
Foreign Language Interface User Guide and Reference Manual

Version 7.0



Copyright and Trademarks

LispWorks Foreign Language Interface User Guide and Reference Manual

Version 7.0

February 2015

Copyright © 2015 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 code for `walker.lisp` and `compute-combination-points` is excerpted with permission from PCL, Copyright © 1985, 1986, 1987, 1988 Xerox Corporation.

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.

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.

Address

LispWorks Ltd
St. John's Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone

From North America: 877 759 8839
(toll-free)
From elsewhere: +44 1223 421860

Fax

From North America: 617 812 8283
From elsewhere: +44 870 2206189

www.lispworks.com

Contents

Preface ix

1 Introduction to the FLI 1

An example of interfacing to a foreign function 1

Using the FLI to get the cursor position 4

Using the FLI to set the cursor position 7

An example of dynamic memory allocation 7

Summary 8

2 FLI Types 11

Immediate types 12

Aggregate types 14

Parameterized types 16

Encapsulated types 16

The void type 18

Summary 19

3 FLI Pointers 21

Creating and copying pointers 21

Pointer testing functions 23

Pointer dereferencing and coercing 24

An example of dynamic pointer allocation 27

More examples of allocation and pointer allocation 28

Summary 30

4	Defining foreign functions and callables	31
	Foreign callables and foreign functions	31
	ARM hard-float and soft-float	35
5	Advanced Uses of the FLI	39
	Passing a string to a Windows function	39
	Modifying, passing and returning strings	41
	Lisp integers	57
	Defining new types	57
	Using DLLs within the LispWorks FLI	58
	Incorporating a foreign module into a LispWorks image	60
	Block objects in C (foreign blocks)	60
	Interfacing to graphics functions	64
	Summary	64
6	Self-contained examples	65
	Foreign block examples	65
	Miscellaneous examples	66
7	Function and Macro Reference	67
	align-of	67
	alloca	68
	allocate-dynamic-foreign-object	68
	allocate-foreign-block	70
	allocate-foreign-object	71
	cast-integer	74
	connected-module-pathname	74
	convert-from-foreign-string	75
	convert-integer-to-dynamic-foreign-object	77
	convert-to-foreign-string	77
	convert-to-dynamic-foreign-string	79
	copy-pointer	81
	decf-pointer	82
	define-c-enum	83
	define-c-struct	85
	define-c-typedef	88
	define-c-union	89
	define-foreign-block-callable-type	91

- define-foreign-block-invoker 92
- define-foreign-callable 93
- define-foreign-converter 98
- define-foreign-forward-reference-type 102
- define-foreign-funcallable 102
- define-foreign-function 103
- define-foreign-pointer 109
- define-foreign-type 110
- define-foreign-variable 112
- define-opaque-pointer 116
- dereference 117
- disconnect-module 120
- enum-symbol-value 121
- enum-value-symbol 121
- enum-values 121
- enum-symbols 121
- enum-symbol-value-pairs 121
- fill-foreign-object 123
- foreign-aref 124
- foreign-array-dimensions 125
- foreign-array-element-type 126
- foreign-array-pointer 127
- foreign-block-copy 128
- foreign-block-release 129
- foreign-function-pointer 130
- foreign-slot-names 132
- foreign-slot-offset 133
- foreign-slot-pointer 134
- foreign-slot-type 135
- foreign-slot-value 137
- foreign-type-equal-p 140
- foreign-type-error 141
- foreign-typed-aref 141
- free 142
- free-foreign-block 143
- free-foreign-object 144
- get-embedded-module 145
- get-embedded-module-data 146
- incf-pointer 148
- install-embedded-module 149

