign block of type `type` such that when the foreign block is invoked it calls the function `function` with the arguments given to the block followed by `extra-arguments` (if any).

`type` is a symbol which must have been defined as a type using `define-foreign-block-callable-type`.

`function` is any Lisp function, but see the [5.7.3 Scope of invocation](#) for potential limitations.

The resulting foreign block lives indefinitely, until it is freed by `free-foreign-block`, and can be used repeatedly and concurrently. It cannot be garbage collected, so if your program repeatedly allocates foreign blocks, you need to free them by calls to `free-foreign-block`. The macro `with-foreign-block` does this for you.

`extra-arguments` allows you to (roughly speaking) "close over" some values to the function, but they are read-only. If the...
function needs to set values, you can either pass some objects and set slots inside them, or make the function a real Lisp closure.

Notes
The result of `allocate-foreign-block` prints with "lisp-foreign-block-pointer".

`allocate-foreign-block` is implemented in LispWorks for Macintosh only.

See also

`define-foreign-block-callable-type`
`free-foreign-block`
`with-foreign-block`

5.7 Block objects in C (foreign blocks)

---

### allocate-foreign-object

#### malloc

**Summary**

Allocates memory for an instance of a foreign object.

**Package**

fli

**Signatures**

#### allocate-foreign-object &key
type pointer-type initial-element initial-contents fill nelems size-slot allocation => pointer

#### malloc &key
type pointer-type initial-element initial-contents fill nelems size-slot allocation => pointer

**Arguments**

- **type**: A FLI type specifying the type of the object to be allocated. If `type` is supplied, `pointer-type` must not be supplied.
- **pointer-type**: A FLI pointer type specifying the type of the pointer object to be allocated. If `pointer-type` is supplied, `type` must not be supplied.
- **initial-element**: The initial value of the newly allocated objects.
- **initial-contents**: A list of values to initialize the contents of the newly allocated objects.
- **fill**: An integer between 0 to 255.
- **nelems**: An integer specifying how many copies of the object should be allocated. The default value is 1.
- **size-slot**: A symbol naming a slot in the object.
- **allocation**: A keyword, either `dynamic` or `static`. 
Values

\[ \text{pointer} \]

A pointer to the specified type or pointer-type.

Description

The function `allocate-foreign-object` allocates memory for a new instance of an object of type `type` or an instance of a pointer object of type `pointer-type`.

If `allocation` is `:static` then memory is allocated in the C heap and must be explicitly freed using `free-foreign-object` once the object is no longer needed.

If `allocation` is `:dynamic`, then `allocate-foreign-object` allocates memory for the object and pointer within the scope of the body of `with-dynamic-foreign-objects`. This is equivalent to using `allocate-dynamic-foreign-object`.

The default value of `allocation` is `:static`.

An integer value of `fill` initializes all the bytes of the object. If `fill` is not supplied, the object is not initialized unless `initial-element` or `initial-contents` is passed.

If `initial-contents` is supplied and its length is less than `nelems`, then the remaining elements are not initialized.

If `initial-contents` is supplied and its length is greater than `nelems`, then the length of `initial-contents` overrides `nelems`. This is a common case where `initial-contents` is supplied and `nelems` is omitted (and hence defaults to 1).

`size-slot` can be used to initialize a slot in a struct or union type to the size of the object in bytes. If `size-slot` is supplied then it must be the name of a slot in that type. The slot named by `size-slot` is set to the size of the object in bytes. This occurs after `fill`, `initial-element` and `initial-contents` are processed. If `nelems` is greater than 1, then the slot named by `size-slot` is initialized in each element. If `size-slot` is not supplied, then no such setting occurs.

The function `malloc` is a synonym for `allocate-foreign-object`.

Notes

When `allocation` is `:static`, memory allocated by `allocate-foreign-object` is in the C heap. Therefore `pointer` (and any copy) cannot be used after `save-image` or `deliver`.

Examples

In the following example a structure is defined and an instance with a specified initial value of 10 is created with memory allocated using `allocate-foreign-object`. The `dereference` function is then used to get the value that `point` points to, and finally it is freed.

```lisp
(fli:define-c-typedef LONG :long)

(setq point (fli:allocate-foreign-object
              :type 'LONG
              :initial-element 10))

(fli:dereference point)

(fli:free-foreign-object point)
```

See also

`allocate-dynamic-foreign-object`
**free-foreign-object**

3 FLI Pointers

---

**cast-integer**  

*Function*

**Summary**

Casts an integer to a given type.

**Package**

fli

**Signature**

\texttt{cast-integer \hspace{1em} integer \hspace{1em} type \Rightarrow result}

**Arguments**

- \texttt{integer} \hspace{1em} A Lisp integer.
- \texttt{type} \hspace{1em} A foreign type.

**Values**

- \texttt{result} \hspace{1em} A Lisp integer.

**Description**

The function \texttt{cast-integer} casts the integer \texttt{integer} to the foreign type \texttt{type}.

\texttt{type} must be a FLI integer type, either primitive or derived.

**Examples**

\begin{verbatim}
  (format nil "~B"
         (fli:cast-integer -1 '(:unsigned :int)))
  =>
  "11111111111111111111111111111111"
\end{verbatim}

See also

\texttt{:signed}

\texttt{:unsigned}

---

**connected-module-pathname**  

*Function*

**Summary**

Returns the real pathname of a connected module.
Package
fli

Signature
connected-module-pathname name => pathname

Arguments
name
A string or symbol.

Values
pathname
A pathname or nil.

Description
The function connected-module-pathname returns the real pathname of the connected module registered with name name.

If no module name is registered, or if the module name is not connected, then connected-module-pathname returns nil.

Examples

(fli:connected-module-pathname "gdi32")
=> #P"C:/WINNT/system32/GDI32.dll"

(fli:register-module :user-dll
 :real-name "user32"
 :connection-style :immediate)
=> :user-dll

(fli:connected-module-pathname :user-dll)
=> #P"C:/WINNT/system32/USER32.dll"

(fli:disconnect-module :user-dll)
=> t

(fli:connected-module-pathname :user-dll)
=> nil

See also
disconnect-module
register-module
convert-from-foreign-string

Summary
Converts a foreign string to a Lisp string.

Package
fli

Signature
convert-from-foreign-string pointer &key external-format length null-terminated-p allow-null => string

Arguments
pointer⇓ A pointer to a foreign string.
external-format⇓ An external format specification.
length⇓ The length of the string to convert.
null-terminated-p⇓ If t, it is assumed the string terminates with a null character. The default value for null-terminated-p is t.
allow-null⇓ A boolean. The default is false.

Values
string A Lisp string, or nil.

Description
The function convert-from-foreign-string, given a pointer to a foreign string, converts the foreign string to a Lisp string. The pointer does not need to be of the correct type, as it will automatically be coerced to the correct type as specified by external-format.

external-format is interpreted as by with-foreign-string. The names of available external formats are listed in section 26.6 External Formats to translate Lisp characters from/to external encodings in the LispWorks® User Guide and Reference Manual.

Either length or null-terminated-p must be non-nil. If null-terminated-p is true and length is not specified, it is assumed that the foreign string to be converted is terminated with a null character.

If allow-null is true and pointer is a null pointer then nil is returned. Otherwise, an error is signalled if pointer is a null pointer.

See also
convert-to-foreign-string
set-locale
set-locale-encodings
with-foreign-string
convert-integer-to-dynamic-foreign-object

Summary
Converts a Lisp integer to foreign bytes.

Package
fli

Signature
convert-integer-to-dynamic-foreign-object integer => pointer, length

Arguments
integer
An integer.

Values
pointer
A foreign pointer.
length
An integer.

Description
The function convert-integer-to-dynamic-foreign-object makes a dynamic foreign object containing the bytes of integer and returns pointer pointing to the first byte of that object and length which is the number of bytes in that object. The layout of the bytes is unspecified, but the bytes and the length are sufficient to reconstruct integer by calling make-integer-from-bytes.

See also
5.3 Lisp integers
with-integer-bytes
make-integer-from-bytes

convert-to-dynamic-foreign-string

Summary
Converts a Lisp string to a foreign string within the scope of the body of a with-dynamic-foreign-objects macro.

Package
fli
7 Function, Macro and Variable Reference

**Signature**

```lisp
convert-to-dynamic-foreign-string string &key external-format null-terminated-p allow-null => pointer, length, byte-count
```

**Arguments**

- `string` ↓ A Lisp string.
- `external-format` ↓ An external format specification.
- `null-terminated-p` ↓ If `t`, the foreign string terminates with a null character. The default value is `t`.
- `allow-null` ↓ A boolean. The default is `nil`.

**Values**

- `pointer` ↓ A FLI pointer to the foreign string.
- `length` The length of the string (including the terminating null character if there is one).
- `byte-count` The number of bytes in the converted string.

**Description**

The function `convert-to-dynamic-foreign-string` converts a Lisp string to a foreign string, and returns a pointer to the string and the length of the string. The memory allocation for the string and pointer is within the scope of the body of a `with-dynamic-foreign-objects` command.

`external-format` is interpreted as by `with-foreign-string`. The names of available external formats are listed in section 26.6 External Formats to translate Lisp characters from/to external encodings in the LispWorks® User Guide and Reference Manual.

`null-terminated-p` specifies whether the foreign string is terminated with a null character. It defaults to `t`.

If `allow-null` is non-nil and `string` is `nil` then a null pointer `pointer` is returned. Otherwise, an error is signalled if `string` is `nil`.

**See also**

- `allocate-dynamic-foreign-object`
- `convert-from-foreign-string`
- `convert-to-foreign-string`  
- `set-locale`
- `set-locale-encodings`
- `with-dynamic-foreign-objects`
- `with-foreign-string`


5.2.5 Calling a C function that takes an array of strings
**convert-to-foreign-string**

**Summary**

Converts a Lisp string to a foreign string.

**Package**

fli

**Signature**

`convert-to-foreign-string string &key external-format null-terminated-p allow-null into limit allocation => pointer, length, byte-count`

**Arguments**

- `string` A Lisp string.
- `external-format` An external format specification.
- `null-terminated-p` If `t`, the foreign string terminates with a null character. The default value is `t`.
- `allow-null` A boolean. The default is `nil`.
- `into` A foreign array, a foreign pointer or `nil`. The default is `nil`.
- `limit` A non-negative fixnum, or `nil`. The default is `nil`.
- `allocation` A keyword, either :dynamic or :static. The default is :static.

**Values**

- `pointer` A FLI pointer to the foreign string.
- `length` The length of the foreign string (including the terminating null character if there is one).
- `byte-count` The number of bytes in the foreign string.

**Description**

The function `convert-to-foreign-string` converts a Lisp string to a foreign string, and returns a pointer to the string. `external-format` is interpreted as by `with-foreign-string`. The names of available external formats are listed in section 26.6 External Formats to translate Lisp characters from/to external encodings in the LispWorks® User Guide and Reference Manual.

`null-terminated-p` specifies whether the foreign string is terminated with a null character. It defaults to `t`.

If `allow-null` is non-nil and `string` is `nil` then a null pointer `pointer` is returned. Otherwise, an error is signalled if `string` is `nil`.

If `into` is `nil`, then a new foreign string is allocated according to `allocation`, and `limit` is ignored.

If `into` is a FLI pointer to a integer type, then `limit` must be a fixnum and up to `limit` elements are filled with elements converted from the characters of `string`. The size of the integer type must equal the foreign size of `external-format`. 

---

64
If \( \text{into} \) is a FLI array of integers or a pointer to a FLI array of integers, up to \( \text{limit} \) elements are filled with elements converted from the characters of \( \text{string} \). If \( \text{limit} \) is \text{nil}, then the dimensions of the array are used. The size of the array element type must equal the foreign size of \text{external-format}.

If \( \text{allocation} \) is \text{dynamic}, then \text{convert-to-foreign-string} allocates memory for the string and pointer within the scope of the body of \text{with-dynamic-foreign-objects} and additional values, \text{length} and \text{byte-count} are returned. This is equivalent to using \text{convert-to-dynamic-foreign-string}. Otherwise, the allocation is static and \text{length} and \text{byte-count} are not returned.

See also

\text{convert-from-foreign-string}
\text{set-locale}
\text{set-locale-encodings}
\text{with-foreign-string}


5.2.5 Calling a C function that takes an array of strings

---

**copy-pointer**

*Function*

**Summary**

Returns a copy of a pointer object.

**Package**

\text{fli}

**Signature**

\text{copy-pointer} \hspace{0.5em} \text{pointer} \hspace{0.5em} \text{&key} \hspace{0.5em} \text{type} \hspace{0.5em} \text{pointer-type} \Rightarrow \text{copy}

**Arguments**

- \text{pointer}:
  A pointer to copy.
- \text{type}:
  A FLI type descriptor.
- \text{pointer-type}:
  A FLI pointer type descriptor.

**Values**

- \text{copy}:
  A copy of \text{pointer}.

**Description**

The function \text{copy-pointer} returns a copy of \text{pointer}.

If \text{type} is supplied, then it is used as the FLI type that \text{copy} points to. Alternatively, if \text{pointer-type} is supplied, then it must be a FLI pointer type and it is used as the pointer type of \text{copy}. If neither \text{type} nor \text{pointer-type} are supplied then the type of \text{copy} is the same as \text{pointer}. An error is signalled if both \text{type} and \text{pointer-type} are supplied.
Examples

In the following example a pointer `point1` is created, pointing to a `:char` type. The variable `point2` is set equal to `point1` using `setq`, whereas `point3` is set using `copy-pointer`. When `point1` is changed using `incf-pointer`, `point2` changes as well, but `point3` remains the same.

```lisp
(setq point1 (fli:allocate-foreign-object :type :char))
(setq point2 point1)
(setq point3 (fli:copy-pointer point1))
(fli:incf-pointer point1)
```

The results of this can be seen by evaluating `point1`, `point2`, and `point3`.

The reason for this behavior is that `point1` and `point2` are Lisp variables containing the same foreign pointer object, a pointer to a `char`, whereas `point3` contains a copy of the foreign pointer object.

See also

- `make-pointer`
- `with-coerced-pointer`
- 3.1.2 Copying pointers

---

**decf-pointer**

*Function*

**Summary**

Decreases the address held by a pointer.

**Package**

`fli`

**Signature**

```lisp
(decf-pointer pointer &optional delta => pointer)
```

**Arguments**

- `pointer` A FLI pointer.
- `delta` An integer. The default is 1.

**Values**

- `pointer` The pointer passed.

**Description**

The function `decf-pointer` decreases the address held by `pointer`. If `delta` is not given the address is decreased by the size
of the type pointed to by \textit{pointer}. The address can be decreased by a multiple of the size of the type by specifying a value for \textit{delta}. If the size of the type is 0 then an error is signalled.

The function \texttt{decf-pointer} is often used to move a pointer through an array of values.

**Examples**

In the following example an array with 10 entries is defined. A copy of the pointer to the array is made, and is incremented and decremented.

\begin{verbatim}
(setq array-obj
   (fli:allocate-foreign-object :type :int
      :nelems 10
      :initial-contents '(0 1 2 3 4 5 6 7 8 9)))

(setq point1 (fli:copy-pointer array-obj))

(dotimes (x 9)
   (print (fli:dereference point1))
   (fli:incf-pointer point1))

(dotimes (x 9)
   (fli:decf-pointer point1)
   (print (fli:dereference point1)))
\end{verbatim}

**See also**

\texttt{incf-pointer}

3.4 An example of dynamic pointer allocation

---

**define-c-enum**

*Macro*

**Summary**

Defines a FLI enumerator type specifier corresponding to the C \texttt{enum} type.

**Package**

\texttt{fli}

**Signature**

\texttt{define-c-enum name-and-options &rest enumerator-list => list}

\texttt{name-and-options ::= name | (name option*)}

\texttt{option ::= (:foreign-name foreign-name) | (:forward-reference-p forward-reference-p)}

\texttt{enumerator-list ::= {entry-name | (entry-name entry-value)}*}

**Arguments**

\begin{itemize}
  \item \texttt{enumerator-list}  Symbols, possibly with integer values, constituting the enumerator type.
  \item \texttt{name} A symbol naming the new enumeration type specifier.
  \item \texttt{foreign-name} A string specifying the foreign name of the type.
\end{itemize}
7 Function, Macro and Variable Reference

forward-reference-p \(\Downarrow\) A boolean.
entry-name \(\Downarrow\) A symbol.
entry-value \(\Downarrow\) An integer value for an entry-name.

Values
list The list (:enum name).

Description
The macro define-c-enum is used to define a FLI enumerator type specifier, which corresponds to the C enum type. It is a convenience function, as an enumerator type could also be defined using define-foreign-type.

The FLI type specifier is named by name, with optional foreign name foreign-name.

Each entry in enumerator-list can either consist of a symbol entry-name, in which case the first entry has an integer value of 0, or of a list of a symbol entry-name and its corresponding integer value entry-value.

When forward-reference-p is true, the new type specifier is defined as a forward reference type and descriptions can be empty. See define-foreign-forward-reference-type.

Examples
In the following example a FLI enumerator type specifier is defined, and the corresponding definition for a C enumerator type follows.

\[
\begin{align*}
&\text{(define-c-enum colors red green blue)} \\
&\text{enum colors \{ red, green, blue\};}
\end{align*}
\]

The next example illustrates how to start the enumerator value list counting from 1, instead of from the default start value of 0.

\[
\begin{align*}
&\text{(define-c-enum half_year (jan 1) feb mar apr may jun)} \\
&\text{enum half_year \{ jan = 1, feb, mar, apr, may, jun \}}
\end{align*}
\]

See also
define-c-struct
define-c-typedef
define-c-union
define-foreign-type
enum-symbol-value
2 FLI Types
define-c-struct

Macro

Summary
 Defines a FLI structure type specifier corresponding to the C `struct` type.

Package
 fli

Signature

```
define-c-struct name-and-options &rest descriptions => list
name-and-options ::= name | (name option*)
option ::= (:foreign-name foreign-name) | (:forward-reference-p forward-reference-p)
descriptions ::= {slot-description | byte-packing | aligned}*
slot-description ::= (slot-name slot-type)
byte-packing ::= (:byte-packing nbytes)
aligned ::= (:aligned nbytes)
```

Arguments

- **name**
  A symbol naming the new structure type specifier.
- **foreign-name**
  A string specifying the foreign name of the structure.
- **forward-reference-p**
  A boolean.
- **byte-packing**
  A list specifying byte packing for the subsequent slots.
- **slot-name**
  A symbol naming the slot.
- **slot-type**
  The foreign type of the slot.
- **nbytes**
  The number of 8-bit bytes to pack.

Values

- **list**
  The list `(:struct name)`.

Description

The macro `define-c-struct` is used to define a FLI structure type specifier, which corresponds to the C `struct` type. It is a convenience function, as a structure type could also be defined using `define-foreign-type`.

A structure is an aggregate type, or collection, of other FLI types. The types contained in a structure are referred to as slots, and can be accessed using the `foreign-slot-type` and `foreign-slot-value` functions.

The FLI type specifier is named by `name`, with optional foreign name `foreign-name`.

Each `slot-description` is a list of a symbol `slot-name` and a corresponding FLI type descriptor `slot-type` which is the type of the slot named by `slot-name`. 
Some C compilers support pragmas such as:

```
#pragma pack(1)
```

which causes fields in a structure to be aligned on a byte boundary even if their natural alignment is larger. This can be achieved from Lisp by specifying suitable byte-packing forms in the structure definition, as in the example below. Each byte-packing form specifies the packing for each slot-description that follows it in the `define-c-struct` form. It is important to use the same packing as the C header file containing the foreign type.

An aligned form specifies that the next slot must be aligned on nbytes bytes. Note that this affects only the alignment of the next slot. It does not affect the length of the slot, or the alignment of other slots. You will need this when the slot is made to be aligned, for example in gcc a slot defined like this:

```
int slot_name __attribute__((aligned (16))) ;
```

needs to be aligned on 16 bytes, even though the native alignment of the type int is 4.

When forward-reference-p is true, the new type specifier is defined as a forward reference type and descriptions can be empty. See `define-foreign-forward-reference-type`.

**Notes**

`foreign-name`, specifying the foreign name, is supported only for documentation purposes.

**Examples**

The first example shows a C structure definition and the corresponding FLI definition:

```
struct a-point {
    int x;
    int y;
    byte color;
    char ident;
};

(fli:define-c-struct a-point (x :int) (y :int) (color :byte) (ident :char))
```

The second example shows how you might retrieve data in Lisp from a C function that returns a structure:

```
struct 3dvector
{
    float x;
    float y;
    float z;
}

static 3dvector* vector;

3dvector* fn ()
{
    return vector;
}

(fli:define-c-struct 3dvector (x :float) (y :float) (z :float))
```
Finally an example to illustrate byte packing. This structure will require 4 bytes of memory because the field named `a-short` will be aligned on a 2 byte boundary and hence a byte will be wasted after the `a-byte` field:

```lisp
(fli:define-c-struct foo ()
  (a-byte (:unsigned :byte))
  (a-short (:unsigned :short)))
```

After adding `byte-packing`, the structure will require only 3 bytes:

```lisp
(fli:define-c-struct foo
  (:byte-packing 1)
  (a-byte (:unsigned :byte))
  (a-short (:unsigned :short)))
```

See also

- `define-c-enum`
- `define-c-typedef`
- `define-c-union`
- `define-foreign-type`
- `foreign-slot-names`
- `foreign-slot-type`
- `foreign-slot-value`

## 2 FLI Types

### define-c-typedef

**Macro**

**Summary**

Defines FLI type specifiers corresponding to type specifiers defined using the C `typedef` command.

**Package**

`fli`

**Signature**

```
define-c-typedef name-and-options type-description => name
```

**Arguments**

- `type-description` [An FLI type descriptor.]
- `name` [A symbol naming the new FLI type.]
7 Function, Macro and Variable Reference

**foreign-name**

A string specifying the foreign name of the type.

**Values**

**name**

The name of the new FLI type.

**Description**

The macro `define-c-typedef` is used to define FLI type specifiers, which corresponds to those defined using the C function `typedef`. It is a convenience function, as types can also be defined using `define-foreign-type`.

The FLI type specifier is named by `name`, with optional foreign name `foreign-name`.

`type-description` is not evaluated and the FLI type `name` will have no parameters.

**Notes**

`foreign-name`, specifying the foreign name, is supported only for documentation purposes.

**Examples**

In the following example three types are defined using the FLI function `define-c-typedef`, and the corresponding C definitions are then given.

```lisp
(fli:define-c-typedef intptr (:pointer :int))
(fli:define-c-typedef bar (:struct (one :int)))
```

These are the corresponding C `typedef` definitions:

```c
typedef int *intptr;
typedef struct {int one;} bar;
```

See also

- `define-c-enum`
- `define-c-struct`
- `define-c-union`
- `define-foreign-type`

2 FLI Types

**define-c-union**

Macro

**Summary**

Defines a FLI union type corresponding to the C `union` type.

**Package**

fli

**Signature**

```lisp
define-c-union name-and-options &rest slot-descriptions => list
```
7 Function, Macro and Variable Reference

name-and-options ::= name | (name option*)

option ::= (:foreign-name foreign-name) | (:forward-reference-p forward-reference-p)

slot-descriptions ::= {slot-description}*

slot-description ::= (slot-name slot-type)

Arguments

name
A symbol naming the new union type descriptor.

foreign-name
A string specifying the foreign name of the type.

forward-reference-p
A boolean.

slot-name
A symbol naming the slot.

slot-type
The FLI type of the slot.

Values

list
The list (:union name).

Description

The macro define-c-union is used to define a FLI union type specifier, which corresponds to the C union type. It is a convenience function, as a union type could also be defined using define-foreign-type.

A union is an aggregate type, or collection, of other FLI types. The types contained in a union are referred to as slots, and can be accessed using the foreign-slot-type and foreign-slot-value functions.

The FLI type specifier is named by name, with optional foreign name foreign-name.

Each slot-description is a list of a symbol slot-name and a corresponding FLI type descriptor slot-type which is the type of the slot named by slot-name.

When forward-reference-p is true, the new type specifier is defined as a forward reference type and descriptions can be empty. See define-foreign-forward-reference-type.

Notes

foreign-name, specifying the foreign name, is supported only for documentation purposes.

Examples

In the following example a union is defined using define-c-union, and the corresponding C code is given.

```
(fli:define-c-union a-point (x :int)
  (color :byte)
  (ident :char))

union a-point {
  int x;
  byte color;
  char ident;
};
```
define-foreign-block-callable-type

Summary
Defines a type for foreign blocks, in LispWorks for Macintosh.

Package
fli

Signature
define-foreign-block-callable-type name result-type arg-types => name

Arguments

- name : A symbol.
- result-type : A foreign type specifier.
- arg-types : A list of foreign type specifiers.

Values

- name : Symbol.

Description

The macro \texttt{define-foreign-block-callable-type} defines a type for foreign blocks.

\textit{name} specifies the name of the type. It must not be the same as the name of a \texttt{define-foreign-callable}.

\textit{result-type} specifies the type of the result of the foreign block.

\textit{arg-types} specifies the types of the arguments that a block of type \textit{name} takes. These must correspond to the arguments types with which the block is called from the foreign call.

Note that \textit{arg-types} specifies the types for a call from foreign code into Lisp, which affects the way \texttt{:reference-return} and \texttt{:reference-pass} are used. If the block is called from the foreign code with a pointer and you want to treat it as pass-by-reference, you need to use \texttt{:reference-return} (like \texttt{define-foreign-callable} does). See the qsort_b example in:

\begin{verbatim}
(exexample-edit-file "fli/foreign-blocks")
\end{verbatim}

\texttt{define-foreign-block-callable-type} returns \textit{name}.  

7 Function, Macro and Variable Reference

See also

\texttt{define-c-enum}  
\texttt{define-c-struct}  
\texttt{define-c-typedef}  
\texttt{define-foreign-type}

2 FLI Types
Notes

`define-foreign-block-callable-type` is implemented in LispWorks for Macintosh only.

See also

`allocate-foreign-block`
`with-foreign-block`
`with-local-foreign-block`

5.7 Block objects in C (foreign blocks)

---

```lisp
define-foreign-block-invoker
```

Macro

**Summary**

Defines an invoker of a foreign block, in LispWorks for Macintosh.

**Package**

`fli`

**Signature**

```lisp
define-foreign-block-invoker the-name args &key lambda-list documentation result-type language no-check calling-convention
```

**Arguments**

- `the-name`\(\downarrow\) A symbol.
- `args`\(\downarrow\) A lambda list.
- `lambda-list`\(\downarrow\) The lambda list to be used for the defined Lisp function.
- `documentation`\(\downarrow\) A string.
- `result-type`\(\downarrow\) A foreign type.
- `language`\(\downarrow\) The language in which the foreign source code is written. The default is :ansi-c.
- `no-check`\(\downarrow\) A boolean.
- `calling-convention`\(\downarrow\) Specifies the calling convention used.