- *locale-external-formats* 150
- make-integer-from-bytes 151
- make-pointer 151
- malloc 153
- module-unresolved-symbols 154
- *null-pointer* 154
- null-pointer-p 155
- pointer-address 156
- pointer-element-size 157
- pointer-element-type 157
- pointer-element-type-p 159
- pointer-eq 159
- pointer-pointer-type 160
- pointerp 161
- print-collected-template-info 162
- print-foreign-modules 163
- register-module 163
- replace-foreign-array 169
- replace-foreign-object 173
- set-locale 174
- set-locale-encodings 175
- setup-embedded-module 176
- size-of 177
- start-collecting-template-info 178
- with-coerced-pointer 178
- with-dynamic-foreign-objects 180
- with-dynamic-lisp-array-pointer 184
- with-foreign-block 185
- with-foreign-slots 187
- with-foreign-string 188
- with-integer-bytes 190
- with-local-foreign-block 191

8 Type Reference 193

- :boolean 193
- :byte 194
- :c-array 194
- :char 196
- :const 197

:double 197
:ef-mb-string 198
:ef-wc-string 199
:enum 200
:enumeration 201
:fixnum 201
:float 201
:foreign-array 202
foreign-block-pointer 203
:function 204
:int 205
:int8 205
:int16 205
:int32 205
:int64 205
:intmax 205
:intptr 205
:lisp-array 206
:lisp-double-float 208
:lisp-float 209
:lisp-simple-1d-array 209
:lisp-single-float 210
:long 210
:long-long 211
:one-of 212
:pointer 213
:ptr 213
:ptrdiff-t 214
:reference 214
:reference-pass 216
:reference-return 216
released-foreign-block-pointer 217
:short 217
:signed 218
:size-t 219
:ssize-t 219
:struct 220
:time-t 221
:uint8 222
:uint16 222

:uint32 222
:uint64 222
:uintmax 222
:uintptr 222
:union 223
:unsigned 224
:void 225
:volatile 226
:wchar-t 226
:wrapper 226

9 The Foreign Parser 229

Introduction 229
Loading the Foreign Parser 230
Using the Foreign Parser 230
Using the LispWorks Editor 232
Foreign Parser Reference 233
preprocessor 233
preprocessor-format-string 233
preprocessor-include-path 234
preprocessor-options 234
process-foreign-file 234

Glossary 239

Index 243

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, Chapter 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-blocks")
```

These examples are Lisp source files in your LispWorks installation under `lib/7-0-0-0/examples/`. You can simply evaluate the given form to view the example source file.

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 “Using C++ DLLs” on page 59. 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`:

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

The FLI interface is as follows:

```
(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).

LispWorks for AIX can load shared libraries such as `/usr/lib/libz.a`.

LispWorks for UNIX can either load object files (usually suffixed with `".o"`) directly into the Lisp image, extract any required object files from the available archive libraries (usually suffixed with `".a"`), or load in shared libraries (usually suffixed with `".so"`).

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 Macintosh:

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

On AIX:

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

Note: LispWorks for UNIX also provides the loader function `link-load:read-foreign-modules` familiar to users of LispWorks 4.3 and earlier. However, this is now deprecated in favor of `register-module`.

Note: It is also possible to embed a DLL in the Lisp image. See “Incorporating a foreign module into a LispWorks image” on page 60.

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:

```
BOOL GetCursorPos( LPPOINT )
```

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

```
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.

```
(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 a `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.

```
(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`.

```
(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.

```
(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.

```
(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.

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

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

```
(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.


```
(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`.

```
(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:

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

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

```
(defun test-cursor ()
  (dotimes (x 10)
    (dotimes (d 300)
      (let ((r (/ (+ d (* 300 x)) 10.0)))
        (set-cursor-position
          (+ 300 (floor (* r (cos (/ (* d pi) 150.0))))))
          (+ 300 (floor (* r (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 Chapter 3, “FLI Pointers”. To free the memory associated with the foreign object requires the use of the function `free-foreign-object`.

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

There are other methods for dealing with the question of memory management. The following example defines a Lisp function that returns the *x* and *y* coordinates of the cursor without permanently tying up memory for structures that are only used once.

```
(defun current-cursor-position ()
  (fli:with-dynamic-foreign-objects ()
    (let ((lppoint (fli:allocate-dynamic-foreign-object
                  :pointer-type 'lppoint)))
      (if (get-cursor-position lppoint)
          (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.

The immediate types of the FLI are `:boolean`, `:byte`, `:char`, `:const`, `:double`, `:enum`, `:float`, `:int`, `:lisp-double-float`, `:lisp-float`, `:lisp-single-float`, `:long`, `:pointer`, `:short`, `:signed` and `:unsigned`. For details on each immediate type, see the relevant reference entry.

2.1.1 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 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 Character types

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

2.1.4 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`.

2.1.5 Pointer types

Pointers are discussed in detail in Chapter 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.

2.2.1 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 Chapter 8.

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`:

```
(fli:define-c-struct tagpoint
  (x :long)
  (y :long)
  (visible (:boolean :byte)))
```

This structure would interface with the following C structure:

```
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`.

```
(setq point (fli:allocate-foreign-object :type 'tagpoint))
(setq names (fli:foreign-slot-names point))
```

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

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

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.

2.3 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:

```
(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`.

`:wrapper`

Type

Allows the specification of automatic conversion functions between Lisp and an instance of a FLI type.

```
: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:

```
(fli:define-foreign-type lisp-object-wrapper ()
  "A mechanism for passing a Lisp object handle to C.
  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 “Allocating a pointer to a pointer to a void” on page 28.

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

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

will return something similar to the following:

```
#<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`.

```
(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`.

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

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

```
(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 “Creating pointers” on page 21. 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 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`.

```
(fli: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`.

```
(fli:pointer-element-size point1)
```

```
(fli: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 Chapter 2, “FLI Types”, and the use of the `dereference` function with aggregate types is discussed further in Chapter 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 `fli:dereference`.

```
(setq point5 (fli:allocate-foreign-object :type :int))
(setf (fli:dereference point5) 12)
(fli: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 though it was set as an `:int`. Because the value at `point5` is not 0, it is returned as `t`.

```
(fli: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:

```
(setq pointer-a (fli: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`.

```
(setq pointer-b (fli:copy-pointer pointer-a :type :byte))
```

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-point-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:

```

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:

```

(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:

```

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

(func-handle-init handle)

```

but do not forget to also free the pointer:

```

(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:

```

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

```

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

```
(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:

```
(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

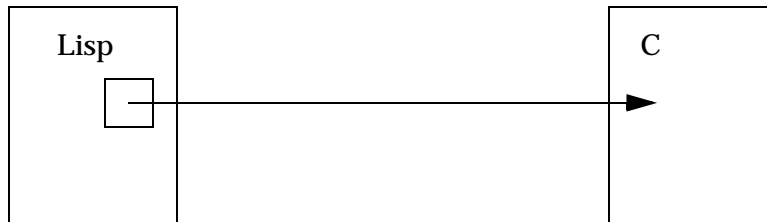
The two main macros for interfacing LispWorks with a foreign language are `define-foreign-callable` which defines Lisp functions that can be called from the foreign language, and `define-foreign-function` which defines a short linking function that can call functions in a foreign language.

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

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

Figure 4.1 is an illustration of `set-cursor-position`, represented by a square, calling the C code which constitutes `SetCursorPos`.

Figure 4.1 A FLI foreign function calling some C code.



The next diagram, Figure 4.2, 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.

Figure 4.2 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`.

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

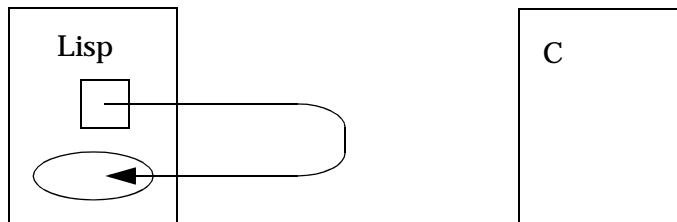
Supposing you had the above foreign callable defined in a real application, you would use

```
(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`.

Figure 4.3 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”.

Figure 4.3 A FLI foreign function calling a callable function.



For more information on calling foreign code see `define-foreign-function`, page 103.

For more information on defining foreign callable functions see “Strings and foreign callables” on page 34 and `define-foreign-callable`, page 93.

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 *LispWorks 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:

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

you would define in Lisp:

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

and call