**Description**

The macro `define-foreign-block-invoker` defines an invoker of a foreign block.

It defines `the-name` to be a function that can be used to invoke foreign blocks which takes arguments that match `args`. The block is then invoked by simply calling the function `the-name` with the block and arguments:

```
(the-name block arg1 arg2 ...)
```

The `block` argument is of type `foreign-block-pointer`.

`define-foreign-block-invoker` is very similar to `define-foreign-funcallable` and `define-foreign-function`, which specify how `documentation`, `result-type`, `language`, `no-check` and `calling-convention`
7 Function, Macro and Variable Reference

are used.

Notes

The lambda list of the invoker is \((block . args)\). When \(lambda-list\) is supplied, \texttt{define-foreign-block-invoker}\ inserts in front of the supplied \(lambda-list\) an additional argument for the block. Therefore a supplied \(lambda-list\) must not include an argument for the block. Similarly a supplied \(lambda-list\) in \texttt{define-foreign-funcallable} should not include an argument for the function.

\texttt{define-foreign-block-invoker} returns \texttt{the-name}.

\texttt{define-foreign-block-invoker} is implemented in LispWorks for Macintosh only.

Examples

\begin{verbatim}
  (example-edit-file "fli/foreign-blocks")

  (example-edit-file "fli/invoke-foreign-block")
\end{verbatim}

See also

\texttt{define-foreign-funcallable}

\texttt{define-foreign-function}

\texttt{foreign-block-pointer}

5.7 Block objects in C (foreign blocks)

\texttt{define-foreign-callable} \hspace{1cm} \textit{Macro}

Summary

Defines a Lisp function which can be called from a foreign language.

Package

\texttt{fli}

Signature

\begin{verbatim}
define-foreign-callable (foreign-name &key encode language result-type result-pointer no-check calling-convention) ((arg)*) &body body => lisp-name
\end{verbatim}

Arguments

\begin{verbatim}
foreign-name : A string or symbol naming the Lisp callable function created.
encode : One of \texttt{:source, :object, :lisp} or \texttt{dbcs}.
language : The language in which the foreign calling code is written. The default is \texttt{:ansi-c}.
result-type : The FLI type of the Lisp foreign callable function's return value which is passed back to the calling code.
\end{verbatim}
7 Function, Macro and Variable Reference

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result-pointer</td>
<td>A variable which will be bound to a foreign pointer into which the result should be written when the result-type is an aggregate type.</td>
</tr>
<tr>
<td>no-check</td>
<td>A boolean.</td>
</tr>
<tr>
<td>calling-convention</td>
<td>Specifies the calling convention used on Windows and ARM.</td>
</tr>
<tr>
<td>arg</td>
<td>The arguments of the Lisp foreign callable function. Each argument can consist either of an <code>arg-name</code>, in which case LispWorks assumes it is an :int, or an <code>arg-name</code> and an <code>arg-type</code>, which is a FLI type.</td>
</tr>
<tr>
<td>body</td>
<td>A list of forms which make up the Lisp foreign callable function.</td>
</tr>
<tr>
<td>arg-name</td>
<td>A Lisp symbol.</td>
</tr>
<tr>
<td>arg-type</td>
<td>A FLI type.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lisp-name</td>
<td>A string or symbol naming the Lisp callable function created.</td>
</tr>
</tbody>
</table>

Description

The macro `define-foreign-callable` defines a Lisp function that can be called from a foreign language, for example from a C function. When the C function is called, data passed to it is converted to the appropriate FLI representation, which is translated to an appropriate Lisp representation for the Lisp part of the function. Once the callable function exits, any return values are converted back into a FLI format to be passed back to the calling language.

When you use `:reference` with `:lisp-to-foreign-p t` as an `arg-type`, you need to set `arg-name` to the value that you want to return in that reference. That value is then converted and stored into the pointer supplied by the calling foreign function. This is done after the visible body of your `define-foreign-callable` form returns.

If `no-check` is `nil`, the result of the foreign callable function, produced by `body`, is checked to see if matches `result-type`, and an error is raised if they do not match. If `no-check` is `t` then this check is not done and the effect will be undefined if the types do not match.

`calling-convention` is ignored on platforms other than Windows and ARM, where there is no calling convention issue. On 32-bit Windows, `stdcall` is the calling convention used to call Win32 API functions and matches the C declarator "stdcall". This is the default value. `cdecl` is the default calling convention for C/C++ programs and matches the C declarator "cdecl". See 4.2.1 Windows 32-bit calling conventions for details.

On ARM platforms, there is also more than one calling convention, but normally you do not need to specify it. See 4.2.2 ARM 32-bit calling conventions and 4.2.3 ARM 64-bit calling conventions for details.

When `result-type` is an aggregate type, an additional variable is bound in the body to allow the value of the function to be returned (the value returned by the body is ignored). This argument is named after `result-pointer` or is named `result-pointer` in the current package if unspecified. While the body is executing, the variable will be bound to a foreign pointer that points to an object of the type `result-type`. The body must set the slots in this foreign object in order for the value to be returned to the caller.

To make a function pointer referencing a foreign callable named "Foo", use:

```
(make-pointer :symbol-name "Foo")
```

By default, LispWorks performs automatic name encoding to translate `foreign-name`. If you want to explicitly specify an encoding, `encode` can be one of the following:

```
:symbol-name foreign-name is the name of the function in the foreign source code. This is the default value of `encode` when `foreign-name` is a string.
```

77
foreign-name is the literal name of the function in the foreign object code.

If foreign-name is a Lisp symbol, it must be translated and encoded. This is the default value of encode if foreign-name is a symbol.

A suffix is automatically appended to the function name depending on the Windows operating system that LispWorks runs in. The suffix is "A" for Windows 95-based systems and "W" for Windows NT-based systems.

Notes

1. For a delivered application where the string name of your foreign callable is not passed in dll-exports, be aware that a call to make-pointer like that above will not retain the foreign callable in a delivered application. Internally a Lisp symbol named |%FOREIGN-CALLABLE/Foo| is used so you could retain that explicitly (see the Delivery User Guide for details, and take care to specify the package). However it is simpler to name the foreign callable with your Lisp symbol, and pass that to make-pointer. This call will keep your foreign callable in the delivered application:

   (make-pointer :symbol-name 'foo :functionp t)

2. If you specify any of the FLI float types :float, :double, :lisp-float, :lisp-single-float and so on, then the value of language should be :ansi-c.

Compatibility note

64-bit integer types such as (:long :long), :int64 and :uint64 are now supported for arg-type in define-foreign-callable in 32-bit LispWorks. In 32-bit LispWorks 6.1 and earlier versions, these types could only be used by define-foreign-function.

Examples

The following example demonstrates the use of foreign callable. A foreign callable function, square, is defined, which takes an integer as its argument, and returns the square of the integer.

```lisp
(fli:define-foreign-callable
  ("square" :result-type :int)
  ((arg-1 :int)) (* arg-1 arg-1))
```

The foreign callable function, square, can now be called from a foreign language. We can mimic a foreign call by using the define-foreign-function macro to define a FLI function to call square.

```lisp
(fli:define-foreign-function (call-two "square")
  ((in-arg :int)) :result-type :int)
```

The call-two function can now be used to call square. The next command is an example of this.

```lisp
(call-two 9)
```

This last example shows how the address of a foreign callable can be passed via a pointer object, which is how you use foreign callables in practice. The foreign library in this example is libgsl:

```lisp
(fli:define-foreign-callable ("gsl-error-handler")
  ((reason (:reference-return :ef-mb-string))
   (file (:reference-return :ef-mb-string))
   (lineno :integer)
   (gsl-errno :integer))
  (error
   (gsl-errno :integer)))
```
"Error number ~a inside GSL [file: ~a, lineno ~a]: ~a"
gsl-errno file lineno reason))

(fli:define-foreign-function gsl-set-error-handler
  ((func :pointer))
  :result-type :pointer)

To set the error handler, you would do:

(gsl-set-error-handler
  (fli:make-pointer :symbol-name "gsl-error-handler"))

See also

define-foreign-function
define-foreign-variable
make-pointer
4 Defining foreign functions and callables
5.7.2 Operations on foreign blocks

define-foreign-converter

Macro

Summary

Defines a new FLI type specifier that converts to or from another type specifier.

Package

fli

Signature

define-foreign-converter type-name lambda-list object-names &key foreign-type foreign-to-lisp lisp-to-foreign predicate tested-value error-form documentation => type-name

doctment-o-::= object-name | (lisp-object-name foreign-object-name)

Arguments

type-name↓ A symbol naming the new FLI type.
lambda-list↓ A lambda list which is the argument list of the new FLI type.
object-names↓ A symbol or a list of two symbols.
foreign-type↓ A macro expansion form that evaluates to a FLI type descriptor.
foreign-to-lisp↓ A macro expansion form to convert between Lisp and the FLI.
lisp-to-foreign↓ A macro expansion form to convert between the FLI and Lisp.
predicate↓ A macro expansion form to check whether a Lisp object is of this type.
tested-value↓ A macro expansion form to give an error if a Lisp object is not of this type.
error-form↓ A macro expansion form to give an error if predicate returns false.
documentation↓ A string.
object-name, lisp-object-name, foreign-object-name

Lisp symbols.

Values

type-name

The name of the new FLI converter type.

Description

Note: this macro is for advanced use of the FLI type system. See define-foreign-type for simple aliasing of FLI type descriptors.

The macro define-foreign-converter defines a new FLI type specifier type-name that wraps another FLI type specifier and optionally performs data conversion and type checking. The string documentation is associated with type-name with the define-foreign-type documentation type.

The lambda list of the new FLI type specifier is lambda-list and its variables are available for use in foreign-type, foreign-to-lisp, lisp-to-foreign, predicate and tested-value.

If object-names is a symbol object-name, then it provides the name of a variable for use in all of the macro expansion forms. Otherwise object-names should be a list of the form (lisp-object-name foreign-object-name), where lisp-object-name provides the name of a variable for use in lisp-to-foreign, predicate and tested-value forms and foreign-object-name provides the name of a variable for use in foreign-to-lisp.

When the new FLI type is used, foreign-type is evaluated to determine the underlying FLI type descriptor to be converted. It can use variables bound by lambda-list, but not object-names.

When type-name is used to convert a foreign value to Lisp (for example when as the result-type in define-foreign-function), foreign-to-lisp is evaluated to determine how the conversion should be made. It works like a macro expansion function, so should return a form that converts the foreign value, which will be bound to object-name (or foreign-object-name). It can use variables bound by lambda-list.

When type-name is used to convert a Lisp value to a foreign value (for example in the argument list of define-foreign-function), the type of the Lisp value can be checked before conversion using tested-value and predicate and then converted using lisp-to-foreign as detailed below.

If tested-value is specified, it is used as a macro expansion function that returns a form that must return object-name (or lisp-object-name) if it is of the required type or give an error. It can use variables bound by lambda-list, but not object-names.

Otherwise, if predicate is specified, it is used as a macro expansion function that returns a form that must return true if object-name (or lisp-object-name) is of the required type. If predicate is specified, then error-form can be specified as a macro expansion function that signals an error about object-name (or lisp-object-name) not being of the required type. If error-form is omitted, a default error is signaled. Both predicate and error-form can use variables bound by lambda-list, but not object-names.

If both tested-value and predicate are omitted, then no type checking is performed.

After type checking, lisp-to-foreign is used as a macro expansion function that returns a form that converts the Lisp object object-name (or lisp-object-name) to the underlying FLI type foreign-type. It can use variables bound by lambda-list, but not object-names.

Examples

This defines a FLI type (real-double lisp-type), which allows any real value in Lisp to be passed to foreign code as a double precision float. When a foreign value is converted to Lisp, it is coerced to type:
This defines a FLI type \texttt{int-signum}, which uses -1, 0 and 1 for values on the foreign side. There is no \texttt{foreign-to-lisp} form specified, so it will return these values to Lisp too:

\begin{verbatim}
(fli:define-foreign-converter int-signum () object
   :foreign-type :int
   :foreign-to-lisp `(signum ,object))
\end{verbatim}

This defines a FLI type \texttt{(bigger-in-lisp \textit{n})}, which is an integer type for values that are \textit{n} bigger in Lisp than on the foreign side.

\begin{verbatim}
(fli:define-foreign-converter bigger-in-lisp (&optional (n 1)) object
   :foreign-type :int
   :foreign-to-lisp `(+ ,object ,n)
   :lisp-to-foreign `(- ,object ,n)
   :predicate `(integerp ,object))
\end{verbatim}

\begin{verbatim}
(fli:with-dynamic-foreign-objects ((x :int 10))
   (fli:dereference x :type '(bigger-in-lisp 2))) => 12
\end{verbatim}

See also

\texttt{define-foreign-type}  
\texttt{define-opaque-pointer}  
\texttt{:wrapper}  
\texttt{2.3 Parameterized types}  

\section*{define-foreign-forward-reference-type} \textit{Macro}

Summary

Defines a FLI type specifier if it is not already defined.

Package

\texttt{fli}

Signature

\texttt{define-foreign-forward-reference-type type-name lambda-list \&body forms => type-name}

Arguments

\begin{itemize}
  \item \texttt{type-name} A symbol naming the new FLI type.
  \item \texttt{lambda-list} A lambda list which is the argument list of the new FLI type.
  \item \texttt{forms} One or more Lisp forms which provide a definition of the new type.
\end{itemize}
Values

type-name

The name of the FLI type.

Description

The macro `define-foreign-forward-reference-type` defines a new FLI type called `type-name`, unless `type-name` is already defined. This macro is useful when a type declaration is needed but the full definition is not yet available.

`lambda-list` and `forms` are used as in `define-foreign-type`.

See also

- `define-foreign-type`
- `define-opaque-pointer`

---

**define-foreign-funcallable**  
*Macro*

Summary

Defines a Lisp function which, when passed a pointer to a foreign function, calls it.

Package

`fli`

Signature

```
define-foreign-funcallable the-name args &key lambda-list documentation result-type language no-check calling-convention variadic-num-of-fixed => the-name
```

```
args ::= ({arg}* )
```

Arguments

- `the-name`  
  A symbol naming the Lisp function.

- `lambda-list`  
  The lambda list to be used for the defined Lisp function.

- `documentation`  
  A documentation string for the foreign function.

- `result-type`  
  A foreign type.

- `language`  
  The language in which the foreign source code is written. The default is `:ansi-c`.

- `no-check`  
  A boolean.

- `calling-convention`  
  Specifies the calling convention used.

- `variadic-num-of-fixed`  
  `nil` or a non-negative integer.

- `arg`  
  Argument specifier as in `define-foreign-function`.

Values

- `the-name`  
  A symbol naming the Lisp function.
Description

The macro `define-foreign-funcallable` is like `define-foreign-function`, but creates a function with an extra argument at the start of the argument list for the address to call.

See `define-foreign-function` for how `the-name`, `lambda-list`, `documentation`, `result-type`, `language`, `no-check`, `calling-convention`, `variadic-num-of-fixed` and `arg` are used.

Examples

Define a caller for this shape:

```lisp
(fli:define-foreign-funcallable
 call-with-string-and-int
 ([string (:reference-pass :ef-mb-string))
  (value :int))
```

Call `printf`. Note that the output goes to console output which is hidden by default:

```lisp
(let ((printf-func
  (fli:make-pointer :symbol-name "printf")))
  (call-with-string-and-int
   printf-func "printf called with %d" 1234))
```

See also

`define-foreign-function`

---

**define-foreign-function**

Macro

Summary

Defines a Lisp function which acts as an interface to a foreign function.

Package

`fli`

Signature

```lisp
define-foreign-function name ([arg]*) &key lambda-list documentation result-type result-pointer language no-check calling-convention module variadic-num-of-fixed => lisp-name
```

Arguments

`lambda-list` The lambda list to be used for the defined Lisp function.
A string.
A foreign type.
The name of the keyword argument that is added to the lambda-list of the Lisp function when \texttt{result-type} is an aggregate type.
The language in which the foreign source code is written. The default is :ansi-c.
A boolean.
Specifies the calling convention used.
A symbol or string naming the module in which the foreign symbol is defined.
\texttt{nil} or a non-negative integer.
A symbol naming the defined Lisp function.
A string or a symbol specifying the foreign name of the function.
A variable.
A foreign type name.
A Lisp object.
A foreign type name.
A Lisp object.

\textbf{Values}

\texttt{lisp-name} \hspace{1cm} A symbol naming the defined Lisp function.

\textbf{Description}

The macro \texttt{define-foreign-function} defines a Lisp function \texttt{lisp-name} which acts as an interface to a foreign language function, for example a C function. When the Lisp function is called its arguments are converted to the appropriate foreign representation before being passed to the specified foreign function. Once the foreign function exits, any return values are converted back from the foreign format into a Lisp format.

\texttt{encoding} specifies how \texttt{lisp-name} is translated into the function name in the foreign object code. Its values are interpreted as follows:

\begin{itemize}
  \item \textbf{source} \hspace{1cm} \texttt{foreign-name} is the name of the function in the foreign source code. This is the default value of \texttt{encoding} when \texttt{foreign-name} is a string.
  \item \textbf{object} \hspace{1cm} \texttt{foreign-name} is the literal name of the function in the foreign object code.
  \item \textbf{lisp} \hspace{1cm} If \texttt{foreign-name} is a Lisp symbol, it must be translated and encoded. This is the default value of \texttt{encoding} if \texttt{foreign-name} is a symbol.
  \item \textbf{dbsc} \hspace{1cm} A suffix is automatically appended to the function name depending on the Windows operating system that LispWorks runs in. The suffix is "A" for Windows 95-based systems and "W" for Windows NT-based systems.
\end{itemize}

The number and types of the arguments of \texttt{lisp-name} must be given. Lisp arguments may take any name, but the types must be accurately specified and listed in the same order as in the foreign function, unless otherwise specified using \texttt{lambda-list}.

If \texttt{arg} is a symbol \texttt{arg-name}, then \texttt{define-foreign-function} assumes that it is of type \texttt{:int}. Otherwise \texttt{arg-type} or \texttt{value-type} specifies the foreign type of the argument.
If arg is of the form (:constant value value-type) then value is always passed through to the foreign code, and arg is
omitted from the lambda list of lisp-name.

If arg is optional or &key, then the lambda list of the Lisp function lisp-name will contain these lambda-list-keywords
too. Any argument following optional or &key can use the
((arg-name default) arg-type) syntax to provide a default value default for arg-name.

If arg is of the form (:ignore arg-type) then nil is always passed through to the foreign code and arg is omitted from the
lambda list of lisp-name. This is generally only useful when arg-type is a :reference-return type, where the value nil
will be ignored.

If documentation is supplied then it is set as the function documentation for lisp-name.

When language is :ansi-c the foreign code is expected to be written in ANSI C. In particular single floats are passed
through as single-floats whereas language :c causes them to be passed through as double floats. Similarly :c causes double
floats to be returned from C and :ansi-c causes a single-floats to be returned. In both cases the type returned to Lisp is
determined by result-type.

If no-check is nil, then the types of the arguments provided when lisp-name is called are compared with the expected types
and an error is raised if they do not match. If no-check is t then this check is not done and the effect will be undefined if the
types do not match. If the compilation safety level is set to 0 then no-check defaults to t, otherwise it defaults to nil.

lambda-list allows you to define the order in which the Lisp function lisp-name takes its arguments to be different from the
order in which the foreign function takes them, and to use standard lambda list keywords such as optional even if they do
not appear in args. If lambda-list is not supplied, the lambda list of lisp-name is generated from the list of args.

If arg-type is a struct then the value arg-name can be either a foreign struct object or a pointer to a foreign struct object.

The :reference, :reference-pass and :reference-return types are useful with define-foreign-function. It is
fairly common for a C function to return a value by setting the contents of an argument passed by reference (that is, as a
pointer). This can be handled conveniently by using the :reference-return type, which dynamically allocates memory
for the return value and passes a pointer to the C function. On return, the pointer is dereferenced and the value is returned as
an extra multiple value from the Lisp function.

The :reference-pass type can be used to automatically construct an extra level of pointer for an argument. No extra
results are returned.

The :reference type is like :reference-return but allows the initial value of the reference argument to be set.

result-type optionally specifies the type of the foreign function’s return value. When result-type is an aggregate type, an
additional keyword argument is placed in the lambda-list of the Lisp function. This keyword is named after result-pointer or
is called :result-pointer if unspecified. When calling the Lisp function, a foreign pointer must be supplied as the value
of this keyword argument, pointing to an object of type result-type. The result of the foreign call is written into this object
and the foreign pointer is returned as the primary value from the Lisp function. This allows the caller to maintain control over the
time of this object (in C this would typically be stored in a local variable). If result-type is :void or is omitted, then no
value is returned.

calling-convention is ignored on some platforms, where there is no calling convention issue. On 32-bit Windows, :stdcall
is the calling convention used to call Win32 API functions and matches the C declarator "__stdcall". This is the default
value. :cdecl is the default calling convention for C/C++ programs and matches the C declarator "__cdecl". See 4.2.1
Windows 32-bit calling conventions for details.

On ARM platforms, there is also more than one calling convention, but normally you do not need to specify it. See 4.2.2
ARM 32-bit calling conventions and 4.2.3 ARM 64-bit calling conventions for details.

On 32-bit x86 platforms (including 32-bit Windows), the :fastcall calling convention can be use (see 4.2.4 Fastcall on 32
-bit x86 platforms for details).

If module is the name of a module registered using register-module then that module is used to look up the symbol.
Otherwise *module* should be a string, and a module named *module* is automatically registered and used to look up the symbol. Such automatically-registered modules have *connection-style:* *manual* - this prevents them being used by other *define-foreign-function* forms which do not specify a module.

When *variadic-num-of-fixed* a non-negative integer, it specifies that the foreign function that it is calling is variadic (like *printf*). The integer must be the number of fixed arguments that the foreign function takes. For *printf*, for example, you need to pass :*variadic-num-of-fixed* 1, and for *sprintf* you need :*variadic-num-of-fixed* 2. When *variadic-num-of-fixed* is *nil* (the default), then the function is specified to be not variadic. Calls to variadic function without using *variadic-num-of-fixed* work on some platforms, but not all. Thus you should always use it when calling variadic functions.

**Compatibility notes**

In LispWorks 4.4 and previous versions, the default value for *language* is :c. In LispWorks 5.0 and later, the default value is :ansi-c.

The :*fastcall calling-convention* was added in LispWorks 7.1.

*variadic-num-of-fixed* was added in LispWorks 7.1.

**Examples**

A simple example of the use of *define-foreign-function* is given in 1.2.2 Defining a FLI function. More detailed examples are given in 5 Advanced Uses of the FLI.

Here is an example using the :*reference-return* type.

Non-Windows version:

```lisp
(int cfloor (int x, int y, int *remainder)
  { int quotient = x/y;
    *remainder = x - y*quotient;
    return quotient;
  }
)
```

Windows version:

```c
__declspec(dllexport) int __cdecl cfloor(int x, int y, int *remainder)
  { int quotient = x/y;
    *remainder = x - y*quotient;
    return quotient;
  }
```

In this foreign function definition the main result is the quotient and the second return value is the remainder:

```lisp
(fli:define-foreign-function cfloor
  ((x :int)
   (y :int)
   (rem (:reference-return :int)))
  :result-type :int)

(cfloor 11 5 t)
=>
2,1
```

This example illustrates a use of the lambda list keyword *optional* and a default value for the optional argument:

```lisp
(define-foreign-function one-or-two-ints

86
```
The call `(one-or-two-ints 1 2)` passes 1 and 2.
The call `(one-or-two-ints 1)` passes 1 and 42.

See also

- `define-foreign-callable`
- `define-foreign-funcallable`
- `define-foreign-variable`
- `register-module`

4 Defining foreign functions and callables

---

**define-foreign-pointer**

Macro

Summary

Defines a new FLI pointer type.

Package

fli

Signature

```
define-foreign-pointer name-and-options points-to-type &rest slots => type-name
```

```
name-and-options ::= type-name | (type-name (option*))
```

```
option ::= (option-name option-value)
```

Arguments

- **points-to-type**
  A foreign type.
- **slots**
  Slots of the new type.
- **type-name**
  A symbol naming the new FLI type.
- **option-name**
  :allow-null or a `defstruct` option.
- **option-value**
  A symbol.

Values

- **type-name**
  The name of the new FLI pointer type.

Description

The macro `define-foreign-pointer` defines two things:

- An FLI pointer type `type-name`, which is a pointer to `points-to-type`.
- A Lisp type specifier `type-name` that is a foreign pointer.
The option `:allow-null` takes an `option-value` of either `t` or `nil`, defaulting to `nil`. It controls whether the type `type-name` accepts `nil`.

The other allowed values for `option-name` are the `defstruct` options `:conc-name`, `:constructor`, `:predicate`, `:print-object`, `:print-function`. In each case the symbol supplied as `option-value` provides the corresponding option for `type-name`.

`slo`ts is a list of `defstruct` slot-descriptions which become slots in `type-name`.

When LispWorks makes a foreign pointer of type `type-name`, then an object of Lisp type `type-name` is made instead of a foreign pointer object. This is useful if you want to associate extra Lisp data with the foreign pointer object, using `slots`.

See also

3.1.1 Creating pointers

---

**define-foreign-type**

Macro

Summary

Defines a new FLI type specifier.

Package

*fli*

Signature

```lisp
define-foreign-type name-and-options lambda-list &body forms => name
```

```lisp
name-and-options ::= name | (name option*)
```

```lisp
option ::= (:foreign-name foreign-name)
```

Arguments

```lisp
lambda-list↓
```

A lambda list which is the argument list of the new FLI type.

```lisp
forms↓
```

One or more Lisp forms which provide a definition of the new type.

```lisp
name↓
```

A symbol naming the new FLI type.

```lisp
foreign-name↓
```

A string specifying the foreign name of the type.

Values

```lisp
name
```

The name of the new FLI type.

Description

The macro `define-foreign-type` defines a new FLI type called `name`, with optional foreign name `foreign-name`.

When `name` is used as a foreign type, `forms` are evaluated as an implicit `progn` with the variables in `lambda-list` bound to the arguments of the foreign type. The value returned by `forms` is used as the definition of the foreign type. This is similar to how `deftype` works for Lisp type specifiers.
Notes

foreign-name, specifying the foreign name, is supported only for documentation purposes.

Examples

In the following example an integer array type specifier is defined. Note that the type takes a list as its argument, and uses this to determine the size of the array.

```lisp
(fli:define-foreign-type :int-array (dimensions)
  `(:c-array :int ,@dimensions))

(setq number-array (fli:allocate-foreign-object
  :type `(:int-array (2 2))))
```

In the next example a boolean type, called :bool, with the same size as an integer is defined.

```lisp
(fli:define-foreign-type :bool () `(:boolean :int))

(fli:size-of :bool)
```

See also

define-c-typedef
define-foreign-converter
define-foreign-forward-reference-type
foreign-type-equal-p
2 FLI Types
5.4 Defining new types

**define-foreign-variable**  
*Macro*

Summary

Defines a Lisp function to access a variable in foreign code.