```
(evalx "(+ 2 3)")
=>
"5"
```

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

```
{
  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:

```

(fli:define-foreign-callable
  ("evalx" :calling-convention :stdcall)
  ((form (:reference :ef-mb-string
                  :lisp-to-foreign-p nil
                  :foreign-to-lisp-p t))
   (result (:reference (:ef-mb-string :limit 1000)
                      :lisp-to-foreign-p t
                      :foreign-to-lisp-p nil))))
  (multiple-value-bind (res err)
    (ignore-errors (read-from-string form)))
  (setq result
    (if (not (fixnump 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 ARM hard-float and soft-float

ARM systems come with two calling conventions: hard-float and soft-float. These calling conventions are binary incompatible, and operating systems are either hard-float or soft-float. Currently, Android and iOS are both soft-float, while Linux distributions are in the process of migration to hard-float: old distributions are soft-float and new distributions are hard-float.

When LispWorks compiles foreign interface (callers or callables), by default it generates "bi-compatible" code that can interface with either hard-float and soft-float foreign code. At run-time, the code checks an internal flag and uses the appropriate convention. The internal flag is set to the correct value on startup. The bi-compatible code is needed only for functions that pass or return `float` values (including small structures containing `float` values). Functions that pass and return only pointers and types like `int` do not need it.

The bi-compatible code adds some overhead, both in size and speed, but this overhead is small and normally would not make a significant difference. You can tell the system to compile for only one calling convention, by passing the keyword argument `:calling-convention` with the value `:soft-float` or `:hard-float` to the definers (`define-foreign-function`, `define-foreign-callable` and `define-foreign-funcallable`). This will result in slightly smaller code which is slightly faster, but will run only on one calling convention.

A special case is variadic functions (for example `printf`), because these always uses the soft-float calling convention, even in hard-float binaries. Currently LispWorks does not have a way of recognizing a variadic function. This means that if you define a caller to such function (by `define-foreign-function` or use it in a call to `define-foreign-callable`) which take `floats`, it will do the wrong thing on a hard-float system. Thus when you define such call you need to tell LispWorks to use soft-float calling convention. For example, defining a call to `printf` which, apart from the string, takes a single argument which is a `double`:

```
(fli:define-foreign-function (printf-with-a-double "printf")
  ((string (:reference-pass :ef-mb-string))
   (double-arg :double-float))
  :calling-convention :soft-float)
```

Because of the bi-compatible code, LispWorks binaries (fasl files) are compatible with both hard-float and soft-float. The Lisp heap is also compatible with both conventions. However, the executable (including LispWorks as a dynamic library) has also a small C program which starts it (the "xstarter"), and that is either hard-float or soft-float.

Therefore a LispWorks executable can run only on one calling convention, but the code that it compiles can run on both (strictly speaking, the executable actually runs, but does not work properly).

In particular, that means that it is possible to compile and build for Android and iOS on either soft-float or hard-float systems, because the delivery process uses the appropriate "xstarter" for the target operating system.

The fact that operating systems are of one calling convention is not necessarily permanent, because in principle the same operating system, and even the same process, can use different calling conventions with different functions.

For example, Android now allows compiling some code using hard-float. Thus in the future there may be cases when some libraries will use a different calling convention from the operating system. In this case the interface to this library will have to be compiled explicitly with the correct calling convention.

4 *Defining foreign functions and callables*

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))
        (set-win-text active-window new-ptr))))))
```

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`, page 188, for more details on the use of foreign strings.

5.2 Modifying, 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 temporary foreign object from the Lisp String, pass that to C, and then convert the data in that foreign object back to a Lisp object when C returns.