Package

fli

Signature

`define-foreign-variable` `the-name` &key `type` `accessor` `language` `no-check` `module` => `lisp-name`

- `the-name` ::= `lisp-name` | `(lisp-name foreign-name [encoding])`
- `encoding` ::= :source | :object | :lisp | :dbc
- `accessor` ::= :value | :address-of | :read-only | :constant
- `language` ::= :c | :ansi-c`
Arguments

the-name
Names the Lisp function which is used to access the foreign variable.

type
The FLI type corresponding to the type of the foreign variable to which Lisp is interfacing.
The default is :int.

language
The language in which the foreign source code for the variable is written. The default is :ansi-c.

no-check
A boolean.

module
A string or symbol naming the module in which the foreign variable is defined.

lisp-name
A symbol naming the Lisp accessor.

foreign-name
A string or a symbol specifying the foreign name of the variable.

Values

lisp-name
A symbol naming the Lisp accessor.

Description

The macro define-foreign-variable defines a Lisp accessor lisp-name which can be used to get and set the value of a variable defined in foreign code.

accessor specifies what kind of accessor is generated for the variable. It can be one of the following:

:value
The value of the foreign variable is returned directly and is the default when type is a non-aggregate type. If type is an aggregate type, then a copy of the object is allocated using allocate-foreign-object, and the copy is returned. In general, it is more useful to use accessor :address-of for aggregate types, to allow the original aggregate to be updated.

:address-of
Returns an FLI pointer pointing to the foreign variable.

:read-only
Ensures that no setf expander is defined for the variable, which means that its value can be read, but it cannot be set.

:constant
Is like :read-only and will return a constant value. For example, this is more efficient for a variable that always points to the same string.

If the foreign variable has a type corresponding to an FLI aggregate type, then accessor must be supplied (there is no default).

encoding controls how the Lisp variable name is translated to match the foreign variable name in the foreign DLL. encoding can be one of the following:

:source
Tells LispWorks that foreign-name is the name of the variable in the foreign source code. This is the default value of encoding when foreign-name is a string.

:object
Tells LispWorks that foreign-name is the literal name of the variable in the foreign object code.

:lisp
Tells LispWorks that if foreign-name is a Lisp symbol, it must be translated and encoded. This is the default value of encoding if foreign-name is a symbol.

:dbs
Modifies the variable name on Windows, as described for define-foreign-function.

If no-check is nil, then the type of the value is provided to the setf expander for lisp-name is compared with type and an error is raised if it does not match. If no-check is t then this check is not done and the effect will be undefined if the type does not match. If the compilation safety level is set to 0 then no-check defaults to t, otherwise it defaults to nil.
Notes

If you specify any of the FLI float types: \texttt{float}, \texttt{double}, \texttt{lisp-float}, \texttt{lisp-single-float} and so on, then the value of \textit{language} should be \texttt{ansi-c}.

\textit{module} is processed as for \texttt{define-foreign-function}.

Examples

The following example illustrates how to use the FLI to define a foreign variable, given the following C variable in a DLL:

\begin{verbatim}
int num;
\end{verbatim}

The first example defines a Lisp variable, \texttt{num1}, to interface with the C variable \texttt{num}.

\begin{verbatim}
(fli:define-foreign-variable (num1 "num") :type :int)
\end{verbatim}

The following commands return the value of \texttt{num}, and increase its value by 1:

\begin{verbatim}
(num1)

(incf (num1))
\end{verbatim}

In the next example, the Lisp variable \texttt{num2} interfaces with \texttt{num} in a read-only manner.

\begin{verbatim}
(fli:define-foreign-variable (num2 "num")
 :type :int :accessor :READ-ONLY)
\end{verbatim}

In this case, the next command still returns the value of \texttt{num}, but the second command raises an error, because \texttt{num2} is read-only.

\begin{verbatim}
(num2)

(incf (num2))
\end{verbatim}

The final example defines a Lisp variable, \texttt{num3}, which accesses \texttt{num} through pointers.

\begin{verbatim}
(fli:define-foreign-variable (num3 "num")
 :type :int :accessor :address-of)
\end{verbatim}

As a result, the next command returns a pointer to \texttt{num}, and to obtain the actual value stored by \texttt{num}, \texttt{num3} needs to be dereferenced.

\begin{verbatim}
(num3)

(fli:dereference (num3))
\end{verbatim}

See also

\texttt{define-foreign-callable}
\texttt{define-foreign-function}
**define-opaque-pointer**

**Macro**

**Summary**

Defines an opaque foreign pointer type.

**Package**

`fli`

**Signature**

`define-opaque-pointer pointer-type structure-type`

**Arguments**

- `pointer-type` ⇓ A symbol.
- `structure-type` ⇓ A symbol.

**Description**

The macro `define-opaque-pointer` defines an opaque foreign pointer type named `pointer-type` and foreign structure type with a name based on `structure-type`. An opaque pointer is a pointer to a structure which does not have a structure description. It is the equivalent to the C declaration:

```
typedef struct structure-type *pointer-type;
```

An opaque pointer is useful for dealing with pointers that are returned by foreign functions and are then passed to other foreign functions. It checks the type of the foreign pointer, and thus prevents passing pointers of the wrong type.

**Examples**

Using the C standard file* pointer:

```
(fli:define-opaque-pointer file-pointer file)

(fli:define-foreign-function fopen
   ((name (:reference-pass :ef-mb-string))
    (mode (:reference-pass :ef-mb-string)))
   :result-type file-pointer)

(fli:define-foreign-function fgetc
   ((file file-pointer))
   :result-type :int)

(fli:define-foreign-function fclose
   ((file file-pointer)))

(fli:define-foreign-function fgets
   ((string (:reference-return (:ef-mb-string :limit 200)))
    (:constant 200 :int)
    (file file-pointer))
   :result-type (:pointer-integer :int)
   :lambda-list (file &aux string))
```
(defun print-a-file (name)
  (let ((file-pointer (fopen name "r")))
    (if (fli:null-pointer-p file-pointer)
        (error "failed to open ~a" name)
      (unwind-protect
          (loop (multiple-value-bind (res line)
                  (fgets file-pointer)
            (when (zerop res) (return))
            (princ line))))
        (fclose file-pointer)))))

See also
define-foreign-type

derereference

Accessor

Summary
Accesses and returns the value of a foreign object.

Package
fli

Signature
derereference  pointer  &key  index  type  copy-foreign-object  =>  value

(setf  (dereference  pointer  &key  index  type  copy-foreign-object)  value  =>  value)

Arguments
pointer↓  An instance of a FLI pointer.
index↓   An integer.
type↓    A foreign type.
copy-foreign-object↓ One of t, nil or :error.
value    The value of the dereferenced object at pointer.

Values
value    The value of the dereferenced object at pointer.

Description
The accessor dereference accesses and returns the value of the FLI object pointed to by pointer.

If index is supplied, dereference assumes that pointer points to one element in an array of object, and returns the element with index index in the array.

If type is supplied, then dereference assumes that pointer points to an object of that type, overriding the type in pointer itself.
copy-foreign-object is only used when the type of pointer (or type if supplied) is an aggregate type, because objects of these types cannot be converted to a Lisp value. If copy-foreign-object is t, dereference makes a copy of the aggregate object pointed to by pointer and returns the copy. If copy-foreign-object is nil, dereference returns the aggregate object directly. If copy-foreign-object is :error (the default) then dereference signals an error.

The value of an object at pointer can be changed using the setf form of dereference. See the examples section for an example of this.

An error is signaled if pointer is a null pointer. You can use null-pointer-p to detect null pointers.

Compatibility note

64-bit integer types such as (:long :long), :int64 and :uint64 are now supported for type in dereference in 32-bit LispWorks. In 32-bit LispWorks 6.1 and earlier versions, these types could only be used by define-foreign-function.

Examples

In the following example a LONG type is defined and an instance, pointed to by point, with a specified initial value of 10 is created with memory allocated using allocate-foreign-object. The dereference function is then used to get the value that point points to.

```
(fli:define-c-typedef LONG :long)

(setq point (fli:allocate-foreign-object :type 'LONG :initial-element 10))

(fli:dereference point)
```

Finally, the value of the object of type LONG is changed to 20 using the setf form of dereference.

```
(setf (fli:dereference point) 20)
```

In the next example, a boolean FLI type is defined, but is accessed as a char.

```
(fli:define-c-typedef BOOL (:boolean :int))

(setq point2 (fli:allocate-foreign-object :type 'BOOL))

(fli:dereference point2 :type :char)
```

See also

allocate-foreign-object
free-foreign-object
foreign-slot-value
null-pointer-p

2 FLI Types
3.3 Pointer dereferencing and coercing
5.2.5 Calling a C function that takes an array of strings
**disconnect-module**

**Function**

**Summary**
Disconnects the DLL associated with a registered module.

**Package**

fli

**Signature**

\texttt{disconnect-module \ name \ \&key \ verbose \ remove \ \Rightarrow \ result}

**Arguments**

- **name**\(\downarrow\): A symbol or string.
- **verbose**\(\downarrow\): \texttt{nil, t} or an output stream.
- **remove**\(\downarrow\): A boolean.

**Values**

- **result**: \texttt{nil, t} or \texttt{:removed}.

**Description**

The function \texttt{disconnect-module} disconnects the DLL associated with a registered module specified by \texttt{name} and registered with \texttt{register-module}.

When disconnecting, if \texttt{verbose} is a stream, then \texttt{disconnect-module} will send disconnection information to that stream. If \texttt{verbose} is \texttt{t}, this is interpreted as standard output. The default value of \texttt{verbose} is \texttt{nil}.

If \texttt{remove} is \texttt{nil} then after disconnection the module will be in the same state as it was when first registered by \texttt{register-module}. That is, lookups for foreign symbols can still automatically reconnect the DLL. If \texttt{remove} is non-nil then \texttt{name} is removed from the list of registered modules. Any foreign symbols which refer to the module will then be reset as unresolved symbols. The default value of \texttt{remove} is \texttt{nil}.

\texttt{disconnect-module} returns \texttt{t} if it actually disconnected the module, which means it unloaded the foreign module, but has not removed the module. It returns \texttt{:removed} when it also removed the module. Note that when \texttt{disconnect-module} is supplied with a non-nil \texttt{remove}, it may still decline to remove the module if there are symbols which are explicitly associated with the module (for example by by passing \texttt{:module to define-foreign-function}). \texttt{nil} is returned if it fails to find the module, or it was not already connected before the call and was not removed by the call.

**See also**

\texttt{register-module}
enum-symbol-value
enum-value-symbol
enum-values
enum-symbols
enum-symbol-value-pairs

Functions

Summary
Finds values and symbols in a FLI enumerator type.

Package
fli

Signatures
enum-symbol-value enum-type symbol => value
enum-value-symbol enum-type value => symbol
enum-values enum-type => values
enum-symbols enum-type => symbols
enum-symbol-value-pairs enum-type => pairs

Arguments

define-c-enum

enum-type
A FLI enumerator type defined by define-c-enum.
symbol
A symbol.
value
An integer.

Values

value
An integer or nil.
symbol
A symbol or nil.
values
A list.
symbols
A list.
pairs
A list of conses.

Description
The function enum-symbol-value returns the value value of symbol symbol in the FLI enumerator type enum-type, or nil if enum-type does not contain symbol.

The function enum-value-symbol returns the symbol symbol in the FLI enumerator type enum-type at value value, or nil if value is out of range for enum-type.

The functions enum-values, enum-symbols and enum-symbol-value-pairs respectively return a list of the values,
symbols and pairs for `enum-type`, where a pair is a cons of symbol and value.

`enum-type` must be defined by `define-c-enum`.

Examples

```lisp
(fli:define-c-enum colors red green blue)
=> (:ENUM COLORS)

(fli:enum-symbol-value 'COLORS 'red)
=> 0

(fli:enum-value-symbol 'COLORS 0)
=> RED

(fli:define-c-enum half_year (jan 1) feb mar apr may jun)
=> (:ENUM HALF_YEAR)

(fli:enum-symbol-value 'HALF_YEAR 'feb)
=> 2

(fli:enum-value-symbol 'HALF_YEAR 2)
=> FEB

(fli:enum-symbol-value-pairs 'HALF_YEAR)
((JAN . 1) (FEB . 2) (MAR . 3) (APR . 4) (MAY . 5) (JUN . 6))
```

See also

`define-c-enum`

---

**fill-foreign-object**

*Function*

Summary

Fills a foreign object, given a pointer to it.

Package

`fli`

Signature

`fill-foreign-object pointer &key nelems byte => pointer`

Arguments

- `pointer` A foreign pointer.
- `nelems` A non-negative integer. The default is 1.
An integer. The default is 0.

Values

pointer The foreign pointer.

Description

The function fill-foreign-object fills the pointer pointer with the value byte. If nelems is greater than 1, an array of objects starting at pointer is filled.

Examples

```
(fli:with-dynamic-foreign-objects ()
  (let ((pp (fli:allocate-dynamic-foreign-object
    :type :char
    :initial-element 66
    :nelems 6)))
    (fli:fill-foreign-object pp :nelems 3 :byte 65)
    (loop for i below 6 collect
      (fli:dereference pp :type :char :index i))))
=>
(\A \A \A \B \B \B)
```

See also

replace-foreign-object

---

**foreign-aref**

Accessor

Summary

Accesses and returns the value at a specified point in an array.

Package

fli

Signature

foreign-aref array &rest subscripts => value

(setf (foreign-aref array &rest subscripts) value) value => value

Arguments

array A FLI array or a pointer to a FLI array.

subscripts A list of valid array indices for array.

value An element of array.

Values

value An element of array.
Description

The accessor **foreign-aref** accesses an element in *array* specified by *subscripts* and returns its value if the element is an immediate type. If it is an aggregate type, such as a :struct, :union, or :c-array, an error is signaled. The function **foreign-array-pointer** should be used to get access to such embedded aggregate data.

The value of an element in an array can be changed using the **setf** form of **foreign-aref**. See the examples section for an example of this.

Examples

In the first example, a 3 by 3 integer array is created, and the **setf** form of **foreign-aref** is used to set all the elements to 42.

```lisp
(setq array1 (fli:allocate-foreign-object
             :type '(:c-array :int 3 3)))

(dotimes (x 3)
   (dotimes (y 3)
      (setf (fli:foreign-aref array1 x y) 42)))
```

Next, **foreign-aref** is used to dereference the value at position 2 2 in *array1*. Remember that the count for the indices of an array start at 0.

```lisp
(fli:foreign-aref array1 2 2)
```

In the following example, an array of arrays of integers is created. When an element is dereferenced, a copy of an array of integers is returned.

```lisp
(setq array2 (fli:allocate-foreign-object
             :type '(:c-array (:c-array :int 3) 3)))

(fli:foreign-array-pointer array2 2)
```

The array returned can be bound to the variable *array3*, and accessed using **foreign-aref** again. This time an integer is returned.

```lisp
(setq array3 *)

(fli:foreign-aref array3 1)
```

See also

- 2 FLI Types
  - foreign-array-dimensions
  - foreign-array-element-type
  - foreign-array-pointer
  - foreign-typed-aref
foreign-array-dimensions

Function

Summary
Returns a list containing the dimensions of an array.

Package
fli

Signature
foreign-array-dimensions array-or-type => dimensions

Arguments
array-or-type
A FLI array, a pointer to a FLI array or the name of a FLI array type.

Values
dimensions
A list containing the dimensions of array-or-type.

Description
The function foreign-array-dimensions returns a list containing the dimensions of array-or-type.

Examples
In the following example an instance of a 3 by 4 array is created, and these dimensions are returned using the foreign-array-dimensions function.

```lisp
(setq array1 (fli:allocate-foreign-object
              :type '(c-array :int 3 4)))
(fli:foreign-array-dimensions array1)
```

See also
foreign-aref
foreign-array-element-type
foreign-array-pointer

foreign-array-element-type

Function

Summary
Returns the type of the elements of an array.
foreign-array-element-type

Arguments
array-or-type

A FLI array, a pointer to a FLI array or the name of a FLI array type.

Values
type

The type of the elements of array-or-type.

Description
The function foreign-array-element-type returns the type of the elements of array-or-type.

Examples
In the following example a 3 by 4 array with integer elements is defined, and the foreign-array-element-type function is used to confirm that the elements of the array are indeed integers.

```
(setq array1 (fli:allocate-foreign-object :type '(:c-array :int 3 4)))

(fli:foreign-array-element-type array1)
```

See also
foreign-aref
foreign-array-dimensions
foreign-array-pointer

---

foreign-array-pointer

Summary
Returns a pointer to a specified element in an array.

Package
fli

Signature

foreign-array-pointer array &rest subscripts => pointer

Arguments
array\downarrow A FLI array or a pointer to a FLI array.

subscripts\downarrow A list of valid array indices for array.

Values

pointer A pointer to the element at position subscripts in array.

Description

The function foreign-array-pointer returns a pointer to an element in array specified by subscripts. You can then use dereference or foreign-slot-value to access the value.

Examples

In this example a 3 by 2 array of integers is created, and a pointer to the element at position 2 0 is returned using foreign-array-pointer.

```
(setq array1 (fli:allocate-foreign-object :type `(:c-array :int 3 2)))
```

```
(setq array-ptr (fli:foreign-array-pointer array1 2 0))
```

The setf form of dereference can now be used to set the value pointed to by array-ptr.

```
(setf (fli:dereference array-ptr) 42)
```

See also

foreign-aref
foreign-array-dimensions
foreign-array-element-type

---

### foreign-block-copy

**Function**

Summary

Makes a copy of a foreign block, in LispWorks for Macintosh.

Package

fli

Signature

foreign-block-copy foreign-block => new-foreign-block

Arguments

foreign-block\downarrow A foreign block pointer.
7 Function, Macro and Variable Reference

Values

new-foreign-block

A foreign block pointer.

Description

The function foreign-block-copy makes and returns a copy of the foreign block foreign-block. It corresponds to the C function _Block_copy.

foreign-block can be any foreign block.

The result of the copy is another foreign block with an indefinite scope, which has the same attributes as foreign-block. In other words, invoking the copy invokes the same function.

The new foreign block cannot be garbage collected. It should be freed when you are finished with it by foreign-block-release.

foreign-block-copy is not expected to be commonly useful. You need it when you get passed a block and you want to use it outside the scope of the call in which it was passed, unless it is documented that the block is global.

Notes

1. If you use new-foreign-block with a function that is documented to release the block, you must not call foreign-block-release on it. However, we do not expect this situation to happen, because a proper interface will only free blocks that it allocates.

2. foreign-block-copy is implemented in LispWorks for Macintosh only.

See also

foreign-block-release

5.7 Block objects in C (foreign blocks)

---

foreign-block-release

Function

Summary

Releases a foreign block, like _Block_release, in LispWorks for Macintosh.

Package

fli

Signature

foreign-block-release foreign-block

Arguments

foreign-block

A foreign block pointer.

Description

The function foreign-block-release releases a foreign block. It corresponds to the C function _Block_release.
foreign-block must be the result of foreign-block-copy. In particular, it is an error to call foreign-block-release on the result of allocate-foreign-block.

Notes

1. In principle, you can also use foreign-block-release on foreign blocks that you received from foreign code, if the interface says that you need to release them. However, we do not expect this to happen, because proper interface will always free blocks that it allocates or copies.

2. After the call to foreign-block-release, foreign-block is of type released-foreign-block-pointer.

3. foreign-block-release has no useful return value.

4. foreign-block-release is implemented in LispWorks for Macintosh only.

5. To free a foreign block that was allocated by Lisp, use free-foreign-block.

See also

foreign-block-copy
free-foreign-block
released-foreign-block-pointer
5.7 Block objects in C (foreign blocks)

foreign-function-pointer

Summary

Returns a FLI pointer with its address set to the address of a foreign symbol.

Package

fli

Signature

foreign-function-pointer symbol-name => pointer

Arguments

symbol-name

A string or a symbol.

Values

pointer

A FLI pointer.

Description

The function foreign-function-pointer returns a FLI pointer with its address set to the address of a foreign symbol, which can be either a symbol defined in a foreign library or a foreign callable.

symbol-name needs to be a name of a foreign symbol specifying a foreign function, either a string naming a symbol defined in a foreign library, or a symbol naming a foreign callable (defined by define-foreign-callable).

foreign-function-pointer returns a FLI pointer with its address set to the address of the symbol. If the symbol is not
defined yet an error is signaled.

The pointer that is returned is associated with the symbol and is returned in further calls to `foreign-function-pointer` with the same argument. The pointer must not be modified by functions like `incf-pointer`.

When a saved image is restarted all the pointers that have been returned by `foreign-function-pointer` are updated to reflect the current address of their symbol (which may be different in the new invocation).

Notes

1. The pointer is not updated if the module containing the symbol is disconnected and registered again.

2. Only the pointer itself is updated, but not any copies of it. `foreign-function-pointer` is very similar to calling `make-pointer` with `symbol-name`, with the following differences:
   - The result of `foreign-function-pointer` is updated on image restart.
   - `foreign-function-pointer` returns the same pointer for the same `symbol-name` each time, so modifying the pointer will break it.
   - `foreign-function-pointer` allocates only in the first call for each symbol. In contrast, `make-pointer` allocates a pointer in each call.
   - `foreign-function-pointer` keeps the pointer, so if you want to use it only once, `make-pointer` is better.

3. `foreign-function-pointer` is especially useful for creating pointers for passing the address of foreign callables to foreign code in situations where the same address is used repeatedly.

See also

- `define-foreign-callable`
- `make-pointer`
- 3.1.1 Creating pointers

---

### `foreign-slot-names` Function

**Summary**

Returns a list of the slot names in a foreign structure.

**Package**

`fli`

**Signature**

```
foreign-slot-names object => slot-names
```

**Arguments**

- `object` A foreign object or a pointer to a foreign object.

**Values**

- `slot-names` A list containing the slot names of `object`. 

---

105
The function `foreign-slot-names` returns a list containing the slot names of `object`, whose foreign type was defined by `define-c-struct`. If `object` is not a structure, an error is signaled.

Examples

In the following example a structure with three slots is defined, an instance of the structure is made, and `foreign-slot-names` is used to return a list of the slot names.

```lisp
(fli:define-c-struct POS
 (x :int)
 (y :int)
 (z :int))

(setq my-pos (fli:allocate-foreign-object :type 'POS))

(fli:foreign-slot-names my-pos)
```

See also

2.2.3 Structures and unions

`define-c-struct`  
`foreign-slot-value`

---

### foreign-slot-offset

**Function**

#### Summary

Returns the offset of a slot in a FLI object.

#### Package

`fli`

#### Signature

`foreign-slot-offset object-or-type slot-name => offset`

#### Arguments

- `object-or-type`  
  A foreign object, a pointer to a foreign object, or a foreign structure or union type.

- `slot-name`  
  A symbol or a list of symbols identifying the slot to be accessed, as described for `foreign-slot-value`.

#### Values

- `offset`  
  The offset, in bytes, of the slot `slot-name` in the FLI object `object`.

#### Description

The function `foreign-slot-offset` returns the offset, in bytes, of the slot `slot-name` in `object-or-type`. The offset is the
number of bytes from the beginning of the object to the start of the slot. For example, the offset of the first slot in any FLI object is 0.

Examples
The following example defines a structure, creates an instance of the structure pointed to by \texttt{dir}, and then finds the offset of the third slot in the object.

\begin{verbatim}
(fli:define-c-struct compass
 (east :int)
 (west (:c-array :char 20))
 (north :int)
 (south :int))

(fli:foreign-slot-offset 'compass 'north)

(setq dir (fli:allocate-foreign-object :type 'compass))

(fli:foreign-slot-offset dir 'north)
\end{verbatim}

See also

\texttt{foreign-slot-value}
\texttt{foreign-slot-pointer}
\texttt{size-of}

\section*{foreign-slot-pointer} \textit{Function}

Summary
Returns a pointer to a specified slot of an object.

Package
\texttt{fli}

Signature
\texttt{foreign-slot-pointer object slot-name &key type object-type => pointer}

Arguments
\begin{itemize}
  \item \texttt{object} A foreign object, or a pointer to a foreign object.
  \item \texttt{slot-name} A symbol or a list of symbols identifying the slot to be accessed, as described for \texttt{foreign-slot-value}.
  \item \texttt{type} A foreign type.
  \item \texttt{object-type} The FLI structure type that contains \texttt{slot-name}.
\end{itemize}

Values
\begin{itemize}
  \item \texttt{pointer} A pointer to the slot identified by \texttt{slot-name}.
The function `foreign-slot-pointer` returns a foreign pointer to the slot `slot-name` in `object`.

If `type` is supplied, then `foreign-slot-pointer` assumes that the slot contains an object of that type, overriding the type in the structure definition.

If `object-type` is supplied then `foreign-slot-pointer` assumes that `object` is of the that type and the compiler might be able to optimize the access to the slot. If `object-type` is not supplied, then the object type is determined dynamically from `object`.

**Examples**

In the following example a structure type called `compass` is defined. An instance of the structure is allocated using `allocate-foreign-object`, pointed to by `point1`. Then `foreign-slot-pointer` is used to get a pointer, called `point2`, to the second slot of the foreign object.

```lisp
(fli:define-c-struct compass
  (west :int)
  (east :int))

(setq point1 (fli:allocate-foreign-object :type 'compass))

(setq point2 (fli:foreign-slot-pointer point1 'east
                                      :type :int))
```

The :type keyword can be used to return the value stored in the slot as a different type, providing the type is compatible. In the next example, `point3` is set to be a pointer to the same address as `point2`, but it expects the value stored there to be a boolean.

```lisp
(setq point3 (fli:foreign-slot-pointer point1 'east
                                          :type '(:boolean :int)))
```

Using `dereference` the value can be set as an integer using `point2` and read as a boolean using `point3`.

```lisp
(setf (fli:dereference point2) 0)

(fli:dereference point3)

(setf (fli:dereference point2) 1)

(fli:dereference point3)
```

See also

2.2.3 Structures and unions

- defc-pointer
- incf-pointer
- make-pointer
- foreign-slot-value
- foreign-slot-offset
**foreign-slot-type**  

*Function*

**Summary**

Returns the type of a specified slot of a foreign object.

**Package**

fli

**Signature**

`foreign-slot-type object-or-type slot-name => type`

**Arguments**

- `object-or-type`  
  A foreign object, a pointer to a foreign object, or a foreign structure or union type.

- `slot-name`  
  A symbol or a list of symbols identifying the slot whose type is to be returned. The value is interpreted as described for `foreign-slot-value`.

**Values**

- `type`  
  The type of `slot-name`.

**Description**

The function `foreign-slot-type` returns the type of the slot `slot-name` in `object-or-type`.