5.2.2 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:

```
#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);
}
```

Linux/Unix/Macintosh version:

```
#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)

(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))))

```

5.2.3 Passing a constant 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:

```

#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;
}

```

Linux/Unix/Macintosh version:

```

#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`:

```

(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

```

5.2.4 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:

```
#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;
}
```

Linux/Unix/Macintosh version:

```
#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"

```

5.2.5 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 (1+ 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)
    array))

(fli:define-foreign-function (%foo foo)
  ((string-array (:pointer (:pointer :char)))))

(defun foo (strings)
  (fli:with-dynamic-foreign-objects () ; provide a dynamic scope
    (%foo (convert-to-dynamic-foreign-array strings))))
```

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):

```
(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:

```
(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:

```
#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;
}
```

Linux/Unix/Macintosh version:

```

#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:

```

(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)
=>
"öãö"

(random-string2 6)
=>
"óãøççâ"

```

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.

```
(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 Unix/Linux 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)
=>
"öëñçëö"
```

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 Unix/Linux/Mac OS X, 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:

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

```

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:

```
(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 99
```

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:

```
(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:

```
(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:

```
(fli:define-foreign-function (create-file "CreateFile" :dbcs)
  ((lpFileName win32:lpcwstr) ...))
```

This will cause it to use the Unicode `w` function implicitly in supported versions of Windows. (In some older operating systems such as Windows ME, this mechanism would implicitly use the ANSI `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 `a` or `w` in the function name. The foreign types `win32:tstr`, `win32:lpcwstr` and `win32:lptstr` 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 *LispWorks User Guide and Reference Manual*.

5.2.9 Mapping Nil to a Null Pointer

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

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

Windows version:


```

#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]);
        }
    }
}

```

Linux/Unix/Macintosh version:

```

#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:

```

(fli:define-foreign-function (strcap "strcap" :source)
  ((string (:reference :ef-mb-string)))
  :language
  :c
  :calling-convention
  :cdecl)

```

```
(strcap "abC")
=>
"Abc"
```

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

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

```
(fli:define-foreign-function (strcap "strcap" :source)
  ((string (:reference :ef-mb-string :allow-null t)))
  :language
  :c
  :calling-convention
  :cdecl)

(strcap 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:

```
(fli:define-foreign-function (strcap "strcap" :source)
  ((string :pointer))
  :language
  :c
  :calling-convention
  :cdecl)

(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)))

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

(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 a 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 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 Chapter 2. 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`, page 110.

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 3 follows. Let us assume you have the following C code in a file called `example.c`.

```
int _stdcall MultiplyMain(void *hinstDll, unsigned long
                        dwReason, void *reserved)
{
    return(1);
}

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 `lcc`.

```
lcc -O -g2 example.c
lclnk.exe -dll -entry MultiplyMain example.obj
example.def -subsystem
windows
```

You now need to create a `multiply.def` file that contains the following line

```
exports multiply=multiply
```

to export the function `multiply` as the symbol `multiply`. If you only include the line “`exports multiply`” then the name of the external symbol is likely to

be “`_multiply`” or “`_multiply@8`” depending on whether the function is compiled as `__cdecl` or `__stdcall`. The addition of the “`= multiply`” overrides the internal function name with the new name.

If you run Windows then you can view the list of exported symbols from a given DLL by selecting the DLL from an explorer, then right clicking on it and selecting QuickView. This brings up some text about the DLL.

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

```
(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.

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

```
(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 “Using C DLLs” on page 58.

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 Mac OS X, and shared object or shared library on other platforms. See “Loading foreign code” on page 3 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 runtime 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 runtime 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`.

Another way to “intern” the module is to define a `lw:defsystem` system containing a C source file member with the `:embedded-module` keyword. When the system is loaded, the value associated with `:embedded-module` is used to create the embedded module. You would then call `install-embedded-module` at runtime 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 encapsulates 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 `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 `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 Mac OS X 10.6 and later (see Apple documentation). There is a simple example in:

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

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 `define-foreign-block-callable-type`. This "type" corresponds to the "signature" in C.

2. At run time, generate the block, for example by calling `allocate-foreign-block` with the "type". Alternatively use one of the macros `with-foreign-block` and `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 `allocate-foreign-block` are released when appropriate by `free-foreign-block`.

Foreign block pointers created by `allocate-foreign-block` are of type `foreign-block-pointer` and print with `"lisp-foreign-block-pointer"`.