**Examples**

In the following example two new types, `east` and `west` are defined. Then a new structure, `compass`, is defined, with two slots. An instance of the structure is created, and `foreign-slot-type` is used to get the type of the first slot of the structure.

```lisp
(fli:define-c-typedef east (:boolean :int))
(fli:define-c-typedef west :long)
(fli:define-c-struct compass
  (x east)
  (y west))
(fli:foreign-slot-type 'compass 'x)
(setq dir (fli:allocate-foreign-object :type 'compass))
(fli:foreign-slot-type dir 'x)
```

**See also**

- 2.2.3 Structures and unions
- `foreign-slot-names`
- `foreign-slot-value`
**foreign-slot-value**

*Accessor*

**Summary**

Returns the value of a slot in a foreign object.

**Package**

fli

**Signature**

```
foreign-slot-value object slot-name &key type object-type copy-foreign-object => value
```

```
(setf foreign-slot-value object slot-name &key type object-type copy-foreign-object) value => value
```

**Arguments**

- `object`\↓ Either an instance of or a pointer to a FLI structure.
- `slot-name`\↓ A symbol or a list of symbols identifying the slot to be accessed.
- `type`\↓ A foreign type.
- `object-type`\↓ The FLI structure type that contains `slot-name`. If this is passed, the compiler might be able to optimize the access to the slot. If this is omitted, the object type is determined dynamically from `object`.
- `copy-foreign-object`\↓ One of `t`, `nil` or `:error`.
- `value` The value of the slot `slot-name` in the FLI object `object` is returned.

**Values**

- `value` The value of the slot `slot-name` in the FLI object `object` is returned.

**Description**

The accessor `foreign-slot-value` accesses and returns the value of a slot in a specified object. An error is signaled if the slot is an aggregate type and `copy-foreign-object` is not supplied as `t` or `nil`. Use `foreign-slot-pointer` to access such aggregate slots.

If `slot-name` is a symbol then it names the slot of `object` to be accessed. If `slot-name` is a list of symbols, then these symbols name slots in nested structures starting with the outermost structure `object`, as in the `inner/middle/outer` example below.

If `type` is supplied, then `foreign-slot-value` assumes that the slot contains an object of that type, overriding the type in the structure definition.

`copy-foreign-object` is only used when the type of the slot (or `type` if supplied) is an aggregate type, because objects of these types cannot be converted to a Lisp value. If `copy-foreign-object` is `t`, `foreign-slot-value` makes a copy of the aggregate object in the slot and returns the copy. If `copy-foreign-object` is `nil`, `foreign-slot-value` returns the aggregate object directly. If `copy-foreign-object` is `:error` (the default) then `foreign-slot-value` signals an error.

If `object-type` is supplied then `foreign-slot-value` assumes that `object` is of the that type and the compiler might be able to optimize the access to the slot. If `object-type` is not supplied, then the object type is determined dynamically from `object`.
The **setf** form of **foreign-slot-value** can be used to set the value of a slot in a structure, as shown in the example below.

**Compatibility note**

64-bit integer types such as (**:long :long**), **:int64** and **:uint64** are now supported for **type** in **foreign-slot-value** in 32-bit LispWorks. In 32-bit LispWorks 6.1 and earlier versions, these types could only be used by **define-foreign-function**.

**Examples**

In the following example a foreign structure is defined, an instance of the structure is made with **my-pos** pointing to the instance, and **foreign-slot-value** is used to set the **y** slot of the object to 10.

```lisp
(fli:define-c-struct POS
  (x :int)
  (y :int)
  (z :int))

(setq my-pos (fli:allocate-foreign-object :type 'POS))

(setf (fli:foreign-slot-value my-pos 'y) 10)
```

The next forms both return the value of the **y** slot at **my-pos**, which is 10.

```lisp
(fli:foreign-slot-value my-pos 'y)

(fli:foreign-slot-value my-pos 'y :object-type 'pos)
```


This example accesses a slot in nested structures:

```lisp
(fli:define-c-struct inner
  (v1 :int)
  (v2 :int))

(fli:define-c-struct middle
  (i1 (:struct inner))
  (i2 (:struct inner)))

(fli:define-c-struct outer
  (m1 (:struct middle))
  (m2 (:struct middle)))

(fli:with-dynamic-foreign-objects
  ((obj (:struct outer)))
  (setf (fli:foreign-slot-value obj '(m1 i2 v1)) 99))
```

See also

**2.2.3 Structures and unions**

**foreign-slot-pointer**

**foreign-slot-offset**

**dereference**


**with-foreign-slots**

---

---

**foreign-typed-aref**

*Accessor*

**Summary**

Accesses a foreign array and can be compiled to efficient code.

**Package**

fli

**Signature**

```lisp
foreign-typed-aref type array index => value
```

```lisp
setf (foreign-typed-aref type array index) value => value
```

**Arguments**

- `type` A type specifier.
- `array` A foreign pointer.
- `index` A non-negative `integer`.
- `value` An element of `array`.

**Values**

- `value` An element of `array`.

**Description**

The accessor `foreign-typed-aref` accesses a foreign array and is compiled to efficient code when compiled at safety 0. It corresponds to `sys:typed-aref` which accesses Lisp vectors.

`type` must evaluate to a supported element type for foreign arrays. In 32-bit LispWorks these types are `double-float`, `single-float`, `(unsigned-byte 32)`, `(signed-byte 32)`, `(unsigned-byte 16)`, `(signed-byte 16)`, `(unsigned-byte 8)`, `(signed-byte 8)` and `sys:int32`. In 64-bit LispWorks `type` can also be `(unsigned-byte 64)`, `(signed-byte 64)` and `sys:int64`.

`array` is a foreign pointer to a FLI array. Memory can be allocated with:

```lisp
(fli:allocate-foreign-object
 :type :double
 :nelems
 (ceiling byte-size
  (fli:size-of :double)))
```

to get sufficient alignment for any call to `foreign-typed-aref`.

In the case the memory is allocated by the operating system the best approach is to reference it from Lisp by a pointer type, to avoid making a `:c-array` foreign type dynamically.

`index` should be a valid byte index for `array`. If `index` is declared to be of type `fixnum` then the compiler will optimize it.
slightly better. Some parts of the FLI (for example, \texttt{allocate-foreign-object}) assume \texttt{fixnum} sizes so it is best to use fixnums only.

Notes
Efficient access to a Lisp vector object is also available. See \texttt{sys:typed-aref} in the \textit{LispWorks\textregistered User Guide and Reference Manual}.

See also

2 FLI Types
\texttt{foreign-aref}

\texttt{foreign-type-equal-p}

\textit{Function}

Summary
Determines whether two foreign types are the same underlying foreign type.

Package
\texttt{fli}

Signature
\texttt{foreign-type-equal-p} \texttt{type1} \texttt{type2} \Rightarrow \texttt{result}

Arguments
\texttt{type1} \texttt{\downarrow} A foreign type.
\texttt{type2} \texttt{\downarrow} A foreign type.

Values
\texttt{result} A boolean.

Description
The function \texttt{foreign-type-equal-p} returns true if \texttt{type1} and \texttt{type2} are the same underlying foreign type, and false otherwise.

Examples

\begin{verbatim}
(fli:define-foreign-type aa () '(:signed :byte)) => aa

(fli:define-foreign-type bb () '(:signed :char)) => bb

(fli:foreign-type-equal-p 'aa 'bb) => t
\end{verbatim}
See also

2 FLI Types
define-foreign-type

---

### foreign-type-error

**Condition Class**

**Summary**

The class of errors signaled when an object does not match a foreign type.

**Package**

fli

**Superclasses**

type-error

**Description**

The condition class `foreign-type-error` is used for errors signaled when an object does not match a foreign type.

---

### free-foreign-block

**Function**

**Summary**

Frees a foreign block that was allocated by Lisp, in LispWorks for Macintosh.

**Package**

fli

**Signature**

`free-foreign-block foreign-block`

**Arguments**

- `foreign-block` A Lisp-allocated `foreign-block-pointer`.

**Description**

The function `free-foreign-block` frees a foreign block that was allocated by Lisp. `foreign-block` must be a result of a call to `allocate-foreign-block`. It is an error to call `free-foreign-block` on the
result of `foreign-block-copy` or on a foreign block coming from foreign code.

Note that the function that was passed to `allocate-foreign-block` may still be invoked after `free-foreign-block`, because the block may have been copied. See the discussion in 5.7.3 Scope of invocation.

It is an error to call `free-foreign-block` more than once on the same `foreign-block`.

`free-foreign-block` has no useful return value.

Notes

1. To free a foreign block that was allocated by foreign code, use `foreign-block-release`.
2. `free-foreign-block` is implemented in LispWorks for Macintosh only.

See also

`allocate-foreign-block`
`with-foreign-block`

5.7 Block objects in C (foreign blocks)

---

### `free-foreign-object`  

#### Summary

Deallocates the space in memory pointed to by a pointer.

#### Package

`fli`

#### Signatures

- `free-foreign-object pointer => null-pointer`
- `free pointer => null-pointer`

#### Arguments

- `pointer` A pointer to the object to de-allocate.

#### Values

- `null-pointer` A pointer with address zero.

#### Description

The `free-foreign-object` function deallocates the space in memory pointed to by `pointer`, which frees the memory for other uses. The address of `pointer` is the start of a block of memory previously allocated by `allocate-foreign-object`.

If `pointer` is a null pointer then `free-foreign-object` takes no action.

The function `free` is a synonym for `free-foreign-object`. 
Examples
In the following example a boolean type is defined and an instance is created with memory allocated using `allocate-foreign-object`. The function `free-foreign-object` is then used to free up the memory used by the boolean.

```
(fli:define-c-typedef BOOL (:boolean :int))

(setq point (fli:allocate-foreign-object :type 'BOOL))

(fli:free-foreign-object point)
```

See also
`allocate-foreign-object`

1.4 An example of dynamic memory allocation
3.1.3 Allocation of FLI memory

---

**get-embedded-module**

### Function

**Summary**

Gets a foreign module from a file and sets up an embedded dynamic module.

**Package**

`fli`

**Signature**

```
get-embedded-module name filename
```

**Arguments**

- `name`
  - A symbol.
- `filename`
  - A pathname specifier for a file containing a dynamic foreign module.

**Description**

The function `get-embedded-module` gets the foreign module in `filename` and sets up an embedded dynamic module named `name`.

**Notes**

1. `get-embedded-module` is called at load time and has no effect except to set up the embedded module. To actually use the code in the module, you need to call `install-embedded-module` at run time.
3. The module should not have dependencies on other non-standard modules, otherwise `install-embedded-module` may fail to install it.
4. To incorporate an embedded module into a fasl file (that is, to load it at compile time) you need to use both
   get-embedded-module-data (at compile time) and setup-embedded-module (at load time), instead of
   get-embedded-module.

5. get-embedded-module does not return a useful value.

See also

install-embedded-module
get-embedded-module-data
setup-embedded-module

5.6 Incorporating a foreign module into a LispWorks image

get-embedded-module-data

Function

Summary

Returns a foreign module as a Lisp object suitable for use at run time, possibly via a fasl file.

Package

fli

Signature

get-embedded-module-data filename => data

Arguments

filename

A pathname specifier for a file containing a dynamic foreign module.

Values

data

A Lisp object containing the data of the foreign module.

Description

The function get-embedded-module-data returns the foreign module in filename as a Lisp object suitable as argument to
setup-embedded-module, but also externalizable, that is the compiler can put it in a fasl file.

Notes

1. get-embedded-module-data is useful when you need to incorporate a foreign dynamic module in a fasl file, which is
   itself useful when the fasl is loaded on the run time computer. In the usual situation when the fasl is loaded on the same
   computer where it is compiled, get-embedded-module is more convenient, and replaces both
   get-embedded-module-data and setup-embedded-module.

2. To incorporate the module in a fasl file, get-embedded-module-data must be called at compile time, which is
   typically done either by doing it at read time with #. or using a macro. The result is then used as argument to
   setup-embedded-module at load time. Examples of both approaches are shown below.

3. To actually use the code in the module, install-embedded-module must be called at run time with the name of the
   module (my-embedded-module-name in the examples below).
4. The module should not have dependencies on other non-standard modules, otherwise `install-embedded-module` may fail to install it.

Examples

Calling `get-embedded-module-data` at read time with `#.`: 

```lisp
(setup-embedded-module 'my-embedded-module-name
  #.(get-embedded-module-data
      (my-locate-the-foreign-module)))
```

Calling `get-embedded-module-data` via a macro. Note that there is no backquote or quote, so the code is executed by the compiler:

```lisp
(defmacro my-get-embedded-module-data ()
  (let ((pathname (my-locate-the-foreign-module)))
    (get-embedded-module-data pathname))

(setup-embedded-module 'my-embedded-module-name
  (my-get-embedded-module-data)
```

See also

- `install-embedded-module`
- `get-embedded-module-data`
- `setup-embedded-module`

5.6 Incorporating a foreign module into a LispWorks image

---

### incf-pointer

**Function**

**Summary**

Increases the address held by a pointer.

**Package**

fli

**Signature**

`incf-pointer pointer &optional delta => pointer`

**Arguments**

- `pointer`: A FLI pointer.
- `delta`: An integer. The default value is 1.

**Values**

- `pointer`: The pointer passed.
Description

The function `incf-pointer` increases the address held by `pointer`. If `delta` is not given the address is increased by the size of the type pointed to by `pointer`. The address can be increased by a multiple of the size of the type by specifying a `delta`. If the size of the type is 0 then an error is signalled.

The function `incf-pointer` is often used to move a pointer through an array of values.

Examples

In the following example an array with 10 entries is defined. A copy of the pointer to the array is made, and is incremented and decremented.

```lisp
(setq array-obj
    (fli:allocate-foreign-object :type :int
      :nelems 10
      :initial-contents '(0 1 2 3 4 5 6 7 8 9)))

(setq point1 (fli:copy-pointer array-obj))

(dotimes (x 9)
    (print (fli:dereference point1))
    (fli:incf-pointer point1))

(dotimes (x 9)
    (fli:decf-pointer point1)
    (print (fli:dereference point1)))
```

See also

`decf-pointer`

3.4 An example of dynamic pointer allocation

### install-embedded-module

**Function**

Summary

Installs an embedded dynamic module.

Package

`fli`

Signature

```
install-embedded-module name &key delay-delete
```

Arguments

- `name`\[\[A symbol.\]
- `delay-delete`\[\[A boolean.\]

119
Description

The function `install-embedded-module` installs the embedded dynamic module name.

`name` must be a name of an embedded dynamic module that was set up either by `get-embedded-module` or `setup-embedded-module`.

`install-embedded-module` installs the module, which means making its code available to be used in Lisp, as if `register-module` was called with the original module.

The module is written to a temporary file that is deleted by LispWorks.

Note: You should consult LispWorks Support before using `delay-delete`.

`delay-delete` controls the time of deletion of the temporary file that is created by `install-embedded-module`. It defaults to the value of `*install-embedded-module-delay-delete*`, which defaults to `nil`. If `delay-delete` is `nil`, the temporary file is deleted during the call to `install-embedded-module`. If `delay-delete` is non-nil, the file is deleted only when LispWorks exists. On Windows it always behave as if `delay-delete` is non-nil.

Deleting the file immediately is better in most cases, because it means that the file is not left in the filesystem if LispWorks does not exit cleanly (for example if POSIX `kill` is used). However, some debugging code may try to find the temporary file, in which case you can delay the deletion.

Notes

1. `install-embedded-module` must be called at run time, normally during the initialization of the application.

2. The effect of `install-embedded-module` does not persist after `save-image` or `deliver`.

3. `install-embedded-module` can be called repeatedly with the same name. The subsequent calls in the same invocation of the application do not have any effect.

4. `install-embedded-module` does not return a useful value.

See also

`get-embedded-module`
`get-embedded-module-data`
`setup-embedded-module`
`*install-embedded-module-delay-delete*`

5.6 Incorporating a foreign module into a LispWorks image

*`install-embedded-module-delay-delete*` Variable

Summary

Default for the keyword `delay-delete` in `install-embedded-module`.

Package

fli

Initial Value

nil
Description

The variable *install-embedded-module-delay-delete* is used as the default value for the keyword delay-delete in install-embedded-module. See install-embedded-module for more details.

See also

install-embedded-module

*locale-external-formats*  

Summary

Provides a mapping from locale names to encodings.

Package

fli

Initial Value

Not specified.

Description

The variable *locale-external-formats* contains the mapping from locale names to external formats that set-locale uses to set the correct defaults for FLI. The value is an alist with elements of the form:

   (locale multi-byte-ef wide-character-ef)

The locale names are given as strings. If the first character of the string is #\*, then that entry matches any locale having the rest of the string as a suffix. If the last character of the string is #\*, then that entry matches any locale having the rest of the string as a prefix. Either external format may be given as nil, in which case the corresponding foreign type cannot be used without specifying an external format.

Notes

*locale-external-formats* is used only on non-Windows platforms. On Windows, the external formats are based on the Windows Code Page.

See also

:ef-mb-string
:ef-wc-string
set-locale
**make-integer-from-bytes**

Summary
Converts foreign bytes back to a Lisp integer.

Package
fli

Signature

make-integer-from-bytes pointer length => integer

Arguments

- **pointer**
  - A foreign pointer.
- **length**
  - An integer.

Values

- **integer**
  - An integer.

Description

The function `make-integer-from-bytes` converts `length` bytes starting at `pointer` into the Lisp integer `integer`. The bytes and `length` must have been generated by `with-integer-bytes` or `convert-integer-to-dynamic-foreign-object`.

See also

5.3 Lisp integers
- `with-integer-bytes`
- `convert-integer-to-dynamic-foreign-object`

**make-pointer**

Summary
Creates a pointer to a specified address.

Package
fli

Signature

make-pointer &key address type pointer-type symbol-name functionp module encoding => pointer
Arguments

address
The address pointed to by the pointer to be created.

type
The type of the object pointed to by the pointer to be created.

pointer-type
The type of the pointer to be made.

symbol-name
A string or a symbol.

functionp
A boolean.

module
A symbol or string naming a module, or nil.

encoding
One of :source, :object, :lisp or :dbcs.

Values

pointer
A pointer to address.

Description
The function make-pointer creates a pointer of a specified type pointing to a given address address, or optionally to a function or foreign callable.

symbol-name is either a string containing the name of a foreign symbol defined in a DLL, or a string or symbol naming a foreign callable defined by define-foreign-callable.

Either address or symbol-name must be supplied, otherwise make-pointer signals an error.

Note that in many cases, especially when :symbol-name is used with a symbol defined by define-foreign-callable, foreign-function-pointer would be better than using make-pointer with :symbol-name.

If type is supplied, then it is used as the FLI type that pointer points to. Alternatively, if pointer-type is supplied, then it must be a FLI pointer type and it is used as the pointer type of pointer. An error is signalled if both type and pointer-type are supplied.

If type or pointer-type are not supplied, then functionp can be used. If functionp is t, then pointer is a pointer to type :function. This is the default value. If functionp is nil, then pointer is a pointer to type :void.

encoding controls how symbol-name is processed. The values are interpreted like the encode argument of define-foreign-callable. The default value of encoding is :source if symbol-name is a string and :lisp if symbol-name is a symbol.

In the case of a pointer to a foreign callable or foreign function, module can be supplied to ensure that the pointer points to the function in the correct DLL if there are other DLLs containing functions with the same name. module is processed as by define-foreign-function.

Examples
In the following example a module is defined, and the variable setpoint is set equal to a pointer to a function in the module.

(fli:register-module :user-dll :real-name "user32")

(setq setpoint
  (fli:make-pointer :symbol-name "SetCursorPos"
                   :module :user-dll))
module-unresolved-symbols

Summary
Returns foreign symbol names that cannot be resolved.

Package
fli

Signature
module-unresolved-symbols &key module => list

Arguments
module⇓ nil, :all, or a string. The default is :all.

Values
list⇓ A list of strings.

Description
The function module-unresolved-symbols returns a list of foreign symbol names, each of which cannot be resolved in the currently known modules.

If module is nil, then list includes only those names not associated with a module.

If module is :all, then list includes the unresolved names in all modules and those not associated with a module.

If module is a string, then it names a module and list contains only the unresolved symbols associated with that module.

See also
5.5.1.1 Testing whether a function is defined
register-module
**null-pointer**  
*Variable*

Summary
A null pointer.

Package
fli

Initial Value
The result of calling `(make-pointer :address 0 :type :void)`.

Description
The variable `*null-pointer*` contains a `(:pointer :void)` with address 0.
This provides a simple way to pass a null pointer when needed.

Examples

```
(fli:pointer-address fli:*null-pointer*)
=> 0

(fli:null-pointer-p fli:*null-pointer*)
=> T
```

See also

`pointer-address`  
`null-pointer-p`  
`:pointer`

---

**null-pointer-p**  
*Function*

Summary
Tests a pointer to see if it is a null pointer.

Package
fli

Signature

```
null-pointer-p pointer => result
```
Arguments

pointer ⊒ A FLI pointer.

Values

result ⊒ A boolean.

Description

The function `null-pointer-p` is used to determine if a pointer is a null pointer. A null pointer is a pointer pointing to address 0.

If `pointer` is a null pointer (that is, a pointer pointing to address 0) then `result` is true, otherwise `null-pointer-p` returns false.

Examples

In the following example a pointer to an `:int` is defined, and tested with `null-pointer-p`. The pointer is then freed, becoming a null pointer, and is once again tested using `null-pointer-p`.

```lisp
(setq point (fli:allocate-foreign-object :type :int))

(fli:null-pointer-p point)

(fli:free-foreign-object point)

(fli:null-pointer-p point)
```

See also

- 3.2 Pointer testing functions
- 5.5.1.1 Testing whether a function is defined
- *null-pointer*
- `pointer-address`
- `pointer-eq`

`pointer-address` Function

Summary

Returns the address of a pointer.

Package

`fli`

Signature

`pointer-address` `pointer => address`
Arguments

\textit{pointer} \downarrow 
A FLI pointer.

Values

\textit{address} 
A non-negative integer.

Description

The function \texttt{pointer-address} returns the address of \textit{pointer} as an integer.

Examples

In the following example a pointer is defined, and its address is returned using \texttt{pointer-address}.

\begin{verbatim}
(setq point (fli:allocate-foreign-object :type :int))
(fli:pointer-address point)
\end{verbatim}

See also

3.2 Pointer testing functions
\texttt{null-pointer-p}
\texttt{pointer-eq}

\texttt{pointer-element-size} \quad \textit{Function}

Summary

Returns the size in bytes of a foreign object or a foreign type.

Package

\texttt{fli}

Signature

\texttt{pointer-element-size \hspace{1em} pointer-or-type \Rightarrow size}

Arguments

\texttt{pointer-or-type} \downarrow 
A FLI pointer to a foreign object or the name of a FLI pointer type.

Values

\texttt{size} \downarrow 
A non-negative integer.

Description

The function \texttt{pointer-element-size} returns the size, in bytes, of the object or type specified.

If \texttt{pointer-or-type} is an FLI pointer, \texttt{size} is the size, in bytes, of the object pointed to by \texttt{pointer-or-type}.
If `pointer-or-type` is the name of a FLI pointer type, `size` is the size, in bytes, of the elements of that type.

Examples

In the following example a pointer to an integer is created. Then the size in bytes of the integer is returned using `pointer-element-size`.

```
(setq point (fli:allocate-foreign-object :type :int))

(fli:pointer-element-size point)
```

See also

3.2 Pointer testing functions

`pointer-element-type`

`size-of`

### `pointer-element-type`

**Function**

**Summary**

Returns the type of the foreign object pointed to by a FLI pointer.

**Package**

fli

**Signature**

`pointer-element-type pointer-or-type => type`

**Arguments**

- `pointer-or-type`
  
  A FLI pointer to a foreign object or the name of a FLI pointer type.

**Values**

- `type`
  
  The name of a FLI pointer type.

**Description**

The function `pointer-element-type` returns the type of the foreign object specified, or the element type of the foreign type specified.

If `pointer-or-type` is a FLI pointer, `type` is the type of the foreign object pointed to by `pointer-or-type`.

If `pointer-or-type` is the name of a FLI pointer type, `type` is the type of the elements of that FLI pointer type.

**Examples**

In the following example a pointer to an integer is defined, and `pointer-element-type` is used to confirm that the pointer points to an integer.
In the next example a new type, \texttt{happy}, is defined. The pointer \texttt{point} is set to point to an instance of \texttt{happy}, and \texttt{pointer-element-type} is used to find the type of the object pointed to by \texttt{point}.

\begin{verbatim}(setq point (fli:allocate-foreign-object :type :int))
(fli:pointer-element-type point)
\end{verbatim}

See also

3.2 Pointer testing functions
\begin{itemize}
  \item \texttt{foreign-slot-type}
  \item \texttt{pointer-element-size}
  \item \texttt{pointer-element-type-p}
\end{itemize}

\section*{pointer-element-type-p} \textit{Function}

\textbf{Summary}
Tests whether a FLI pointer matches a given element type.

\textbf{Package}
fli

\textbf{Signature}
\begin{verbatim}pointer-element-type-p \hspace{1em} pointer type \Rightarrow \hspace{1em} result\end{verbatim}

\textbf{Arguments}

- \textit{pointer} \hspace{1em} A FLI pointer to a foreign object.
- \textit{type} \hspace{1em} A foreign type.

\textbf{Values}

- \textit{result} \hspace{1em} A boolean.

\textbf{Description}
The function \texttt{pointer-element-type-p} returns true if the element type of the foreign object pointed to by \texttt{pointer} has the same underlying type as \texttt{type}.

\textbf{Examples}
See also

3.2 Pointer testing functions
pointer-element-type

## pointer-eq

### Summary

Test whether two pointers point to the same memory address.

### Package

fli

### Signature

```
pointer-eq pointer1 pointer2 => boolean
```

### Arguments

- **pointer1**: A FLI pointer.
- **pointer2**: A FLI pointer.

### Values

- **boolean**: A boolean.

### Description

The function `pointer-eq` tests whether `pointer1` points to the same address as `pointer2` and returns `t` if they do, and `nil` if they do not.

### Examples

In the following example a pointer, `point1`, is defined, and `point2` is set equal to it. Both are then tested to see if they are equal to each other using `pointer-eq`. Then `point2` is defined to point to a different object, and the two pointers are tested for equality again.

```
(setq point1 (fli:allocate-foreign-object :type :int))

(setq point2 point1)

(fli:pointer-eq point1 point2)
```
pointerp

Summary
Tests whether an object is a pointer or not.

Package
fli

Signature
pointerp pointer => result

Arguments
pointer\[ An object that may be a FLI pointer.

Values
result\[ A boolean.

Description
The function pointerp tests whether the argument pointer is a pointer.
result is t if pointer is a pointer, otherwise nil is returned.

Examples
In the following example a pointer, point, is defined, and an object which is not a pointer is defined. Both are tested using pointerp.

(setq point (fli:allocate-foreign-object :type :int))

(setq not-point 7)

(fli:pointerp point)

(fli:pointerp not-point)


See also

3.2 Pointer testing functions

null-pointer-p
pointer-address
pointer-eq

pointer-pointer-type

Function

Summary

Returns the pointer type of a FLI pointer.

Package

fli

Signature

pointer-pointer-type pointer => pointer-type

Arguments

pointer

A FLI pointer.

Values

pointer-type

The pointer type of pointer.

Description

The function pointer-pointer-type returns the pointer type of the foreign pointer pointer.

Examples

(setq point (fli:allocate-foreign-object :type :int)) => #<Pointer to type :INT = #x007F3DF0>

(fli:pointer-pointer-type point) => (:POINTER :INT)

(fli:free-foreign-object point) => #<Pointer to type :INT = #x00000000>

See also

3.3 Pointer dereferencing and coercing

make-pointer
print-collected-template-info

Summary
Prints the FLI Template information in the image.

Package
fli

Signature
print-collected-template-info &key output-stream => nil

Arguments
output-stream↓ An output stream designator. The default is nil, meaning standard output.

Description
The FLI converters require pieces of compiled code known as FLI templates, and sometimes your delivered application will need extra templates not included in LispWorks as shipped.

The function print-collected-template-info prints the information about FLI templates that has been collected. These must be compiled and loaded into your application. The output is printed to output-stream.

See the Delivery User Guide for further details.

See also
start-collecting-template-info

print-foreign-modules

Summary
Prints the foreign modules loaded into the image by register-module.

Package
fli

Signature
print-foreign-modules &optional stream verbose => nil

Arguments
stream↓ An output stream.
verbose↓ A generalized boolean.
Description

The function `print-foreign-modules` prints a list of the foreign modules loaded via `register-module`, to the stream `stream`.

The default value of `stream` is the value of `*standard-output*`. `verbose` is ignored.

See also

`register-module`

---

**register-module**

*Function*

**Summary**

Informs LispWorks of the presence of a dynamic library.

**Package**

`fli`

**Signature**

```
register-module name &key connection-style lifetime real-name file-name dlopen-flags => name
```

**Arguments**

- `name` A symbol or string specifying the Lisp name the module will be registered under.
- `connection-style` A keyword determining when the connection to the dynamic library is made. One of `:automatic`, `:manual` or `:immediate`. The default value is `:automatic`.
- `lifetime` A keyword specifying the lifetime of the connection. One of `:indefinite` or `:session`. The default value is `:indefinite`.
- `real-name` Deprecated. Use `file-name` instead.
- `file-name` A pathname designator or `nil`.
- `dlopen-flags` Controls use of `dlopen` on non-Windows platforms. One of `t` (the default), `nil`, `:local-now`, `:global-now`, `:global-lazy`, `:local-lazy`, or a fixnum.

**Values**

- `name` A symbol or string specifying the Lisp name the module will be registered under.

**Description**

The function `register-module` explicitly informs LispWorks of the presence of a DLL or shared object file, referred to here as a dynamic library. Functions such as `make-pointer` and `define-foreign-function` have a `module` keyword which can be used to specify which module the function refers to.

The main use of modules is to overcome ambiguities that can arise when two different dynamic libraries have functions with the same name.
If an application is delivered after calling `register-module`, then the application attempts to reload the module on startup but does not report any errors. Therefore it is strongly recommended that you call `register-module` during initialization of your application, rather than at compile time or build time. Loading the module at run time allows you to:

- Report loading errors to the user or application error log.
- Compute the path (as described below), if needed.
- Make the loading conditional, if needed.

You should compute and supply the appropriate full path if possible.

`name` is used for explicit look up from the `module` keyword of functions such as `define-foreign-function`. If `name` is a symbol, then `file-name` should also be supplied to provide a filename. `file-name` defaults to the deprecated argument `real-name`, which defaults to `nil`. If `file-name` is `nil` then `name` must be a string that specifies the actual name of the dynamic library to connect to.

The naming convention for the module `name` can contain the full pathname for the dynamic library. For example, a pathname such as:

```lisp
#P"C:/MYPRODUCT/LIBS/MYLIBRARY.DLL"
```

is specified as:

```
"C:\MYPRODUCT\LIBS\MYLIBRARY.DLL"
```

On Windows, if the module is declared without an extension, "*.DLL" is automatically appended to the name. To declare a name without an extension it must end with the period character ("."). On other platforms, you should provide the extension, since there is more than one library format. Typical would be `.so` on Linux, x86/x64 Solaris or FreeBSD and `.dylib` on macOS.

If a full pathname is not specified for the module, then it is searched for.

On Windows the following directories (in the given order) are searched:

1. The directory of the executable.
2. The Windows system directory (as specified by `GetSystemDirectory`).
3. The 16-bit system directory.
5. The current directory. This step can be made to happen earlier, though this is considered less safe as described in the Microsoft documentation.
6. Directories specified by the `PATH` environment variable.

The simplest approach is usually to place the DLL in the same directory as the LispWorks executable or application. However if you really need different directories then be sure to call `register-module` at run time with the appropriate pathname.

On Linux, FreeBSD and Solaris the search is conducted in this order:

1. Directories on the user's `LD_LIBRARY_PATH` environment variable.
2. The list of libraries known to the operating system (for example, in `/etc/ld.so.cache` on Linux).
3. `/usr/lib`, followed by `/lib`. 

135
On macOS, the search is conducted in this order:

1. Directories on the user's `LD_LIBRARY_PATH` environment variable.
2. Directories on the user's `DYLD_LIBRARY_PATH` environment variable.
3. `~/lib`
4. `/usr/local/lib`
5. `/usr/lib`

If `connection-style` is `:automatic` then the system automatically connects to a dynamic library when it needs to resolve currently undefined foreign symbols.

If `connection-style` is `:manual` then the system only connects to the dynamic library if the symbol to resolve is explicitly marked as coming from this module via the `:module` keyword of functions such as `define-foreign-function`.

If `connection-style` is `:immediate` then the connection to the dynamic library is made immediately. This checks that the library can actually be loaded before its symbols are actually needed: an error is signalled if loading fails.

If `lifetime` is `:session` then the module is disconnected when Lisp starts up.

You should load only libraries of the correct architecture into LispWorks. You will need to obtain a 32-bit dynamic library for use with 32-bit LispWorks and similarly you need a 64-bit dynamic library for use with 64-bit LispWorks. (If you build the dynamic library, pass `--m32` or `--m64` as appropriate to `cc`.) You can conditionalize the argument to `register-module` as in the example below.

**Note:** On Linux, you may see a spurious "No such file or directory" error message when loading a dynamic library of the wrong architecture. The spurious message might be localized.

**Note:** static libraries are not supported. For example, on Linux evaluating this form:

```lisp
(fli:register-module "libc.a"
 :real-name "/usr/lib/libc.a"
 :connection-style :immediate)
```

would result in an error like this:

```
Could not register handle for external module "libc"
/usr/lib/libc.a : invalid ELF header
```

The problem is that `libc.a` is a static library. Instead, do:

```lisp
(fli:register-module "libc.so"
 :real-name "libc.so.6"
 :connection-style :immediate)
```

Note that `:real-name` is given a relative path in this case, because `libc` is a standard library on Linux and it is best to let the operating system locate it.

`dlopen-flags` has an effect only on non-Windows platforms. It controls the value that is passed to `dlopen` as second argument when the module is connected.

The keyword values of `dlopen-flags` correspond to combinations of `RTLD_*` constants (see `/usr/include/dlfcn.h`). The values `t` and `nil` mean the same as `:local-lazy`.

A fixnum value means pass this value `dlopen-flags` to `dlopen` without checking. It is the responsibility of the caller to get it right in this case.
7 Function, Macro and Variable Reference

Compatibility note:

In LispWorks 7.1 and earlier versions, \texttt{dlopen-flags} defaults to \texttt{nil} on macOS, which caused it to use the older interfaces instead of \texttt{dlopen}. Since LispWorks 8.0, this is no longer supported.

Notes

1. It is strongly recommended that you call \texttt{register-module} during initialization of your application, rather than at compile time or build time.

2. When developing with foreign code in LispWorks, the utilities provided in the Editor are useful - see \texttt{9.4.2 Compiling and Loading Foreign Code with the Editor}.

Examples

In the following example on Windows, the \texttt{user32} DLL is registered, and then a foreign function called \texttt{set-cursor-pos} is defined to explicitly reference the \texttt{SetCursorPos} function in the \texttt{user32} DLL.

\begin{verbatim}
(fli:register-module :user-dll :real-name "user32")

(fli:define-foreign-function (set-cursor-pos
                             "SetCursorPos")
   ((x :long)
    (y :long))
   :module :user-dll)
\end{verbatim}

This example on Linux loads the shared library even though its symbols are not yet needed. An error is signalled if loading fails:

\begin{verbatim}
(fli:register-module "libX11.so"
   :connection-style :immediate)
\end{verbatim}

This example loads a module from the same directory as the Lisp executable, by executing this code at run time:

\begin{verbatim}
(fli:register-module
   modulename
   :file-name
   (merge-pathnames "modulefilename.dylib"
                    (lisp-image-name)))
\end{verbatim}

In this last example a program which runs in both 32-bit LispWorks and 64-bit LispWorks loads the correct library for each architecture:

\begin{verbatim}
(fli:register-module #+:lispworks-32bit "mylib32"
                    #+:lispworks-64bit "mylib64")
\end{verbatim}

See also

5.6 Incorporating a foreign module into a LispWorks image
\texttt{connected-module-pathname}
\texttt{define-foreign-function}
\texttt{make-pointer}
\texttt{module-unresolved-symbols}
\texttt{print-foreign-modules}
replace-foreign-array

Summary
Copies the contents of one foreign or Lisp array into another.

Package
fli

Signature
replace-foreign-array to from &key start1 start2 end1 end2 allow-sign-mismatch => to

Arguments
to↓ A foreign array, foreign pointer or a Lisp array.
from↓ A foreign array, foreign pointer or a Lisp array.
start1↓, start2↓, end1↓, end2↓ Integers.
allow-sign-mismatch↓ A boolean, default value nil.

Values
to A foreign array, foreign pointer or a Lisp array.

Description
The function replace-foreign-array copies the contents of the array specified by from into another array specified by to. The arrays element types must have the same size and both be either signed or unsigned. When allow-sign-mismatch is nil (the default), the array element types must also match for sign, that is they must be either both signed or both unsigned. When allow-sign-mismatch is non-nil, the array element types do not need to match.

The argument to is destructively modified by copying successive elements into it from from. Elements of the subsequence of from bounded by start2 and end2 are copied into the subsequence of to bounded by start1 and end1. If these subsequences are not of the same length, then the shorter length determines how many elements are copied; the extra elements near the end of the longer subsequence are not involved in the operation.

Each of to and from can be one of the following:

A Lisp array The start and end are handled in the same way as Common Lisp sequence functions. The array must be "raw", which means either an integer array of length 8, 16, 32 or 64 bits, or an array of one of cl:base-char, lw:bmp-char, cl:single-float and cl:double-float. For matching with the other argument, the latter are considered as "unsigned", with size 8, 16, 32 and 64 bits respectively. Note that arrays with element type cl:character are not allowed.

A foreign array The start and end are handled in the same way as Common Lisp sequence functions.

A pointer to a foreign array The start and end are handled in the same way as Common Lisp sequence functions.
A pointer to any other foreign object

In this case, the pointer is assumed to point to an array of such objects. Start and end are used as indices into that array, but without any bounds checking.

Compatibility note:

In LispWorks 6.1 and earlier versions you can use an array of `lw:simple-char`, that is `lw:text-string`, because `lw:simple-char` was limited to the range that is now `lw: bmp-char` and had width of 16.

In LispWorks 7.0 and later versions `lw:simple-char` is a synonym for `cl:character`, and thus arrays of `lw:simple-char` (that is, `lw:text-string`) cannot be used in `replace-foreign-array`.

Examples

This example demonstrates copying from a foreign pointer to a Lisp array.

An initial array filled with 42:

```lisp
(setq lisp-array
  (make-array 10
   :element-type '(unsigned-byte 8)
   :initial-element 42))
```

A foreign pointer to 10 consecutive unsigned chars:

```lisp
(setq foreign-array
  (fli:allocate-foreign-object
   :type '(:unsigned :char)
   :nelems 10
   :initial-contents '(1 2 3 4 5 6 7 8 9 10)))
```

Copy some of the unsigned char into the Lisp array. Without :start2 and :end2, only the first unsigned char would be copied:

```lisp
(fli:replace-foreign-array
 lisp-array foreign-array
 :start1 3
 :start2 5 :end2 8)
=>
#(42 42 42 6 7 8 42 42 42 42)
```

This example demonstrates copying from a foreign array to a Lisp array.

A pointer to a foreign array of 10 unsigned chars:

```lisp
(setq foreign-array
  (fli:allocate-foreign-object
   :type
    '(:c-array (:unsigned :char) 10)))

(dotimes (i 10)
  (setf (fli:foreign-aref foreign-array i) (1+ i)))
```

Copy part of the foreign array into the Lisp array:

```lisp
(fli:replace-foreign-array
 lisp-array foreign-array :start1 7)
```
See also

allocate-foreign-object  
copy-pointer  
make-pointer  
replace-foreign-object

replace-foreign-object

Summary
Copies the contents of one foreign object into another.

Package
fli

Signature
replace-foreign-object to from &key nelems => pointer

Arguments

to↓ A foreign object or a pointer to a foreign object.
from↓ A foreign object or a pointer to a foreign object.
nelems↓ An integer.

Values
pointer A pointer to the object specified by from.

Description
The function replace-foreign-object copies the contents of the foreign object specified by from into another foreign object specified by to. Block copying on an array of elements can also be performed by supplying the number of elements to copy using nelems.

Examples
In the following object two sets of ten integers are defined. The object from-obj contains the integers from 0 to 9. The object to-obj contains random values. The replace-foreign-object function is then used to copy the contents of from-obj into to-obj.

```
(setf from-obj
  (fli:allocate-foreign-object
    :type :int
    :nelems 10
    :initial-contents
    '(0 1 2 3 4 5 6 7 8 9)))
```
7 Function, Macro and Variable Reference

(setf to-obj
     (fli:allocate-foreign-object
      :type :int
      :nelems 10 ))

(fli:replace-foreign-object to-obj from-obj :nelems 10)

See also

5.2.4 Modifying a string in a C function
allocate-foreign-object
fill-foreign-object
copy-pointer
make-pointer
replace-foreign-array

set-locale

Summary
Sets the C locale and the default for FLI string conversions.

Package
fli

Signature
set-locale &optional locale => c-locale

Arguments
locale A string, the locale name.

Values
c-locale A string naming the C locale, or nil.

Description
The function set-locale can be called to set the C locale; if you set the locale in any other way, then Lisp might not do the right thing when passing strings and characters to C. It calls setlocale to tell the C library to switch and then calls set-locale-encodings to tell the FLI what conversions to do when passing strings and characters to C. locale should be a locale name; if not passed, it defaults according to the OS conventions.

If set-locale fails to set the C locale, a warning is signaled, nil is returned and the FLI conversion defaults are not modified.

Examples
On a Windows system:

(fli:set-locale "English_UK")
On a Linux system:

```lisp
(fli:set-locale) =>
"en_US"
```

See also

- `convert-from-foreign-string`
- `convert-to-foreign-string`
- `:ef-mb-string`
- `:ef-wc-string`
- `*locale-external-formats*`
- `set-locale-encodings`
- `with-foreign-string`

### set-locale-encodings

**Function**

**Summary**

Tells the FLI what default conversions to use when passing strings and characters to C.

**Package**

`fli`

**Signature**

```lisp
(set-locale-encodings mb wc => mb)
```

**Arguments**

- `mb` An external format specification.
- `wc` An external format specification, or `nil`.

**Values**

- `mb` An external format specification.

**Description**

The function `set-locale-encodings` changes the default encodings used by those FLI functions and types which convert strings and characters and accept an `:external-format` argument.

`mb` is set as the external format for multi-byte encodings.

If `wc` is non-nil, then it is set as the external format for wide-character encodings, such as `:ef-wc-string`.

`set-locale` calls `set-locale-encodings` after successfully setting the C locale.
setup-embedded-module

Summary
Sets up an embedded dynamic module.

Package
fli

Signature

setup-embedded-module name data

Arguments

name⇓
A symbol.

data⇓
A Lisp object containing the data of the foreign module.

Description
The function setup-embedded-module sets up an embedded dynamic module named name using data.
data must be a result of a call to get-embedded-module-data.

Notes

1. setup-embedded-module is called at load time and has no effect except to set up the embedded module. To actually use the code in the module, you need to call install-embedded-module at run time.


3. See get-embedded-module-data for more discussion and examples.

4. setup-embedded-module does not return a useful value.

See also
install-embedded-module
get-embedded-module-data
get-embedded-module

5.6 Incorporating a foreign module into a LispWorks image
size-of

Summary
Returns the size in bytes of a foreign type.

Package
fli

Signature
size-of type-name => size

Arguments
type-name
A foreign type whose size is to be determined.

Values
size
The size of the foreign type type-name in bytes.

Description
The function size-of returns the size in bytes of the foreign language type named by type-name.

Examples
This example returns the size of the C integer type (usually 4 bytes on supported platforms):

(fli:size-of :int)

This example returns the size of a C array of 10 integers:

(fli:size-of '(:c-array :int 10))

The function size-of can also be used to determine the size of a structure:

(fli:define-c-struct POS
  (x :int)
  (y :int)
  (z :int))

(fli:size-of 'POS)

See also
2 FLI Types
allocate-foreign-object
free-foreign-object
start-collections-template-info

Summary

Nullifies the FLI Template information in the image.

Package

fli

Signature

start-collections-template-info => nil

Description

The FLI converters require pieces of compiled code known as FLI templates, and sometimes your delivered application will need extra templates not included in LispWorks as shipped.

The function start-collections-template-info throws away any information about FLI templates that has been collected. Call it when you want to start collecting to create a definitive set of template information.

See the Delivery User Guide for further details.

See also

print-collected-template-info

*use-sse2-for-ext-vector-type*

Summary

32-bit x86 specific: control whether to pass/receive vector type arguments/results using SSE2.

Package

fli

Initial Value

t on macOS, nil on other platforms.

Description

On 32-bit x86 platforms, the variable *use-sse2-for-ext-vector-type* controls whether the code that is generated by foreign interface definitions that pass or receive vector type arguments or results (see 2.2.4 Vector types) uses SSE2 to pass or receive these arguments or results.

SSE2 is a feature of the x86 CPU, which was introduced by Intel in 2001, and is supported by all new x86 CPUs. However, the C compiler can still pass arguments without using SSE2 for backwards compatibility. The Lisp definitions must pass/receive arguments in the same way that as the C compiler that compiled the foreign code they call/are called from.
On macOS, code always uses SSE2, so *use-sse2-for-ext-vector-type* is set to t initially and you should not change it. On other platforms (Linux, FreeBSD, Solaris) the situation is less clear.

*use-sse2-for-ext-vector-type* affects the code at macro expansion time, so if you use compile-file and later load the compiled file, the value of *use-sse2-for-ext-vector-type* at the time of compile-file determine what the code does. When evaluating the definition, the value at the time of evaluating the definition determines what the code does.

Notes

On FreeBSD, the default C compiler is Clang, which currently (Dec 2016 in FreeBSD 10.3) does not use SSE2 by default, and therefore matches what LispWorks does by default.

On other platforms, or using other compilers or newer versions of Clang, if you use vector types then you will need to check what the C compiler does. If you have any doubt, contact LispWorks support.

See also

2.2.4 Vector types

---

### valid-foreign-type-p

**Summary**

Checks if the argument is a valid foreign type.

**Package**

fli

**Signature**

\[
\text{valid-foreign-type-p } \text{type} \Rightarrow \text{boolean}
\]

**Arguments**

\[
\begin{align*}
\text{type} & \downarrow \\
\text{A Lisp object.}
\end{align*}
\]

**Values**

\[
\begin{align*}
\text{boolean} & \\
\text{A boolean.}
\end{align*}
\]

**Description**

The function valid-foreign-type-p returns true if type is a valid foreign type and returns false otherwise.

An object is a valid foreign type if it matches any of the types which are described in chapter 2 FLI Types.

See also

2 FLI Types
**with-coerced-pointer**

*Macro*

**Summary**
Executes forms with a variable bound to a dynamic-extent copy of an FLI pointer, possibly with a different type.

**Package**
fli

**Signature**

\[
\text{with-coerced-pointer} \ (\text{coerced-pointer} \ &\text{key} \ \text{type} \ \text{pointer-type}) \ \text{pointer} \ &\text{body} \ \text{body} \Rightarrow \text{last}
\]

**Arguments**

- **coerced-pointer**
  A variable bound to a copy of \text{pointer}.

- **type**
  The type of the object pointed to by the temporary pointer. This keyword can be used to access the data at the pointer as a different type.

- **pointer-type**
  The pointer type of the temporary pointer.

- **pointer**
  A FLI pointer of which a copy is made.

- **body**
  A list of forms to be executed across the scope of the temporary pointer binding.

**Values**

- **last**
  The value of the last form in \text{body}.

**Description**

The macro **with-coerced-pointer** makes a temporary copy of a pointer, and executes a list of forms which may use the copy across the scope of the macro. Once the macro has terminated the memory allocated to the copy of the pointer is automatically freed.

The macro **with-coerced-pointer** evaluates \text{body} with **coerced-pointer** bound to a dynamic-extent copy of the FLI pointer \text{pointer}.

**coerced-pointer** points to the same foreign object as \text{pointer}.

If **type** is specified, then it must be a FLI type specifying the type that **coerced-pointer** points to. Alternatively, if **pointer-type** is specified, then it must be a FLI pointer type specifying the pointer type of **coerced-pointer**. If neither **type** nor **pointer-type** are specified then the type is the same as **pointer**.

You can use **with-coerced-pointer** in a similar way to casting a pointer type in C. You can also use it make a temporary FLI pointer that can be changed using **incf-pointer** or **defc-pointer**, without affecting \text{pointer}.

Note that **coerced-pointer** has dynamic-extent, so you should not use it after returning from \text{body}.

**Examples**

In the following example an array of ten integers is defined, pointed to by **array-obj**. The macro **with-coerced-pointer** is used to return the values stored in the array, without altering **array-obj**, or permanently tying
up memory for a second pointer.

```
(setf array-obj
    (fli:allocate-foreign-object :type :int
      :nelems 10
      :initial-contents
      '(0 1 2 3 4 5 6 7 8 9)))

(fli:with-coerced-pointer (temp) array-obj
  (dotimes (x 10)
    (print (fli:dereference temp))
    (fli:incf-pointer temp)))
```

See also

3.4 An example of dynamic pointer allocation
allocate-dynamic-foreign-object
free-foreign-object
with-dynamic-foreign-objects

---

**with-dynamic-foreign-objects**  

*Macro*

**Summary**

Does the equivalent of `dynamic-extent` for foreign objects.

**Package**

fli

**Signature**

```
with-dynamic-foreign-objects bindings &body body => last
```

```
bindings ::= (binding*)
binding ::= (var foreign-type &key initial-element initial-contents fill nelems size-slot)
```

**Arguments**

- `body`  
  Forms to be executed with `bindings` in effect.

- `var`  
  A symbol to be bound to a pointer to a foreign object.

- `foreign-type`  
  A foreign type descriptor.

- `initial-element`  
  The initial value of the newly allocated objects.

- `initial-contents`  
  A list of values to initialize the contents of the newly allocated objects.

- `fill`  
  An integer between 0 to 255.

- `nelems`  
  An integer specifying how many copies of the object should be allocated. The default value is 1.

- `size-slot`  
  A symbol naming a slot in the object.
7 Function, Macro and Variable Reference

Values

last

The value of the last form in body.

Description

The macro `with-dynamic-foreign-objects` binds variables according to the list `bindings`, and then evaluates the forms in `body` as an implicit `progn`. Each element of `bindings` is a list which caused `var` to be bound to a pointer to a locally allocated instance of `foreign-type`.

`initial-element`, `initial-contents`, `fill`, `nelems` and `size-slot` initialize the allocated instance as if by `allocate-foreign-object`.

The lifetime of the bound foreign objects, and hence the allocation of the memory they take up, is within the scope of the `with-dynamic-foreign-objects` function.

Any object created with `allocate-dynamic-foreign-object` within `body` will automatically be deallocated once the scope of the `with-dynamic-foreign-objects` function has been left.

Compatibility note

There is an alternative syntax for `binding` with an optional `initial-element` which is the only way to supply an initial element in LispWorks 5.0 and previous versions. Like this:

```lisp
binding ::= (var foreign-type &optional initial-element)
```

This alternative syntax is deprecated in favor of the keyword syntax for `binding` defined above, which is supported in LispWorks 5.1 and later.

Examples

This example shows the use of `with-dynamic-foreign-objects` with an implicitly created pointer.

Windows version:

```c
typedef struct {
    int one;
    float two;
} foo;

__declspec(dllexport) void __cdecl init_alloc(foo *ptr, int a, float b)
{
    ptr->one = a;
    ptr->two = b;
};
```

Non-Windows version:

```c
typedef struct {
    int one;
    float two;
} foo;

void init_alloc(foo * ptr, int a, float b)
{
    ptr->one = a;
    ptr->two = b;
};
```
Here are the FLI definitions interfacing to the above C code:

```
(fli:define-c-typedef (foo (:foreign-name "foo"))
  (:struct (one :int) (two :float)))

(fli:define-foreign-function (init-alloc "init_alloc")
  ((ptr (:pointer foo))
   (a :int)
   (b :float))
  :result-type :void
  :calling-convention :cdecl)
```

Try this test function which uses `with-dynamic-foreign-objects` to create a transient `foo` object and pointer:

```
(defun test-alloc (int-value float-value &optional (level 0))
  (fli:with-dynamic-foreign-objects ((object foo))
    (init-alloc object int-value float-value)
    (format t "Level - ~D~%   object : ~S~%   slot one : ~S~%   slot two : ~S~"
            level object
            (fli:foreign-slot-value object 'one)
            (fli:foreign-slot-value object 'two))
    (when (> int-value 0)
      (test-alloc (1- int-value)
                  (1- float-value) (1+ level)))
    (when (> float-value 0)
      (test-alloc (1- int-value)
                  (1- float-value) (1+ level)))))
```

(test-alloc 1 2.0)

=>
Level - 0
  object : #<Pointer to type FOO = #x007E6338>
  slot one : 1
  slot two : 2.0

Level - 1
  object : #<Pointer to type FOO = #x007E6340>
  slot one : 0
  slot two : 1.0

Level - 2
  object : #<Pointer to type FOO = #x007E6348>
  slot one : -1
  slot two : 0.0

Level - 1
  object : #<Pointer to type FOO = #x007E6340>
  slot one : 0
  slot two : 1.0

Level - 2
  object : #<Pointer to type FOO = #x007E6348>
  slot one : -1
  slot two : 0.0

A further example using `with-dynamic-foreign-objects` and a pointer created explicitly by `allocate-dynamic-foreign-object` is given in 1.4 An example of dynamic memory allocation.