For examples see:

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

and

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

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

```
(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 `qsort_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 `qsort_b` returned, because the arguments to the block are based on the data (first argument to `qsort_b`), and `qsort_b` and its callees cannot be guaranteed that the data is still valid once `qsort_b` returned. On the other hand, we cannot be sure that the block is not invoked on another thread(s) before `qsort_b` returns. Currently it is probably always called in the same thread where `qsort_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 `gp:ensure-gdiplus`. See the *C API User Guide and Reference Manual* for a detailed explanation.

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 “Block objects in C (foreign blocks)” on page 60.

These examples apply to LispWorks for Macintosh only:

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

6 *Self-contained examples*

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

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

6.2 **Miscellaneous examples**

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

7

Function and Macro Reference

align-of

Function

Summary	Returns the alignment in bytes of a foreign type.	
Package	<code>ffi</code>	
Signature	<code>align-of <i>type-name</i> => <i>alignment</i></code>	
Arguments	<i>type-name</i>	A foreign type whose alignment is to be determined.
Values	<i>alignment</i>	The alignment of the foreign type <i>type-name</i> in bytes.
Description	The function <code>align-of</code> returns the alignment in bytes of the foreign language type named by <i>type-name</i> .	
Example	The following example shows types with various alignments.	

```
(fli:align-of :char)
=>
1

(fli:align-of :int)
=>
4

(fli:align-of :double)
=>
8

(fli:align-of :pointer)
=>
4
```

See also `allocate-foreign-object`
`free-foreign-object`

alloca *Function*

Summary `allocate-dynamic-foreign-object`.

Package `fli`

Signature `alloca &key type pointer-type initial-element initial-contents nelems => pointer`

Description The function `alloca` is a synonym for `allocate-dynamic-foreign-object`.

See also `allocate-dynamic-foreign-object`

allocate-dynamic-foreign-object *Function*

Summary Allocates memory for an instance of a foreign object within the scope of a `with-dynamic-foreign-objects` macro.

Package	<code>fli</code>	
Signature	<code>allocate-dynamic-foreign-object &key <i>type</i> <i>pointer-type</i> <i>initial-element</i> <i>initial-contents</i> <i>fill</i> <i>nelems</i> <i>size-slot</i> => <i>pointer</i></code>	
Arguments	<i>type</i>	A FLI type specifying the type of the object to be allocated. If <i>type</i> is supplied, <i>pointer-type</i> must not be supplied.
	<i>pointer-type</i>	A FLI pointer type specifying the type of the pointer object to be allocated. If <i>pointer-type</i> is supplied, <i>type</i> must not be supplied.
	<i>initial-element</i>	A keyword setting the initial value of every element in the newly allocated object to <i>initial-element</i> .
	<i>initial-contents</i>	A list of forms which initialize the contents of each element in the newly allocated object.
	<i>fill</i>	An integer between 0 to 255.
	<i>nelems</i>	An integer specifying how many copies of the object should be allocated. The default value is 1.
	<i>size-slot</i>	A symbol naming a slot in the object.
Values	<i>pointer</i>	A pointer to the specified <i>type</i> or <i>pointer-type</i> .
Description	<p>The function <code>allocate-dynamic-foreign-object</code> allocates memory for a new instance of an object of type <i>type</i> or an instance of a pointer object of type <i>pointer-type</i> within the scope of the body of the macro <code>with-dynamic-foreign-objects</code>.</p> <p>The object is initialized as if by <code>allocate-foreign-object</code>.</p> <p>Once this macro has executed, the memory allocated using <code>allocate-dynamic-foreign-object</code> is therefore automatically freed for other uses.</p>	

Example A full example using `with-dynamic-foreign-objects` and `allocate-dynamic-foreign-object` is given in “An example of dynamic memory allocation” on page 7.

See also `allocate-foreign-object`
`with-dynamic-foreign-objects`
 “An example of dynamic memory allocation” on page 7
 “More examples of allocation and pointer allocation” on page 28
 “Modifying a string in a C function” on page 41

allocate-foreign-block

Function

Summary Allocates a foreign block, in LispWorks for Macintosh.

Package `fl1`

Signature `allocate-foreign-block` *type function* &rest *extra-arguments*
 => *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 “Scope of invocation” on page 62 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`
 "Block objects in C (foreign blocks)" on page 60

`allocate-foreign-object` *Function*

Summary Allocates memory for an instance of a foreign object.

Package `fli`

Signature `allocate-foreign-object &key type pointer-type initial-element initial-contents fill nelems size-slot => 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.

	<i>pointer-type</i>	A FLI pointer type specifying the type of the pointer object to be allocated. If <i>pointer-type</i> is supplied, <i>type</i> must not be supplied.
	<i>initial-element</i>	A keyword setting the initial value of every element in the newly allocated object to <i>initial-element</i> .
	<i>initial-contents</i>	A list of forms which initialize the contents of each element in the newly allocated object.
	<i>fill</i>	An integer between 0 to 255.
	<i>nelems</i>	An integer specifying how many copies of the object should be allocated. The default value is 1.
	<i>size-slot</i>	A symbol naming a slot in the object.
	<i>allocation</i>	A keyword, either <code>:dynamic</code> or <code>:static</code> .
Values	<i>pointer</i>	A pointer to the specified <i>type</i> or <i>pointer-type</i> .
Description		<p>The function <code>allocate-foreign-object</code> allocates memory for a new instance of an object of type <i>type</i> or an instance of a pointer object of type <i>pointer-type</i>.</p> <p>If <i>allocation</i> is <code>:static</code> then memory is allocated in the C heap and must be explicitly freed using <code>free-foreign-object</code> once the object is no longer needed.</p> <p>If <i>allocation</i> is <code>:dynamic</code>, then <code>allocate-foreign-object</code> allocates memory for the object and pointer within the scope of the body of <code>with-dynamic-foreign-objects</code>. This is equivalent to using <code>allocate-dynamic-foreign-object</code>.</p> <p>The default value of <i>allocation</i> is <code>:static</code>.</p> <p>An integer value of <i>fill</i> initializes all the bytes of the object. If <i>fill</i> is not supplied, the object is not initialized unless <i>initial-element</i> or <i>initial-contents</i> is passed.</p>

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).

A supplied value of *size-slot* applies if the type is a struct or union type. The slot *size-slot* is set to the size of the object in bytes. This occurs after the *fill*, *initial-element* and *initial-contents* arguments are processed. If *nelems* is greater than 1, then the slot *size-slot* is initialized in each element. If *size-slot* is not supplied, then no such setting occurs.

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`.

Example

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.