See also

5.2.4 Modifying a string in a C function
allocate-dynamic-foreign-object
free-foreign-object
with-dynamic-lisp-array-pointer

Summary
Creates a dynamic-extent foreign pointer which points to the data in a given Lisp array while the forms are executed.

Package
fli

Signature
with-dynamic-lisp-array-pointer (pointer-var lisp-array &key start type) &body body => last

Arguments
pointer-var A variable to be bound to the foreign pointer.
lisp-array A static or pinned Lisp array (a string or a byte/single-float/double-float array).
start An index into the Lisp array.
type A foreign type. The default is :void.
body A list of forms.

Values
last The value of the last form in body.

Description
The macro with-dynamic-lisp-array-pointer enables the data in a Lisp array to be shared directly with foreign code, without making a copy. A dynamic-extent pointer to the array's data can be used within body wherever the :pointer foreign type allows.

with-dynamic-lisp-array-pointer creates a dynamic extent foreign pointer, with element type type, which is initialized to point to the element of lisp-array at index start. The default value of start is 0.

This foreign pointer is bound to pointer-var, the forms of body are executed and the value of the last form is returned.

Pointers created with this macro must be used with care. There are three restrictions:

1. lisp-array must be static or pinned, for example allocated as shown below.

2. The pointer has dynamic extent and lisp-array is guaranteed to be preserved only during the execution of body. If you keep the value of the pointer, you must also preserve lisp-array, that is you must ensure it is not garbage-collected.

3. Lisp strings and arrays are not null-terminated, therefore foreign code must only access the data of lisp-array up to its known length.

Examples
An example of using a static array:
An example of using a pinned array:

(let ((vector
  (make-array 3 :element-type '(unsigned-byte 8)
   :initial-contents '(65 77 23)
   :allocation :pinnable))
  (with-pinned-objects (vector)
    (fli:with-dynamic-lisp-array-pointer
      (ptr vector :start 1 :type '(:unsigned :byte))
      (fli:dereference ptr)))
  =>
  77

See also
:lisp-array
:lisp-simple-ld-array
with-pinned-objects

with-foreign-block  

Macro

Summary
Allocates a foreign block, executes code and frees the block, in LispWorks for Macintosh.

Package
fli

Signature

with-foreign-block (foreign-block-var type function &rest extra-args) &body body => results

Arguments

foreign-block-var  A symbol.

A symbol naming a foreign block type defined using
define-foreign-block-callable-type.

function  A Lisp function.

eextra-args  Arguments for function.

body  Lisp forms.
Values

results The results of body.

Description

The macro with-foreign-block allocates a foreign block using type, function and extra-args in the same way as allocate-foreign-block. It then binds foreign-block-var to the foreign block, execute the code of body and frees the foreign block using free-foreign-block, using unwind-protect.

with-foreign-block is a convenient way to ensure that you do not forget to free the foreign block.

Notes

If the foreign block is copied in the code of body, the copy may be invoked, and hence the function called, after exiting this macro. See the discussion in 5.7.3 Scope of invocation.

with-foreign-block returns the results of body.

with-foreign-block is implemented in LispWorks for Macintosh only.

See also

allocate-foreign-block
free-foreign-block
with-local-foreign-block
5.7 Block objects in C (foreign blocks)

with-foreign-slots Macro

Summary

Allows convenient access to the slots of a foreign structure.

Package

fli

Signature

with-foreign-slots slots-and-options form &body body

slots-and-options ::= (slots &key object-type) | slots
slots ::= (slot-spec*)
slot-spec ::= slot-name | (variable-name slot-name &key copy-foreign-object)

Arguments

form A form evaluating to an instance of (or a pointer to) a FLI structure.
body Forms to be executed.
object-type A FLI structure type.
7 Function, Macro and Variable Reference

**Description**

The macro with-foreign-slots is analogous to the Common Lisp macro with-slots. Within body, each slot-name (or variable-name) evaluates to the result of calling foreign-slot-value on form with that slot. setf can be used to set the foreign slot value.

If the first syntax of slots-and-options is used, then object-type is passed as the value of the :object-type keyword argument in all the generated calls to foreign-slot-value. If the second syntax of slots-and-options is used, no object-type is passed.

Each slot-spec can either be a symbol slot-name naming a slot in the object, which will be also be used in body, or a list of variable-name, a symbol naming a slot, and a plist of options. In this case copy-foreign-object is passed as the value of the :copy-foreign-object keyword argument in the generated call to foreign-slot-value. The default value of copy-foreign-object is :error.

The with-foreign-slots form returns the value of the last form in body.

**Examples**

```lisp
(fli:define-c-struct abc
    (a :int)
    (b :int)
    (c :int))

(= (:STRUCT ABC)

(setf abc (fli:allocate-foreign-object :type 'abc))

(#<Pointer to type (:STRUCT ABC) = #x007F3BE0>

(fli:with-foreign-slots (a b c) abc
    (setf a 6 b 7 c (* a b)))

42

(fli:foreign-slot-value abc 'c)

42
```

**See also**

2.2.3 Structures and unions

foreign-slot-value

---

**with-foreign-string**

Macro

**Summary**

Converts a Lisp string to a foreign string, binds variables to a pointer to the foreign string, the number of elements in the string, and the number of bytes taken up by the string, then executes a list of forms, and finally de-allocates the foreign string and pointer.
7 Function, Macro and Variable Reference

Package

fli

Signature

\texttt{with-foreign-string (pointer element-count byte-count \&key external-format null-terminated-p allow-null) string \\
\&body body => last}

Arguments

\texttt{pointer} \downarrow \hspace{1cm} \text{A symbol bound to a pointer to the foreign string.}

\texttt{element-count} \downarrow \hspace{1cm} \text{A symbol bound to the number of elements in the foreign string.}

\texttt{byte-count} \downarrow \hspace{1cm} \text{A symbol bound to the number of bytes occupied by the foreign string. If the element size of the string is equal to one byte, then byte-count will be the same as element-count.}

\texttt{external-format} \downarrow \hspace{1cm} \text{An external format specification.}

\texttt{null-terminated-p} \downarrow \hspace{1cm} \text{If t, the foreign string is terminated by a null character. The null character is included in the value of element-count.}

\texttt{allow-null} \downarrow \hspace{1cm} \text{A boolean. The default is \texttt{nil}.}

\texttt{string} \downarrow \hspace{1cm} \text{The Lisp string to convert.}

\texttt{body} \downarrow \hspace{1cm} \text{A list of forms to be executed.}

Values

\texttt{last} \hspace{1cm} \text{The value of the last form in body.}

Description

The macro \texttt{with-foreign-string} is used to dynamically convert a Lisp string to a foreign string and execute a list of forms using the foreign string. The macro first converts \texttt{string}, a Lisp string, into a foreign string. The symbol \texttt{pointer} is bound to a pointer to the start of the string, the symbol \texttt{element-count} is set equal to the number of elements in the string, and the symbol \texttt{byte-count} is set equal to the number of bytes the string occupies. Then the list of forms specified by \texttt{body} is executed. Finally, the memory allocated for the foreign string and pointer is de-allocated.

\texttt{external-format} is used to specify the encoding of the foreign string. It defaults to a format appropriate for C string of type \texttt{char}. For Unicode encoded strings, specify \texttt{:unicode}. If you want to pass a string to the Win32 API, known as \texttt{STR} in the Win32 API terminology, specify \texttt{*multibyte-code-page-ef*}, which is a variable holding the external format corresponding to the current Windows multi-byte code page. To change the default, call \texttt{set-locale} or \texttt{set-locale-encodings}. The names of available external formats are listed in section 26.6 External Formats to translate Lisp characters from/to external encodings in the LispWorks® User Guide and Reference Manual.

\texttt{null-terminated-p} specifies whether the foreign string is terminated with a null character. It defaults to \texttt{t}. If the string terminates in a null character, it is included in the value of \texttt{element-count}.

If \texttt{allow-null} is non-nil, then if \texttt{string} is \texttt{nil} a null pointer is passed.

See also

5.2.4 Modifying a string in a C function
5.1 Passing a string to a Windows function
Converts a Lisp integer to foreign bytes while executing a body of code.

**package**

fli

**Signature**

```lisp
(with-integer-bytes (pointer length) integer &body body => last)
```

**Arguments**

- `pointer`\: A variable to be bound to the foreign pointer.
- `length`\: A variable to be bound to the length in bytes.
- `integer`\: An integer.
- `body`\: Forms to be executed.

**Values**

- `last`\: The value of the last form in `body`.

**Description**

The macro `with-integer-bytes` evaluates the forms in `body` with `pointer` bound to a dynamic foreign object containing the bytes of `integer` and `length` bound to the number of bytes in that object. The layout of the bytes is unspecified, but the bytes and the length are sufficient to reconstruct `integer` by calling `make-integer-from-bytes`.

**See also**

- 5.3 Lisp integers
- `convert-integer-to-dynamic-foreign-object`
- `make-integer-from-bytes`

---

Allocates a foreign block, executes code and frees the block, in LispWorks for Macintosh.
with-local-foreign-block (foreign-block-var type function &rest extra-args) &body body => results

Arguments

| foreign-block-var | A symbol. |
| type | A symbol naming a foreign block type defined using define-foreign-block-callable-type. |
| function | A Lisp function. |
| extra-args | Arguments for function |
| body | Lisp forms. |

Values

| results | The results of body. |

Description

The macro with-local-foreign-block allocates a foreign block using type, function and extra-args in the same way as allocate-foreign-block, but with dynamic extent. It then binds foreign-block-var to the foreign block and executes the code of body.

with-local-foreign-block can be used only if the code in body can be guaranteed not to invoke the block or a copy of it either outside the scope of with-local-foreign-block or in another thread. Unless you can be sure of that, you need to use with-foreign-block.

with-local-foreign-block returns the results of body.

with-local-foreign-block can be a little faster than with-foreign-block.

Notes

with-local-foreign-block is implemented in LispWorks for Macintosh only.

See also

allocate-foreign-block
free-foreign-block
with-foreign-block

5.7 Block objects in C (foreign blocks)
8 Type Reference

:boolean

Summary
Converts between a Lisp boolean value and a C representation of a boolean value.

Package

keyword

Syntax
:boolean &optional encapsulates

Arguments

encapsulates
An integral type or :standard.

Description
The FLI type :boolean converts between a Lisp boolean value and a C representation of a boolean value. encapsulates specifies the size of the value from which the boolean value is obtained, which defaults to :int. For example, if a byte is used in C to represent a boolean, the size to map across for the FLI will be one byte, but if an int is used, then the size will be four bytes. If encapsulates is :standard, then the type maps to the _Bool type in the C99 language definition.

A value of 0 in C represents a nil boolean value in Lisp, and a non-zero value in C represents a t boolean value in Lisp.

Examples
In the following three examples, the size of a :boolean, a (:boolean :int) and a (:boolean :byte) are returned.

(fli:size-of :boolean)

(fli:size-of '(:boolean :int))

(fli:size-of '(:boolean :byte))

See also

size-of
2.1.5 Boolean types
8 Type Reference

:byte

Summary
Converts between a Lisp integer with a C signed char.

Package
keyword

Syntax
:byte

Description
The FLI type :byte converts between a Lisp integer type and a C signed char type.

See also
:char
:short

2.1.1 Integral types

:c-array

Summary
Converts between a FLI array and a C array type.

Package
keyword

Syntax
:c-array type &rest dimensions

Arguments

The type of the elements of the array.

dimensions
A sequence of the dimensions of the new array.

Description
The FLI type :c-array converts between FLI arrays and the C array type. In C, pointers are used to access the elements of an array. The implementation of the :c-array type takes this into account, by automatically dereferencing any pointers returned when accessing an array using foreign-aref.

When using the :c-array type in the specification of an argument to define-foreign-function, a pointer to the array
is passed to the foreign function, as specified by the C language. You are allowed to call the foreign function with a FLI pointer pointing to an object of type type instead of a FLI array.

When using the :c-array type in other situations, it acts as an aggregate type like :foreign-array. In particular, :c-array with more than one dimension is an array containing embedded arrays, not an array of pointers.

dimensions is the dimensions of the array.

Notes

1. :c-array uses the C convention that the first index value of an array is 0.

2. Only use the :c-array type when the corresponding C code uses an array with a constant declared size. If you need a dynamically sized array, then use a pointer type, allocate the array using the nelems argument to allocate-foreign-object or with-dynamic-foreign-objects and use dereference to access the elements. The pointer type is more efficient than making :c-array types dynamically with different dimensions because the FLI caches information about every different FLI type descriptor that is used.

Examples

The following code defines a 3 by 3 array of integers:

```lisp
(setq aaa (fli:allocate-foreign-object
:type '(:c-array :int 3 3)))
```

The type of this is equivalent to the C declaration:

```
int aaa[3][3];
```

The next example defines an array of arrays of bytes:

```lisp
(setq bbb (fli:allocate-foreign-object
:type '(:c-array (:c-array :byte 3) 2)))
```

The type of this is equivalent to the C declaration:

```
int bbb[2][3];
```

Note the reversal of the 3 and 2.

See foreign-aref and foreign-array-pointer for more examples on the use of arrays.

See also

foreign-aref
:foreign-array
foreign-array-pointer
2.2.1 Arrays
8 Type Reference

:char

Summary
Converts between a Lisp character type and a C char type.

Package

keyword

Syntax
:char

Description
The FLI type :char converts between a Lisp character and a C char type.

Notes
If you want an integer on the Lisp side, rather than a character, then you should use (:signed :char) or (:unsigned :char).

See also
:byte
:signed
:unsigned
2.1.4 Character types

:const

Summary
Corresponds to the C const type.

Package

keyword

Syntax
:const &optional type

Arguments

type
The type of the constant. The default is :int.
8 Type Reference

Description

The FLI type :const corresponds to the C const type qualifier. The behavior of a :const is exactly the same as the behavior of its type, and it is only included to ease the readability of FLI code and for naming conventions.

Examples

In the following example a constant is allocated and set equal to 3.141.

\[
\begin{align*}
\texttt{(setq pi1 (fli:allocate-foreign-object}} \\
\texttt{\ :type '(:const :float)))} \\
\texttt{(setf (fli:dereference pi1) 3.141))}
\end{align*}
\]

See also

:volatile
2.1 Immediate types

:double

FLI Type Descriptor

Summary

Converts a Lisp double float to a C double.

Package

keyword

Syntax

:double

Description

The FLI type :double converts between a Lisp double float and the C double type.

Compatibility Note

In LispWorks 4.4 and previous on Windows and Linux platforms, all Lisp floats are doubles. In later versions, there are three disjoint Lisp float types in 32-bit LispWorks and two in 64-bit LispWorks, on all platforms.

See also

:float
2.1.2 Floating point types
8 Type Reference

:double-complex

Summary
Converts a Lisp double float complex number to a C double complex.

Package
keyword

Syntax
:double-complex

Description
The FLI type :double-complex converts between a Lisp (complex double-float) and the C double complex type.

See also
:float-complex

2.1.3 Complex number types

:ef-mb-string

Summary
Converts between a Lisp string and a C multi-byte string.

Package
keyword

Syntax
:ef-mb-string &key limit external-format null-terminated-p

Arguments

- limit: The maximum number of bytes of the C multi-byte string.
- external-format: An external format specification.
- null-terminated-p: A boolean controlling the null termination byte.

Description
The FLI type :ef-mb-string converts between a Lisp string and a C multi-byte string. The C string may have a maximum length of limit bytes. limit can be omitted in cases where a new foreign string is being allocated.

external-format is used to specify the encoding of the foreign string. It defaults to an encoding appropriate for C string of type char*. If you want to pass a string to the Windows API, known as STR in the Windows API terminology, specify
8 Type Reference

`win32:*multibyte-code-page-ef*`, which is a variable holding the external format corresponding to the current Windows multi-byte code page. To change the default, call `set-locale` or `set-locale-encodings`.

If null-terminated-p is non-nil, a NULL byte is added to the end of the string.

Notes

If you want to pass a string argument by reference but also allow conversion from Lisp `nil` to a null pointer, specify the `:reference` type :allow-null argument, for example:

```
(:reference-pass :ef-mb-string :allow-null t)
```

See also

`:ef-wc-string`
`:reference`
`set-locale`
`set-locale-encodings`

2.2.2 Strings

---

`:ef-wc-string`  

**FLI Type Descriptor**

**Summary**

Converts between a Lisp string and a C wide-character string.

**Package**

`keyword`

**Syntax**

`:ef-wc-string &key limit external-format null-terminated-p`

**Arguments**

- `limit`  
  The maximum number of characters of the C wide-character string.

- `external-format`  
  An external format specification.

- `null-terminated-p`  
  A boolean controlling the null termination byte.

**Description**

The FLI type `:ef-wc-string` converts between a Lisp string and a C wide-character string. The C string may have a maximum length of `limit` characters. `limit` can be omitted in cases where a new foreign string is being allocated.

`external-format` is used to specify the encoding of the foreign string. It defaults to an encoding appropriate for C string of type `wchar_t*`. For Unicode encoded strings, specify `:unicode`. If you want to pass a string to the Windows API, known as `WSTR` in the Windows API terminology, also specify `:unicode`. To change the default, call `set-locale` or `set-locale-encodings`.

If null-terminated-p is non-nil, a NULL word is added to the end of the string.
See also

:ef-mb-string
set-locale
set-locale-encodings
2.2.2 Strings

:enum
:enumeration

Summary

Converts between a Lisp symbol and a C enum.

Package

keyword

Syntax

:enum &rest enum-constants
:enumeration &rest enum-constants

enum-constants ::= {entry-name | (entry-name entry-value)}*

Arguments

enum-constants↓ A sequence of one or more symbols naming the elements of the enumeration.
entry-name↓ A symbol naming an element of the enumeration.
entry-value↓ An integer specifying the value of entry-name.

Description

The FLI type :enum converts between a Lisp symbol and the C enum type. Each entry in enum-constants can either consist of a symbol entry-name, in which case the first entry has a value 0, or of a list of a symbol entry-name and its corresponding integer value entry-value.

:enumeration is a synonym for :enum.

Examples

See define-c-enum, for an example using the :enum type.

See also

define-c-enum
2.1.1 Integral types
8 Type Reference

`:fixnum`  

Summary
Converts between a Lisp fixnum and a 32 bit raw integer.

Package

Keyword

Syntax
`:fixnum`

Description
The FLI type `:fixnum` converts between a Lisp fixnum and a 32 bit integer in C.

See also

2.1.1 Integral types

`:float`

Summary
Converts a Lisp single float to a C `float`.

Package

Keyword

Syntax
`:float`

Description
The FLI type `:float` converts between a Lisp single float and the C `float` type.

Compatibility note
In LispWorks 4.4 and previous on Windows and Linux platforms, all Lisp floats are doubles. In later versions, there are three disjoint Lisp float types in 32-bit LispWorks and two in 64-bit LispWorks, on all platforms.

See also

:double
2.1.2 Floating point types
8 Type Reference

:float-complex

Summary
Converts a Lisp single float complex number to a C float complex.

Package
keyword
Syntax
:float-complex

Description
The FLI type :float-complex converts between a Lisp (complex single-float) and the C float complex type.

See also
:double-complex
2.1.3 Complex number types

:foreign-array

Summary
Converts between a FLI array and a foreign array type.

Package
keyword
Syntax
:foreign-array type dimensions

Arguments

<table>
<thead>
<tr>
<th>type</th>
<th>The type of the elements of the array.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimensions</td>
<td>A list containing the dimensions of the array.</td>
</tr>
</tbody>
</table>

Description
The FLI type :foreign-array converts between FLI arrays and the foreign array type. It creates an array with the dimensions specified in dimensions, of elements of the type specified by type.

The :foreign-array type is an aggregate type. In particular, :foreign-array with more than one dimension is an array containing embedded arrays, not an array of pointers.
Notes

Only use the `:foreign-array` type when the corresponding foreign code uses an array with a constant declared size. If you need a dynamically sized array, then use a pointer type, allocate the array using the `nelems` argument to `allocate-foreign-object` or `with-dynamic-foreign-objects` and use `dereference` to access the elements. The pointer type is more efficient than making `:foreign-array` types dynamically with different dimensions because the FLI caches information about every different FLI type descriptor that is used.

Examples

The following code defines a 3 by 4 foreign array with elements of type `:byte`.

```lisp
(setq farray (fli:allocate-foreign-object
              :type `(:foreign-array :byte (3 4)))
```

The type of this is equivalent to the C declaration:

```c
signed char array2[3][4];
```

See also

`:c-array
foreign-aref
foreign-array-pointer
2.2.1 Arrays

---

**foreign-block-pointer**

*FLI Type Descriptor*

Summary

The foreign type corresponding to the opaque "Block" object in C and derived languages.

Package

fli

Syntax

*foreign-block-pointer*

Description

The FLI type `foreign-block-pointer` corresponds to the opaque "Block" object in C and derived languages that are introduced in CLANG and used by Apple.

A foreign block pointer should be regarded as opaque, and should not be manipulated or used except as described in 5.7 Block objects in C (foreign blocks).

Notes

A foreign block that is allocated directly by the Lisp side (for example by `allocate-foreign-block` or `with-foreign-block`) prints as "lisp-foreign-block-pointer".

*foreign-block-pointer* is implemented in LispWorks for Macintosh only.
See also

allocate-foreign-block
define-foreign-block-callable-type
define-foreign-block-invoker
foreign-block-copy
foreign-block-release
free-foreign-block
released-foreign-block-pointer
with-foreign-block
with-local-foreign-block
5.7 Block objects in C (foreign blocks)

:FUNCTION

Summary

Converts between Lisp and the C function type.

Package

keyword

Syntax

:function &optional args-spec return-spec &key calling-convention

Arguments

args-spec⇓ A list of foreign types.

return-spec⇓ A foreign type.

calling-convention⇓ A keyword naming the calling convention.

Description

The FLI type :function allows for conversion from the C function type. It is typically used in conjunction with the
:pointer type to reference an existing foreign function.

args-spec and return-spec specify the argument types and return type respectively.

calling-convention is as described for define-foreign-function.

Examples

The following code lines present a definition of a pointer to a function type, and a corresponding C definition of the type. The
function type is defined for a function which takes as its arguments an integer and a pointer to a void, and returns an integer value.

(:pointer (:function (:int (:pointer :void)) :int))

int (*)( int, void * )
8 Type Reference

See also

:int8
:int16
:int32
:int64
:intmax
:intptr

Summary

The signed sized integer types.

Package

keyword

Syntax

:int8
:int16
:int32
:int64
:intmax
:intptr

Description

FLI types are defined for integers of particular sizes. These are equivalent to the types defined by ISO C99. For example, Lisp :int8 is ISO C99 int8_t.

The types have these meanings:

:int8 8-bit signed integer.
:int16 16-bit signed integer.
:int32 32-bit signed integer.
:int64 64-bit signed integer.
:intmax The largest type of signed integer available.
:intptr A signed integer the same size as a pointer.

See also

:uint8
2.1.1 Integral types

: int

**FLI Type Descriptor**

Summary

Converts between a Lisp integer and a C int type.

Package

keyword

Syntax

: int

Description

The FLI type : int converts between an Lisp integer and a C int type. It is equivalent to the : signed and (:signed : int) types.

See also

: signed

2.1.1 Integral types

:lisp-array

**FLI Type Descriptor**

Summary

A foreign type which passes the address of a Lisp array direct to C.

Package

keyword

Syntax

:lisp-array &optional type

Arguments

A list. The default is nil.

type

Description

The FLI type :lisp-array accepts a Lisp array and passes a pointer to the first element of that array. The Lisp array may be non-simple.

It is vital that the garbage collector does not move the Lisp array, hence :lisp-array checks that the array is statically allocated, or allocated pinnable and pinned using with-pinned-objects.
Note also that the Lisp garbage collector does not know about the array in the C code. Therefore, if the C function retains a pointer to the array, then you must ensure the Lisp object is not collected, for example by retaining a pointer to it in Lisp.

The argument type, if non-nil, is a list (element-type &rest dimensions) and is used to check the element type and dimensions of the Lisp array passed.

Examples

This C function fills an array of doubles from an array of single floats.

Windows version:

```c
__declspec(dllexport) void __cdecl ProcessFloats(int count, float * fvec, double * dvec)
{
    for(--count ; count >= 0 ; count--) {
        dvec[count] = fvec[count] * fvec[count];
    }
}
```

Non-Windows version:

```c
void ProcessFloats(int count, float * fvec, double * dvec)
{
    for(--count ; count >= 0 ; count--) {
        dvec[count] = fvec[count] * fvec[count];
    }
}
```

The following Lisp code demonstrates the use of :lisp-array in a call to ProcessFloats:

```lisp
(fli:define-foreign-function (process-floats
    "ProcessFloats")
 ((count :int)
  (fvec :lisp-array)
  (dvec :lisp-array)))

(defun test-process-floats (length)
  (let ((f-vector (make-array length
                      :element-type 'single-float
                      :initial-contents
                      (loop for x below length
                            collect
                            (coerce x 'single-float)))
                      :allocation :static))
    (d-vector (make-array length
                      :element-type 'double-float
                      :initial-element 0.0D0
                      :allocation :static)))
    (process-floats length f-vector d-vector)
    (dotimes (x length)
      (format t "f-vector[-D] = ~A; d-vector[-D] = ~A-%%"
              x (aref f-vector x)
              x (aref d-vector x))))
)
```

Now:

```lisp
(test-process-floats 3)
```
prints:

```
single-array[0] = 0.0; double-array[0] = 0.0
single-array[1] = 1.0; double-array[1] = 1.0
```

See also

`:lisp-simple-1d-array

with-dynamic-lisp-array-pointer

with-pinned-objects

---

### :lisp-double-float

**FLI Type Descriptor**

**Summary**

A synonym for `double`.

**Package**

**keyword**

**Syntax**

```
:lisp-double-float
```

**Description**

The FLI type `:lisp-double-float` is the same as the FLI `:double` type.

See also

`:double

2.1.2 Floating point types

---

### :lisp-float

**FLI Type Descriptor**

**Summary**

Converts between any Lisp float and the C `double` type or the C `float` type.

**Package**

**keyword**

**Syntax**

```
:lisp-float &optional float-type
float-type ::= :single | :double
```
8 Type Reference

Arguments

float-type

Determines the C type to convert to. The default is :single.

Description

The FLI type :lisp-float converts between any Lisp float and either the C float or the C double type. The default is to convert to the C float type, but by specifying :double for float-type, conversion occurs between any Lisp float and the C double type.

See also

:double

:float

2.1.2 Floating point types

:lisp-simple-1d-array

FLI Type Descriptor

Summary

A foreign type which passes the address of a Lisp simple vector direct to C.

Package

keyword

Syntax

:lisp-simple-1d-array &optional type

Arguments

type

A list. The default is nil.

Description

The FLI type :lisp-simple-1d-array accepts a Lisp simple vector and passes a pointer to the first element of that vector. The Lisp vector must be simple. That is, it does not have a fill pointer, is not adjustable, and it is not a displaced array.

It is vital that the garbage collector does not move the Lisp vector, hence :lisp-simple-1d-array checks that the vector is statically allocated or allocated pinnable, in which case it is pinned implicitly as if by with-pinned-objects.

The argument type, if non-nil, is a list (element-type &rest dimensions) and is used to check the element type and dimensions of the Lisp array passed.

See also

:lisp-array

with-dynamic-lisp-array-pointer
*:lisp-single-float*  

**FLI Type Descriptor**

**Summary**
A synonym for `:float`.

**Package**
*keyword*

**Syntax**

```
:lisp-single-float
```

**Description**
The FLI type `:lisp-single-float` is the same as the FLI `:float` type.

**See also**

`:float`  

2.1.2 Floating point types

---

*:long*  

**FLI Type Descriptor**

**Summary**
Converts between a Lisp `integer` and a C `long`.

**Package**
*keyword*

**Syntax**

```
:long &optional integer-type
```

```
integer-type ::= :int | :double | :long
```

**Arguments**

**Description**
The FLI type `:long` converts between the Lisp `integer` type and the C `long` type. See A comparison between Lisp and C long types for comparisons between Lisp and C long types.
A comparison between Lisp and C long types

<table>
<thead>
<tr>
<th>Lisp type</th>
<th>FLI type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>:long</td>
<td>long</td>
</tr>
<tr>
<td>integer</td>
<td>:long :int</td>
<td>long</td>
</tr>
<tr>
<td>integer</td>
<td>:long :double</td>
<td>long double</td>
</tr>
<tr>
<td>integer</td>
<td>:long :long:long-long</td>
<td>long long</td>
</tr>
</tbody>
</table>

See also

:long
:long-long
:short

2.1.1 Integral types

:long-long

FLI Type Descriptor

Summary

Converts between a Lisp integer and a signed C long long.

Package

keyword

Syntax

:long-long

Description

The FLI type :long-long converts between the Lisp integer type and the C long long type.

Notes

This is supported only on platforms where the C long long type is the same size as the C long type.

See also

:long

2.1.1 Integral types
8 Type Reference

:one-of

Summary
Converts between Lisp and C types of the same underlying type.

Package

keyword

Syntax

:one-of &rest types

Arguments

types ⇐ A list of types sharing the same underlying type.

Description

The FLI type :one-of is used to allocate an object which can be one of a number of types specified by types. The types must have the same underlying structure, which means they must have the same size and must be referenced in the same manner. The FLI :one-of type is useful when a foreign function returns a value whose underlying type is known, but whose exact type is not.

Examples

In the following example, a :one-of type is allocated.

```lisp
(setq thing (fli:allocate-foreign-object
                :type '(:one-of :ptr :int :unsigned)))
```

If thing is set to be 100 using dereference, it is taken to be an object of type :int, as this is the first element in the sequence of types defined by :one-of which matches the type of the number 100.

```lisp
(setf (fli:dereference thing) 100)
```

However, if thing is now dereferenced, it is returned as a pointer to the address 100 (Or hex address 64), as there is no method for determining the type of thing, and therefore the first element in the list of :one-of is used.

```lisp
(fli:dereference thing)
```

See also

:union
**:pointer**

**Summary**
Defines a C-style FLI pointer to an object of a specified type.

**Package**

**keyword**

**Syntax**

```
:pointer type
:ptr type
```

**Arguments**

- `type` The type of FLI object pointed to by the pointer.

**Description**

The FLI type `:pointer` is part of the FLI implementation of pointers. It defines a C-style pointer to an object of `type`. Passing `nil` instead of a pointer is treated the same as passing a null pointer (that is, a pointer to address 0).

`:ptr` is a synonym for `:pointer`.

For more details on pointers, including examples on pointer coercion, dereferencing, making, and copying see [FLI Pointers](#).

**See also**

- `copy-pointer`
- `dereference`
- `make-pointer`
- `*null-pointer*`
- [2.1.6 Pointer types](#)

---

**:ptdiff-t**

**FLI Type Descriptor**

**Summary**

Converts between a Lisp integer and an ISO C `ptrdiff_t`.

**Package**

**keyword**

---

178
8 Type Reference

Syntax

:ptrdiff-t

Description

The FLI type :ptrdiff-t converts between a Lisp integer and an ISO C `ptrdiff_t` type, which is a signed integer representing the difference in bytes between two pointers.

:reference

FLI Type Descriptor

Summary

Passes a foreign object of a specified type by reference, and automatically dereferences the object.

Package

keyword

Syntax

:reference type &key allow-null lisp-to-foreign-p foreign-to-lisp-p

Arguments

<table>
<thead>
<tr>
<th>type</th>
<th>The type of the object to pass by reference.</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow-null</td>
<td>A boolean.</td>
</tr>
<tr>
<td>lisp-to-foreign-p</td>
<td>If non-nil, allow conversion from Lisp to the foreign language. The default value is t.</td>
</tr>
<tr>
<td>foreign-to-lisp-p</td>
<td>If non-nil, allow conversion from the foreign language to Lisp. The default value is t.</td>
</tr>
</tbody>
</table>

Description

The FLI type :reference is essentially the same as a :pointer type, except that :reference is automatically dereferenced when it is processed.

The :reference type is useful as a foreign function argument. When a function is called with an argument of the type (:reference type), an object of type is dynamically allocated across the scope of the foreign function, and is automatically de-allocated once the foreign function terminates. The value of the argument is not copied into the temporary instance of the object if lisp-to-foreign-p is nil, and similarly, the return value is not copied back into a Lisp object if foreign-to-lisp-p is nil.

If allow-null is non-nil and the input argument is nil then a null pointer is passed instead of a reference to an object containing nil. allow-null defaults to nil.

Notes

If the argument is of an aggregate type and foreign-to-lisp-p is true, then a malloc'd copy is made which you should later free explicitly. It is usually better to use :pointer, make the temporary foreign object using with-dynamic-foreign-objects and then copy whatever slots you need into a normal Lisp object on return.
8 Type Reference

Examples

In the following example an :int is allocated, and a pointer to the integer is bound to the Lisp variable number. Then a pointer to number, called point1, is defined. The pointer point1 is set to point to number, itself a pointer, but to an :int.

(setq number (fli:allocate-foreign-object :type :int))

(setf (fli:dereference number) 42)

(setq point1 (fli:allocate-foreign-object :type '(:pointer :int)))

(setf (fli:dereference point1) number)

If point1 is dereferenced, it returns a pointer to an :int. To get at the value stored in the integer, we need to dereference twice:

(fli:dereference (fli:dereference point1))

However, if we dereference point1 as a :reference, we only have to dereference it once to get the value:

(fli:dereference point1 :type '(:reference :int))

See also

:reference-pass
:reference-return

:reference-pass  

FLI Type Descriptor

Summary

Passes an object from Lisp to the foreign language by reference.

Package

keyword

Syntax

:reference-pass type &key allow-null

Arguments

type The type of the object to pass by reference.
allow-null A boolean.

Description

The FLI type :reference-pass is equivalent to:
8 Type Reference

(:reference :lisp-to-foreign-p t
           :foreign-to-lisp-p nil)

See :reference for the details of how type and allow-null are used.

See also

:reference
:reference-return

:reference-return

FLI Type Descriptor

Summary

Passes an object from the foreign language to Lisp by reference.

Package

keyword

Syntax

:reference-return type &key allow-null

Arguments

type↓ The type of the object to return by reference.
allow-null↓ A boolean.

Description

The FLI type :reference-return is equivalent to:

!(:reference :lisp-to-foreign-p nil
            :foreign-to-lisp-p t)

See :reference for the details of how type and allow-null are used.

See also

:reference
:reference-pass

released-foreign-block-pointer

FLI Type Descriptor

Summary

The type of foreign blocks that have been released.
8 Type Reference

Package

fli

Syntax

released-foreign-block-pointer

Description

The FLI type released-foreign-block-pointer is the type of released foreign blocks.

The system marks foreign blocks that have been released by foreign-block-release as being of foreign type released-foreign-block-pointer.

See also

foreign-block-pointer
foreign-block-release

:short

FLI Type Descriptor

Summary

Converts between a Lisp fixnum type and a C short type.

Package

keyword

Syntax

:short &optional integer-type

integer-type ::= :int

Arguments

integer-type If specified, must be :int, which associates a Lisp fixnum with a C int.

Description

The FLI type :short associates a Lisp fixnum with a C short.

The FLI types :short, (:short :int), (:signed :short), and (:signed :short :int) are equivalent.

See also

:int
:signed

2.1.1 Integral types
8 Type Reference

:signed

Summary

Converts between a Lisp integer and a foreign signed integer.

Package

keyword

Syntax

:s signed &optional integer-type


Arguments

integer-type          The type of the signed integer.

Description

The FLI type :signed converts between a Lisp integer and a foreign signed integer. The optional integer-type argument specifies other kinds of signed integer types. See Table A comparison of Lisp and C signed types for a comparison between Lisp and C signed types.

A comparison of Lisp and C signed types

<table>
<thead>
<tr>
<th>Lisp type</th>
<th>FLI type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>:signed</td>
<td>signed int</td>
</tr>
<tr>
<td>fixnum</td>
<td>:signed :byte</td>
<td>signed char</td>
</tr>
<tr>
<td>fixnum</td>
<td>:signed :char</td>
<td>signed char</td>
</tr>
<tr>
<td>fixnum</td>
<td>:signed :short</td>
<td>signed short</td>
</tr>
<tr>
<td>integer</td>
<td>:signed :int</td>
<td>signed int</td>
</tr>
<tr>
<td>integer</td>
<td>:signed :long</td>
<td>signed long</td>
</tr>
<tr>
<td>fixnum</td>
<td>:signed :short :int</td>
<td>signed short</td>
</tr>
<tr>
<td>integer</td>
<td>:signed :long :int</td>
<td>signed long</td>
</tr>
</tbody>
</table>

See also

cast-integer
:unsigned
2.1.1 Integral types
:size-t

Summary

Converts between a Lisp integer and an ISO C size_t.

Package

keyword

Syntax

:size-t

Description

The FLI type :size-t converts between a Lisp integer and an ISO C size_t type, which is an unsigned integer representing the size of an object in bytes.

See also

:size-t

:ssize-t

Summary

Converts between a Lisp integer and the platform-specific ssize_t type.

Package

keyword

Syntax

:ssize-t

Description

The FLI type :ssize-t converts between a Lisp integer and a platform-specific ssize_t type, which is a signed integer representing the size of an object in bytes.

See also

:size-t
Summary
Converts between a FLI structure and a C `struct`.

Package

Keyword

Syntax

```
:struct &rest slots
slots ::= {symbol | (symbol slot-type)} *
slot-type ::= type | (:bit-field integer-type size)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slots</td>
<td>A sequence of one or more slots making up the structure.</td>
</tr>
<tr>
<td>symbol</td>
<td>A symbol naming the slot.</td>
</tr>
<tr>
<td>type</td>
<td>The slot type. If no type is given it defaults to an <code>:int</code>.</td>
</tr>
<tr>
<td>integer-type</td>
<td>An integer type. Only <code>:int</code>, <code>:unsigned :int</code> and <code>:signed :int</code> are guaranteed to work on all platforms.</td>
</tr>
<tr>
<td>size</td>
<td>An integer specifying a number of bits for the field.</td>
</tr>
</tbody>
</table>

Description

The FLI type `:struct` is an aggregate type, and converts between a FLI structure and a C `struct` type. The FLI structure consists of a collection of one or more slots. Each slot has a name `symbol` and a type `type`. A structure can also contain bit fields, which are integers of type `integer-type` with `size` bits.

The `foreign-slot-names`, `foreign-slot-type`, and `foreign-slot-value` functions can be used to access and change the slots of the structure. The convenience FLI function `define-c-struct` is provided to simplify the definition of structures.

Examples

In the following example a structure for passing coordinates to Windows functions is defined.

```
(fli:define-c-struct tagPOINT (x :long) (y :long))
```

An instance of the structure is allocated and bound to the Lisp variable `place`.

```
(setq place
    (fli:allocate-foreign-object :type 'tagPOINT))
```

Finally, the `x` slot of `place` is set to be 4 using `fli:foreign-slot-value`.

```
(setf (fli:foreign-slot-value place 'x) 4)
```
8 Type Reference

See also

```lisp
define-c-struct
define-slot-names
define-slot-offset
define-slot-pointer
define-slot-type
define-slot-value
```

### :time-t

**FLI Type Descriptor**

**Summary**

Converts between a Lisp integer and the platform-specific time_t type.

**Package**

```lisp
keyword
```

**Syntax**

```lisp
:time-t
```

**Description**

The FLI type :time-t converts between a Lisp integer and an ISO C time_t type, which is an integer type used for storing system time values.

### :uint8

### :uint16

### :uint32

### :uint64

### :uintmax

### :uintptr

**FLI Type Descriptors**

**Summary**

The unsigned sized integer types.

**Package**

```lisp
keyword
```

**Syntax**

```lisp
:uint8
:uint16
```
8 Type Reference

:uint32
:uint64
:uintmax
:uintptr

Description

FLI types are defined for integers of particular sizes. These are equivalent to the types defined by ISO C99. For example, Lisp :uint8 is ISO C99 uint8_t.

The types have these meanings:

:uint8 8-bit unsigned integer.
:uint16 16-bit unsigned integer.
:uint32 32-bit unsigned integer.
:uint64 64-bit unsigned integer.
:uintmax The largest type of unsigned integer available.
:uintptr An unsigned integer the same size as a pointer.

See also

:uint8

2.1.1 Integral types

:union

Summary

Converts between a FLI union and a C union type.

Package

keyword

Syntax

:union &rest slots
slots ::= {symbol | (symbol type)}*

Arguments

slots A sequence of one or more slots making up the union.
symbol A symbol naming the slot.
type The slot type. If no type is given, it defaults to an :int.
Description

The FLI type `:union` is an aggregate type, and converts between a FLI union and a C `union` type. The FLI union consists of a collection of one or more slots, only one of which can be active at any one time. The size of the whole union structure is therefore equal to the size of the largest slot. Each slot has a name `symbol` and a type `type`.

The `foreign-slot-names`, `foreign-slot-type`, and `foreign-slot-value` functions can be used to access and change the slots of the union. The convenience FLI function `define-c-union` is provided to simplify the definition of unions.

Examples

In the following example a union type with two slots is defined.

```lisp
(fli:define-c-union my-number
  (small :byte) (large :int))
```

An instance of the union is allocated and bound to the Lisp variable `length`.

```lisp
(setq length
  (fli:allocate-foreign-object :type 'my-number))
```

Finally, the `small` slot of the union is set equal to 24.

```lisp
(setf (fli:foreign-slot-value length 'small))
```

See also

`define-c-union`
`foreign-slot-names`
`foreign-slot-offset`
`foreign-slot-pointer`
`foreign-slot-type`
`foreign-slot-value`

2.2.3 Structures and unions

:unsigned

FLI Type Descriptor

Summary

Converts between a Lisp integer and a foreign unsigned integer.

Package

keyword

Syntax

:unsigned &optional integer-type

Arguments

integer-type

The type of the unsigned integer.

Description

The FLI type :unsigned converts between a Lisp integer and a foreign unsigned integer. The optional integer-type argument specifies other kinds of unsigned integer types. See Table A comparison of Lisp and C unsigned types for a comparison between Lisp and C unsigned types.

A comparison of Lisp and C unsigned types

<table>
<thead>
<tr>
<th>Lisp type</th>
<th>FLI type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>:unsigned</td>
<td>unsigned int</td>
</tr>
<tr>
<td>fixnum</td>
<td>:unsigned :byte</td>
<td>unsigned char</td>
</tr>
<tr>
<td>fixnum</td>
<td>:unsigned :char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>fixnum</td>
<td>:unsigned :short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>integer</td>
<td>:unsigned :int</td>
<td>unsigned int</td>
</tr>
<tr>
<td>integer</td>
<td>:unsigned :long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>fixnum</td>
<td>:unsigned :short :int</td>
<td>unsigned short</td>
</tr>
<tr>
<td>integer</td>
<td>:unsigned :long :int</td>
<td>unsigned long</td>
</tr>
</tbody>
</table>

See also

cast-integer
:signed

2.1.1 Integral types

vector-char2
vector-char3
vector-char4
vector-char8
vector-char16
vector-char32
vector-uchar2
vector-uchar3
vector-uchar4
vector-uchar8
vector-uchar16
vector-uchar32
vector-short2
vector-short3
vector-short4
vector-short8
vector-short16
vector-short32
vector-ushort2
vector-ushort3
vector-ushort4
vector-ushort8
vector-ushort16
vector-ushort32
vector-int2
vector-int3
vector-int4
vector-int8
vector-int16
vector-uint2
vector-uint3
vector-uint4
vector-uint8
vector-uint16
vector-long1
vector-long2
vector-long3
vector-long4
vector-long8
vector-ulong1
vector-ulong2
vector-ulong3
vector-ulong4
vector-ulong8
vector-float2
vector-float3
vector-float4
vector-float8
vector-float16
vector-double2
8 Type Reference

vector-double3
vector-double4
vector-double8

Summary
Convert between Lisp vectors and C vector types.

Package
fli

Syntax
vector-char2
vector-char3
vector-char4
vector-char8
vector-char16
vector-char32
vector-uchar2
vector-uchar3
vector-uchar4
vector-uchar8
vector-uchar16
vector-uchar32
vector-short2
vector-short3
vector-short4
vector-short8
vector-short16
vector-short32
vector-ushort2
vector-ushort3
vector-ushort4
vector-ushort8
vector-ushort16
vector-ushort32
vector-int2
8 Type Reference

vector-int3
vector-int4
vector-int8
vector-int16
vector-uint2
vector-uint3
vector-uint4
vector-uint8
vector-uint16
vector-long1
vector-long2
vector-long3
vector-long4
vector-long8
vector-ulong1
vector-ulong2
vector-ulong3
vector-ulong4
vector-ulong8
vector-float2
vector-float3
vector-float4
vector-float8
vector-float16
vector-double2
vector-double3
vector-double4
vector-double8

Description

See 2.2.4 Vector types for a full description.
**void**

**Summary**

Represents the C `void` type.

**Package**

*keyword*

**Syntax**

`:void`

**Description**

The FLI type `:void` represents the C `void` type. It can only be used in a few limited circumstances, as the:

- *result-type* of a `define-foreign-function`, `define-foreign-funcallable` or `define-foreign-callable` form. In this case, it means that no values are generated.

- element type of a `:pointer` type, that is `(:pointer :void)`. Any FLI pointer can be converted to this type, for example when used like this as the argument type in `define-foreign-function`.

- element type of a FLI pointer when memory is not being allocated, for example in a call to `make-pointer`. It is an error to dereference a FLI pointer with element type `:void` (but `with-coerced-pointer` can be used).

- expansion of a `define-c-typedef` or `define-foreign-type` form. The type defined in this way can only be used in situations where `:void` is allowed.

**See also**

`:pointer`

2.5 The void type

**volatile**

**Summary**

Corresponds to the C `volatile` type.

**Package**

*keyword*

**Syntax**

`:volatile &optional type`

**Arguments**
8 Type Reference

### type

The type of the volatile. The default is :int.

#### Description

The FLI type :volatile corresponds to the C++ volatile type. The behavior of a :volatile is exactly the same as the behavior of its type, and it is only included to ease the readability of FLI code and for naming conventions.

#### See also

:const

---

### :wchar-t

**FLI Type Descriptor**

#### Summary

Converts between a Lisp character and a C wchar_t.

#### Package

**keyword**

#### Syntax

:wchar-t

#### Description

The FLI type :wchar-t converts between a Lisp character and a C wchar_t type.

---

### :wrapper

**FLI Type Descriptor**

#### Summary

Allows the specification of automatic conversion functions between Lisp and an instance of a FLI type.

#### Package

**keyword**

#### Syntax

:wrapper foreign-type &key lisp-to-foreign foreign-to-lisp

#### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreign-type</td>
<td>The underlying type to wrap.</td>
</tr>
<tr>
<td>lisp-to-foreign</td>
<td>Code specifying how to convert between Lisp and the FLI.</td>
</tr>
<tr>
<td>foreign-to-lisp</td>
<td>Code specifying how to convert between the FLI and Lisp.</td>
</tr>
</tbody>
</table>

---
8 Type Reference

Description

The FLI type :wrapper allows for an extra level of conversion between Lisp and a foreign language through the FLI. With the :wrapper type you can use lisp-to-foreign and foreign-to-lisp to specify conversion functions from and to an instance of another type foreign-type. Whenever data is passed to the object, or received from the object it is passed through the conversion function. See below for an example of a use of :wrapper to pass values to an :int as strings, and to receive them back as strings when the pointer to the :int is dereferenced.

Examples

In the following example an :int is allocated with a wrapper to allow the :int to be accessed as a string.

```lisp
(setq wrap (fli:allocate-foreign-object
               :type '(:wrapper :int
                        :lisp-to-foreign read-from-string
                        :foreign-to-lisp prin1-to-string)))
```

The object pointed to by wrap, although consisting of an underlying :int, is set with dereference by passing a string, which is automatically converted using the Lisp function read-from-string. Similarly, when wrap is dereferenced, the value stored as an :int is converted using prin1-to-string to a Lisp string, which is the returned. The following two commands demonstrate this.

```lisp
(setf (fli:dereference wrap) "#x100")

(fli:dereference wrap)
```

The first command sets the value stored at wrap to be 256 (100 in hex), by passing a string to it. The second command dereferences the value at wrap, but returns it as a string. The pointer wrap can be coerced to return the value as an actual :int as follows:

```lisp
(fli:dereference wrap :type :int)
```

See also

2.4 Encapsulated types
9 The Foreign Parser

9.1 Introduction

The Foreign Parser automates the generation of Foreign Language Interface defining forms, given files containing C declarations.

The result does often need some editing, due to ambiguities in C.

9.1.1 Requirements

The Foreign Parser requires a C preprocessor, so you must have a suitable preprocessor installed on your machine.

By default LispWorks invokes cl.exe (VC++) on Windows and cc on other platforms. If you have this installed, then make sure it is on your PATH.

On Windows, if you don't have cl.exe, download the VC++ toolkit from Microsoft.

Preprocessors known to work with LispWorks are:

- Microsoft Visual Studio's cl.exe.
- cc
- gcc

To use a preprocessor other than the default, set the variable foreign-parser:*preprocessor*, for example:

(setf foreign-parser:*preprocessor* "gcc")

9.2 Loading the Foreign Parser

The Foreign Parser is in a loadable module foreign-parser.

Load it by:

(require "foreign-parser")

9.3 Using the Foreign Parser

The interface is the function foreign-parser:process-foreign-file.

Suppose we wish to generate the FLI definitions which interface to the C example from 5.2.4 Modifying a string in a C function. The header file test.h needs to be slightly different depending on the platform.

Windows version:

__declspec(dllexport) void __cdecl modify(char *string)
9 The Foreign Parser

Non-Windows version:

```lisp
void modify(char *string)
```

1. Load the Foreign Parser:

```lisp
(require "foreign-parser")
```

2. Now generate prototype FLI definitions:

```lisp
(foreign-parser:process-foreign-file
 "test.h"
 :case-sensitive nil)
=>
;;;; Output dff file #P"test-dff.lisp"
;;;; Parsing source file "test.h"

;;;; Process-foreign-file : Preprocessing file
;;;; Process-foreign-file : Level 1 parsing
;;;; Process-foreign-file : Selecting foreign forms
NIL
```

3. You should now have a Lisp file `test-dff.lisp` containing a form like this:

```lisp
(fli:define-foreign-function
 (modify "modify" :source)
 ((string (:pointer :char)))
 :result-type
 :void
 :language
 :c
 :calling-convention
 :cdecl)
```

4. This edited version passes a string using `:ef-mb-string`:

```lisp
(fli:define-foreign-function
 (modify "modify" :source)
 ((string (:reference (:ef-mb-string :limit 256))))
 :result-type
 :void
 :language
 :c
 :calling-convention
 :cdecl)
=>
MODIFY
```

5. Create a DLL containing the C function.

6. Load the foreign code by:

```lisp
(fli:register-module "test.dll")
```

or:

```lisp
(fli:register-module "/tmp/test.so")
```
7. Call the C function from LISP:

```
(modify "Hello, I am in LISP")
=> NIL
"'Hello, I am in LISP' modified in a C function"
```

9.4 Using the LispWorks Editor

The LispWorks Editor’s C Mode offers a convenient alternative to using `foreign-parser:process-foreign-file` directly as above. It also allows you to generate and load a C object file.

To use this, you should be familiar with the LispWorks Editor as described in the *LispWorks IDE User Guide* and the *Editor User Guide*.

9.4.1 Processing Foreign Code with the Editor

1. Open the file `test.h` in the LispWorks Editor. Note that the buffer is in C Mode, indicated by "(C)" in the mode line.
2. Use the menu command *Buffer > Evaluate*, or equivalently run *Meta+X Evaluate Buffer*.
3. A new buffer named `test.h (C->LISP)` is created. It contains the prototype FLI definition forms generated by `foreign-parser:process-foreign-file`.
4. You can now edit the Lisp forms if necessary (note that your new buffer is in Lisp mode) and save them to file. Follow the previous example from Step 4.

9.4.2 Compiling and Loading Foreign Code with the Editor

1. Open the file `test.c` in the LispWorks Editor. Note that the buffer is in C Mode, indicated by "(C)" in the mode line.
2. Use the menu command *Buffer > Compile*, or equivalently run *Meta+X Compile Buffer*.
3. Your C file is compiled with the same options as `lw:compile-system` would use, and the object file is loaded. The object file name is printed in the Output tab. It is written in your temporary directory (see `create-temp-file`) and deleted after `register-module` is called on it.

9.5 Foreign Parser Reference

*preprocessor*  

*Variable*

**Summary**

The default value for the `preprocessor` used by `process-foreign-file`.

**Package**

`foreign-parser`

**Initial Value**

"cc" on Non-Windows systems and "cl" on Windows.
Description

The variable *preprocessor* provides the default value for the preprocessor used by process-foreign-file.

See also

*preprocessor-options*

process-foreign-file

**preprocessor-format-string**

Summary

Provides the default value for the preprocessor-format-string used by process-foreign-file.

Package

foreign-parser

Initial Value

On Windows:

""~A" /nologo /E ~A ~(D~A~)~/I"~A" ~/Tc ""~A""

On Non-Windows systems:

"~A ~E ~A ~(D~A~)~(I~A~)"~A"

Description

The variable *preprocessor-format-string* provides the default value for the preprocessor-format-string used by process-foreign-file.

See also

process-foreign-file

**preprocessor-include-path**

Summary

Provides the default value for the preprocessor-include-path used by process-foreign-file.

Package

foreign-parser

Initial Value

nil
Description

The variable `*preprocessor-include-path*` provides the default value for the `preprocessor-include-path` used by `process-foreign-file`.

See also

`process-foreign-file`

`*preprocessor-options*` Variable

Summary

Provides the default `preprocessor-options` passed to the `preprocessor` used by `process-foreign-file`.

Package

`foreign-parser`

Initial Value

`nil`

Description

The variable `*preprocessor-options*` provides the default `preprocessor-options` passed to the `preprocessor` used by `process-foreign-file`.

See also

`*preprocessor*`  
`process-foreign-file`

`process-foreign-file` Function

Summary

Parses foreign declarations to create Lisp FLI definition.

Package

`foreign-parser`

Signature

`process-foreign-file source &key dff language preprocess preprocessor preprocessor-format-string preprocessor-options include-path case-sensitive package`

Arguments

`source`  
One or more filenames.
9 The Foreign Parser

dff \downarrow A filename.

language \downarrow A keyword.

preprocess \downarrow A boolean.

preprocessor \downarrow A string.

preprocessor-format-string \downarrow A string.

preprocessor-options \downarrow A string.

include-path \downarrow A list.

case-sensitive \downarrow See description.

package \downarrow A package designator or nil.

Description

The function process-foreign-file takes a file or files of foreign declarations — usually header files — and parses them, producing `dff' files of Lisp definitions using define-foreign-function, define-foreign-variable, define-foreign-type, and so on, providing a Lisp interface to the foreign code.

source gives the name of the header files or file to be processed. The name of a file consists of source-file-name and source-file-type (typically .h).

dff is an output file which will contain the Lisp foreign function definitions. The default value is nil, in which case the dff file will be source-file-name-dff.lisp. (See source, above.)

language specifies the language the header files are written in. Currently the supported languages are :c (standard K&R C header files) and :ansi-c. The default value is :ansi-c.

preprocess, when non-nil, runs the preprocessor on the input files. The default value is t.

preprocessor-format-string should be a format string which is used to make a preprocessor command line. The format arguments are a pathname or string giving the preprocessor executable, a list of strings giving the preprocessor options, a list of strings giving macro names to define, a list of pathnames or strings contain the include path, and a source pathname. On Windows, the default contains options needed for VC++. The default is the value of *preprocessor-format-string*.

preprocessor is a string containing the pathname of the preprocessor program. By default this is the value of *preprocessor*.

preprocessor-options is a string containing command line options to be passed to the preprocessor if it is called. By default this is the value of *preprocessor-options*.

include-path should be a list of pathnames or strings that will be added as the include path for the preprocessor. The default is the value of *preprocessor-include-path*.

case-sensitive specifies whether to maintain case sensitivity in symbol names as in the source files. Values can be:

t The names of all Lisp functions and classes created are of the form |name|. This is the default value.

nil All foreign names are converted to uppercase and an error is signalled if any name clashes occur as a result of this conversion. For example, OneTwoTHREE becomes ONETWOTHREE.

:split-name Attempts to split the name up into something sensible. For example, OneTwoTHREE becomes ONE-TWO-THREE.
9 The Foreign Parser

:prefix
Changes lowercase to uppercase and concatenates the string with the string held in
sys:*prefix-name-string*. For example, OneTwoTHREE becomes
FOREIGN-ONETWOTHREE.

(:user-routine function-name)
Enables you to pass your own function for name formatting. Your function must take a string
argument and return a string result. It is not advised to use destructive functions (for example,
reverse) as this may cause unusual side effects.

If case-sensitive takes any other value, names are not changed.

package is used to generate an in-package form at the start of the output (dff) file. The name of the package designated by
package is used in this form. The default value of package is the value of *package*.

Note that in some cases the derived Lisp FLI definitions will not be quite correct, due to an ambiguity in C. char* can mean
a pointer to a character, or a string, and in many cases you will want to pass a string. Therefore, process-foreign-file
is useful for generating prototype FLI definitions, especially when there are many, but you do need to check the results when
char* is used.

See also

register-module
*preprocessor*
*preprocessor-options*
Glossary

aggregate type

Any FLI type which is made up of other FLI types. This can be either an array of instances of a given FLI type, or a structured object.

Arrays, string, structure, and unions are all aggregate types. Pointers are not aggregates.

callable function

A Lisp function, defined with the FLI macro `define-foreign-callable`, which can be called from a foreign language.

coered pointer

A coerced pointer is a pointer that is dereferenced with the :type key in order to return the value pointed to as a different type than specified by the pointer type. For example, a pointer to a byte can be coerced to return a boolean on dereferencing.

FLI

The Foreign Language Interface, which consists of the macros, functions, types and variables defined in the fli package.

FLI code

Code written in Lisp using the functions, macros and types in the fli package.

FLI function

A function in the fli package used to interface Lisp with a foreign language.

FLI type

A data type specifier in the fli package used to define data objects that interface between Lisp and the foreign language. For example, a C long might be passed to LispWorks through an instance of the FLI type :long, from which it is transferred to a Lisp integer.

foreign callable function

See callable function.

foreign function

A Lisp function, defined using the FLI macro `define-foreign-function`, which calls a function written in a foreign language. A foreign function contains no body, consisting only of a name and a list of arguments. The function in the foreign language provides the body of the foreign function.

foreign language

A language to which Lisp can be interfaced using the FLI. Currently the FLI interfaces to C, and therefore also the Win32 API functions.
Glossary

**immediate type**

See scalar type.

**pointer**

A FLI type consisting of an address and a type specification. A pointer normally points to the memory location of an instance of the type specified, although there might not actually be an allocated instance of the type at the pointer location.

A pointer is a boxed foreign object because it contains type information about the type it is pointing to (so that we can dereference it). In `C` a pointer can be represented by a single register.

**scalar type**

A FLI type that is not an aggregate type. The FLI type maps directly to a single foreign type such as integer, floating point, enumeration and pointer.

**wrapper**

A description of the :wrapper FLI type which "wraps" around an object, allowing data to be passed to or obtained from the object as though it was of a different type. A wrapper can be viewed as a set of conversion functions defined on the object which are automatically invoked when the wrapped object is accessed.
Index

A
accessors
dereference 93
foreign-aref 98
foreign-slot-value 110
foreign-typed-aref 112
aggregate FLI types 2 : FLI Types 17, 2.2 : Aggregate types 18
aggregate types 2 : FLI Types 17
align-of function 54
alloca function 55
allocate-dynamic-foreign-object function 55
allocate-foreign-block function 56
allocate-foreign-object function 57
allocating memory dynamically 1.4 : An example of dynamic memory allocation 16, allocate-foreign-object 58
Android 4.2.2 : ARM 32-bit calling conventions 35

B
Block object 5.7 : Block objects in C (foreign blocks) 50
:boolean FLI type descriptor 158
:byte FLI type descriptor 159

C
5.7 : Block objects in C (foreign blocks) 50
calling from Lisp 1.1.1 : Defining the FLI function 12, 5.7.1 : Calling foreign code that receives a block as argument 50
calling from Lisp with a block 5.7.1 : Calling foreign code that receives a block as argument 50
calling into Lisp 4.1 : Foreign callables and foreign functions 32
C++
calling from Lisp 1.1.1 : Defining the FLI function 12, 5.7.1 : Calling foreign code that receives a block as argument 50
calling from Lisp with a block 5.7.1 : Calling foreign code that receives a block as argument 50
calling into Lisp 4.1 : Foreign callables and foreign functions 32
calling convention specifying 4.2 : Specifying a calling convention 34
:c-array FLI type descriptor 159
cast-integer function 59
C code
declarations 9.1 : Introduction 196
Index

:char FLI type descriptor 161
CLANG 5.7: Block objects in C (foreign blocks) 50
condition classes
  foreign-type-error 114
connected-module-pathname function 59
  argument to register-module register-module 134
:const FLI type descriptor 161
convert-from-foreign-string function 61
convert-integer-to-dynamic-foreign-object function 62
convert-to-dynamic-foreign-string function 62
convert-to-foreign-string function 64
copy-pointer function 65

D
defc-pointer function 66
define-c-enum macro 67
define-c-struct macro 69
define-c-typedef macro 71
define-c-union macro 72
define-foreign-block-callable-type macro 74
define-foreign-block-invoker macro 75
define-foreign-callable macro 76 4.1: Foreign callables and foreign functions 32, 4.1.1: Strings and foreign callables 34
define-foreign-converter macro 79
define-foreign-forward-reference-type macro 81
define-foreign-funcallable macro 82
define-foreign-function macro 83 4.1: Foreign callables and foreign functions 32
define-foreign-pointer macro 87 3.1.1: Creating pointers 25
define-foreign-type macro 88 2: FLI Types 17
define-foreign-variable macro 89
define-opaque-pointer macro 92
defining FLI functions 1.2.2: Defining a FLI function 15
defining FLI types 1.2.1: Defining FLI types 14
defining forms
  ambiguity process-foreign-file 202
  automated generation 9.1: Introduction 196
defsystem macro 5.6: Incorporating a foreign module into a LispWorks image 50
dereference accessor 93
disconnect-module function 95
DLLs
  exporting functions from 5.5: Using DLLs within the LispWorks FLI 48
Index

documentation strings  2: FLI Types  17
:double FLI type descriptor  162
:double-complex FLI type descriptor  163
dynamic memory allocation  1.4: An example of dynamic memory allocation  16

E
:ef-mb-string FLI type descriptor  163
:ef-wc-string FLI type descriptor  164
Embedded dynamic modules  5.6: Incorporating a foreign module into a LispWorks image  49
:embedded-module member option for defsystem
type: cfile  5.6: Incorporating a foreign module into a LispWorks image  50
:enum FLI type descriptor  165
:enumeration FLI type descriptor  165
enum-symbols function  96
enum-symbol-value function  96
enum-symbol-value-pairs function  96
enum-values function  96
enum-value-symbol function  96
environment variable
  DYLD_LIBRARY_PATH register-module  136
  LD_LIBRARY_PATH register-module  135, register-module  136
  PATH register-module  135

F
fill-foreign-object function  97
:fixnum FLI type descriptor  166
FLI functions
defining  1.2.2: Defining a FLI function  15
FLI templates print-collected-template-info  133, start-collecting-template-info  145
FLI type constructors  2: FLI Types  17
FLI type descriptors
:boolean  158
:byte  159
:c-array  159
:char  161
:const  161
:double  162
:double-complex  163
:ef-mb-string  163
:ef-wc-string  164
:enum  165
:enumeration  165
Index

:fixnum 166
:float 166
:float-complex 167
:foreign-array 167
foreign-block-pointer 168  5.7: Block objects in C (foreign blocks)  50
:function 169
:int16 170
:int32 170
:int64 170
:int8 170
:int 171
:intmax 170
:intptr 170
:lisp-array 171
:lisp-double-float 173
:lisp-float 173
:lisp-simple-1d-array 174
:lisp-single-float 175
:long 175
:long-long 176
lpstr 5.2.8: Win32 API functions that handle strings 45
lpcstr 5.2.8: Win32 API functions that handle strings 46
lpcwstr 5.2.8: Win32 API functions that handle strings 45
lptstr 5.2.8: Win32 API functions that handle strings 46
:one-of 177
:pointer 178
:ptr 178
:ptrdiff-t 178
:reference 179
:reference-pass 180
:reference-return 181
released-foreign-block-pointer 181
:short 182
:signed 183
:size-t 184
:ssize-t 184
str 5.2.8: Win32 API functions that handle strings 45
:struct 185
:time-t 186
tstr 5.2.8: Win32 API functions that handle strings 46
:uint16 186

208
Index

:uint32 186
:uint64 186
:uint8 186
:uintmax 186
:uintptr 186
:union 187
:unsigned 188
vector-char16 189
vector-char2 189
vector-char3 189
vector-char32 189
vector-char4 189
vector-char8 189
vector-double2 190
vector-double3 190
vector-double4 191
vector-double8 191
vector-float16 190
vector-float2 190
vector-float3 190
vector-float4 190
vector-float8 190
vector-int16 190
vector-int2 190
vector-int3 190
vector-int4 190
vector-int8 190
vector-long1 190
vector-long2 190
vector-long3 190
vector-long4 190
vector-long8 190
vector-short16 190
vector-short2 189
vector-short3 189
vector-short32 190
vector-short4 190
vector-short8 190
vector-uchar16 189
vector-uchar2 189
vector-uchar3 189
foreign callable
  defining 4.1: Foreign callables and foreign functions 32
  passing and returning strings 4.1.1: Strings and foreign callables 33
foreign-function-pointer function 104
Foreign Parser 9.1: Introduction 196
foreign-slot-names function 105
foreign-slot-offset function 106
foreign-slot-pointer function 107
foreign-slot-type function 109
foreign-slot-value accessor 110
foreign-typed-aref accessor 112
foreign-type-equal-p function 113
foreign-type-error condition class 114
free function 115
free-foreign-block function 114
free-foreign-object function 115
:function FLI type descriptor 169
functions
  align-of 54
 alloca 55
  allocate-dynamic-foreign-object 55
  allocate-foreign-block 56
  allocate-foreign-object 57
  cast-integer 59
  connected-module-pathname 59
  convert-from-foreign-string 61
  convert-integer-to-dynamic-foreign-object 62
  convert-to-dynamic-foreign-string 62
  convert-to-foreign-string 64
  copy-pointer 65
  decf-pointer 66
  disconnect-module 95
  enum-symbols 96
  enum-symbol-value 96
  enum-symbol-value-pairs 96
  enum-values 96
  enum-value-symbol 96
  fill-foreign-object 97
  foreign-array-dimensions 100
  foreign-array-element-type 100
  foreign-array-pointer 101
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreign-block-copy</td>
<td>102</td>
</tr>
<tr>
<td>foreign-block-release</td>
<td>103</td>
</tr>
<tr>
<td>foreign-function-pointer</td>
<td>104</td>
</tr>
<tr>
<td>foreign-slot-names</td>
<td>105</td>
</tr>
<tr>
<td>foreign-slot-offset</td>
<td>106</td>
</tr>
<tr>
<td>foreign-slot-pointer</td>
<td>107</td>
</tr>
<tr>
<td>foreign-slot-type</td>
<td>109</td>
</tr>
<tr>
<td>foreign-type-equal-p</td>
<td>113</td>
</tr>
<tr>
<td>free</td>
<td>115</td>
</tr>
<tr>
<td>free-foreign-block</td>
<td>114</td>
</tr>
<tr>
<td>free-foreign-object</td>
<td>115</td>
</tr>
<tr>
<td>get-embedded-module</td>
<td>116</td>
</tr>
<tr>
<td>get-embedded-module-data</td>
<td>117</td>
</tr>
<tr>
<td>incf-pointer</td>
<td>118</td>
</tr>
<tr>
<td>install-embedded-module</td>
<td>119</td>
</tr>
<tr>
<td>make-integer-from-bytes</td>
<td>122</td>
</tr>
<tr>
<td>make-pointer</td>
<td>122</td>
</tr>
<tr>
<td>malloc</td>
<td>57</td>
</tr>
<tr>
<td>module-unresolved-symbols</td>
<td>124</td>
</tr>
<tr>
<td>null-pointer-p</td>
<td>125</td>
</tr>
<tr>
<td>pointer-address</td>
<td>126</td>
</tr>
<tr>
<td>pointer-element-size</td>
<td>127</td>
</tr>
<tr>
<td>pointer-element-type</td>
<td>128</td>
</tr>
<tr>
<td>pointer-element-type-p</td>
<td>129</td>
</tr>
<tr>
<td>pointer-eq</td>
<td>130</td>
</tr>
<tr>
<td>pointerp</td>
<td>131</td>
</tr>
<tr>
<td>pointer-pointer-type</td>
<td>132</td>
</tr>
<tr>
<td>print-collected-template-info</td>
<td>133</td>
</tr>
<tr>
<td>print-foreign-modules</td>
<td>133</td>
</tr>
<tr>
<td>process-foreign-file</td>
<td>200</td>
</tr>
<tr>
<td>register-module</td>
<td>134</td>
</tr>
<tr>
<td>replace-foreign-array</td>
<td>138</td>
</tr>
<tr>
<td>replace-foreign-object</td>
<td>140</td>
</tr>
<tr>
<td>set-locale</td>
<td>141</td>
</tr>
<tr>
<td>set-locale-encodings</td>
<td>142</td>
</tr>
<tr>
<td>setup-embedded-module</td>
<td>143</td>
</tr>
<tr>
<td>size-of</td>
<td>144</td>
</tr>
<tr>
<td>start-collecting-template-info</td>
<td>145</td>
</tr>
<tr>
<td>valid-foreign-type-p</td>
<td>146</td>
</tr>
</tbody>
</table>
Index

G
GCD  5.7 : Block objects in C (foreign blocks)  50
gni+  5.8 : Interfacing to graphics functions  52
gdiplus  5.8 : Interfacing to graphics functions  52
get-embedded-module function  116
get-embedded-module-data function  117
Grand Central Dispatch  5.7 : Block objects in C (foreign blocks)  50
graphics functions  5.8 : Interfacing to graphics functions  52

I
immediate FLI types  2 : FLI Types  17
incf-pointer function  118
install-embedded-module function  119
*install-embedded-module-delay-delete* variable  120
:int16 FLI type descriptor  170
:int32 type foreign-typed-aref  112
:int32 FLI type descriptor  170
:int64 type foreign-typed-aref  112
:int64 FLI type descriptor  170
:int8 FLI type descriptor  170
:int FLI type descriptor  171
:intmax FLI type descriptor  170
:intptr FLI type descriptor  170
iOS  4.2.2 : ARM 32-bit calling conventions  35

L
languages supported  1 : Introduction to the FLI  12
:lifetime
  argument to register-module register-module  134
Linux  4.2.2 : ARM 32-bit calling conventions  35

Lisp
calling from C  4.1 : Foreign callables and foreign functions  32
calling from C++  4.1 : Foreign callables and foreign functions  32
calling into C  1.1.1 : Defining the FLI function  12,  5.7.1 : Calling foreign code that receives a block as argument  50
calling into C++  1.1.1 : Defining the FLI function  12,  5.7.1 : Calling foreign code that receives a block as argument  50
calling into C with a block  5.7.1 : Calling foreign code that receives a block as argument  50
calling into C++ with a block  5.7.1 : Calling foreign code that receives a block as argument  50
:lisp-array FLI type descriptor  171
:lisp-double-float FLI type descriptor  173
:lisp-float FLI type descriptor  173
:lisp-simple-1d-array FLI type descriptor  174
Index

: lisp-single-float  FLI type descriptor  175
*locale-external-formats*  variable  121
: long  FLI type descriptor  175
: long-long  FLI type descriptor  176
lpCTSTR  FLI type descriptor  5.2.8 : Win32 API functions that handle strings  45
lpcwchar  FLI type descriptor  5.2.8 : Win32 API functions that handle strings  45
lptstr  FLI type descriptor  5.2.8 : Win32 API functions that handle strings  46

M

macros

  define-c-enum  67
  define-c-struct  69
  define-c-typedef  71
  define-c-union  72
  define-foreign-block-callable-type  74
  define-foreign-block-invoker  75
  define-foreign-callable  76  4.1 : Foreign callables and foreign functions  32, 4.1.1 : Strings and foreign callables  34
  define-foreign-converter  79
  define-foreign-forward-reference-type  81
  define-foreign-funcallable  82
  define-foreign-function  83  4.1 : Foreign callables and foreign functions  32
  define-foreign-pointer  87  3.1.1 : Creating pointers  25
  define-foreign-type  88  2 : FLI Types  17
  define-foreign-variable  89
  define-opaque-pointer  92

defsystem  5.6 : Incorporating a foreign module into a LispWorks image  50
  with-coerced-pointer  147
  with-dynamic-foreign-objects  148
  with-dynamic-lisp-array-pointer  151
  with-foreign-block  152
  with-foreign-slots  153
  with-foreign-string  154
  with-integer-bytes  156
  with-local-foreign-block  156

make-integer-from-bytes  function  122
make-pointer  function  122
malloc  function  57

memory allocation  1.4 : An example of dynamic memory allocation  16, 3.1.3 : Allocation of FLI memory  26

module-unresolved-symbols  function  124
Index

N
New in LispWorks 7.0
64-bit integer FLI types supported in 32-bit LispWorks define-foreign-callable 78, dereference 94, foreign-slot-value 111
foreign-function-pointer function 104
hard-float and soft-float calling conventions for ARM platforms 4.2.2: ARM 32-bit calling conventions 35
released-foreign-block-pointer FLI type descriptor 181
replace-foreign-array function 138
Store a foreign module in a Lisp image with defsystem member option :embedded-module 5.6: Incorporating a foreign module into a LispWorks image 50
New in LispWorks 7.1
ARM 64-bit platform 4.2.3: ARM 64-bit calling conventions 36
fastcall calling convention for 32-bit x86 platforms 4.2.4: Fastcall on 32-bit x86 platforms 36
iOS calling convention for ARM 32-bit platforms 4.2.2: ARM 32-bit calling conventions 35
specifying variadic foreign functions define-foreign-function 86
*use-sse2-for-ext-vector-type* variable 145
:variadic-num-of-fixed keyword define-foreign-function 86
vector types 2.2.4: Vector types 20
New in LispWorks 8.0
delay-delete argument to install-embedded-module install-embedded-module 119
:double-complex FLI type descriptor 163
:float-complex FLI type descriptor 167
*install-embedded-module-delay-delete* variable 120
valid-foreign-type-p function 146
*null-pointer* variable 125
null-pointer-p function 125
null pointers 3.2: Pointer testing functions 26
O
:one-of FLI type descriptor 177
P
:pointer FLI type descriptor 178
pointer-address function 126
pointer-element-size function 127
pointer-element-type function 128
pointer-element-type-p function 129
pointer-eq function 130
pointerp function 131
pointer-pointer-type function 132
pointers 3: FLI Pointers 25
coercing 3.3: Pointer dereferencing and coercing 27
copying 3.1.2: Copying pointers 26
creating 3.1.1: Creating pointers 25
dereferencing 3.3: Pointer dereferencing and coercing 27
dynamically allocating 3.4: An example of dynamic pointer allocation 28
null pointers 3.2: Pointer testing functions 26
test functions for 3.2: Pointer testing functions 26
*preprocessor* variable 198 9.1.1: Requirements 196
*preprocessor-format-string* variable 199
*preprocessor-include-path* variable 199
*preprocessor-options* variable 200
print-collected-template-info function 133
print-foreign-modules function 133
process-foreign-file function 200
:ptr FLI type descriptor 178
:ptrdiff_t FLI type descriptor 178
R
:reference FLI type descriptor 179
:reference-pass FLI type descriptor 180
:reference-return FLI type descriptor 181
register-module function 134
released-foreign-block-pointer FLI type descriptor 181
replace-foreign-array function 138
replace-foreign-object function 140
S
Self-contained examples
foreign blocks 6.1: Foreign block examples 53
miscellaneous examples 6.2: Miscellaneous examples 53
set-locale function 141
set-locale-encodings function 142
setup-embedded-module function 143
:short FLI type descriptor 182
:signed FLI type descriptor 183
size-of function 144
:size-t FLI type descriptor 184
:size-t FLI type descriptor 184
start-collecting-template-info function 145
str FLI type descriptor 5.2.8: Win32 API functions that handle strings 45
strings
modifying in C 5.2.4: Modifying a string in a C function 40
passing to C 5.1: Passing a string to a Windows function 37, 5.2.2: Passing a string 38
returning from C 5.2.3: Returning a string via a buffer 39
Index

:struct FLI type descriptor 185

T
templates, FLI print-collected-template-info 133, start-collecting-template-info 145
:time-t FLI type descriptor 186
tstr FLI type descriptor 5.2.8: Win32 API functions that handle strings 46
type constructors 2: FLI Types 17
types
  int32 foreign-typed-aref 112
  int64 foreign-typed-aref 112

U
:uint16 FLI type descriptor 186
:uint32 FLI type descriptor 186
:uint64 FLI type descriptor 186
:uint8 FLI type descriptor 186
:uintmax FLI type descriptor 186
:uintptr FLI type descriptor 186
:union FLI type descriptor 187
:unsigned FLI type descriptor 188
*use-sse2-for-ext-vector-type* variable 145

V
valid-foreign-type-p function 146
variables
  *install-embedded-module-delay-delete* 120
  *locale-external-formats* 121
  *null-pointer* 125
  *preprocessor* 198 9.1.1: Requirements 196
  *preprocessor-format-string* 199
  *preprocessor-include-path* 199
  *preprocessor-options* 200
  *use-sse2-for-ext-vector-type* 145
vector-char16 FLI type descriptor 189
vector-char2 FLI type descriptor 189
vector-char3 FLI type descriptor 189
vector-char32 FLI type descriptor 189
vector-char4 FLI type descriptor 189
vector-char8 FLI type descriptor 189
vector-double2 FLI type descriptor 190
vector-double3 FLI type descriptor 190
vector-double4 FLI type descriptor 191
### Index

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector-doublen</code></td>
<td>FLI type descriptor</td>
<td>191</td>
</tr>
<tr>
<td><code>vector-floaten</code></td>
<td>FLI type descriptor</td>
<td>190</td>
</tr>
<tr>
<td><code>vector-floatn</code></td>
<td>FLI type descriptor</td>
<td>190</td>
</tr>
<tr>
<td><code>vector-intn</code></td>
<td>FLI type descriptor</td>
<td>190</td>
</tr>
<tr>
<td><code>vector-longn</code></td>
<td>FLI type descriptor</td>
<td>190</td>
</tr>
<tr>
<td><code>vector-ucharn</code></td>
<td>FLI type descriptor</td>
<td>189</td>
</tr>
<tr>
<td><code>vector-unsignedn</code></td>
<td>FLI type descriptor</td>
<td>190</td>
</tr>
</tbody>
</table>

218
vector-ushort3  FLI type descriptor  190
vector-ushort32 FLI type descriptor  190
vector-ushort4  FLI type descriptor  190
vector-ushort8  FLI type descriptor  190
:void     FLI type descriptor  193
:volatile FLI type descriptor  193

W
:wchar-t   FLI type descriptor  194
with-coerced-pointer macro  147
with-dynamic-foreign-objects macro  148
with-dynamic-lisp-array-pointer macro  151
with-foreign-block macro  152
with-foreign-slots macro  153
with-foreign-string macro  154
with-integer-bytes macro  156
with-local-foreign-block macro  156
:wrapper  FLI type descriptor  194
wstr     FLI type descriptor  5.2.8 : Win32 API functions that handle strings  45