```
(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`
“FLI Pointers” on page 21

cast-integer*Function*

Summary	Casts an integer to a given type.	
Package	<code>fli</code>	
Signature	<code>cast-integer <i>integer type</i> => <i>result</i></code>	
Arguments	<i>integer</i>	A Lisp integer.
	<i>type</i>	A foreign type.
Values	<i>result</i>	A Lisp integer.
Description	The function <code>cast-integer</code> casts the integer <i>integer</i> to the foreign type <i>type</i> . <i>type</i> must be a FLI integer type, either primitive or derived.	
Example	<pre>(format nil "~B" (fli:cast-integer -1 '(:unsigned :int))) => "111"</pre>	
See also	<code>:signed</code>	
	<code>:unsigned</code>	

connected-module-pathname*Function*

Summary	Returns the real pathname of a connected module.	
Package	<code>fli</code>	
Signature	<code>connected-module-pathname <i>name</i> => <i>pathname</i></code>	
Arguments	<i>name</i>	A string or symbol.
Values	<i>pathname</i>	A pathname or <code>nil</code> .

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`.

Example

```
(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

Function

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	<i>pointer</i>	A pointer to a foreign string.
	<i>external-format</i>	An external format specification.
	<i>length</i>	The length of the string to convert.
	<i>null-terminated-p</i>	If τ , it is assumed the string terminates with a null character. The default value for <i>null-terminated-p</i> is τ .
	<i>allow-null</i>	A boolean. The default is false.
Values	<i>string</i>	A Lisp string, or <code>nil</code> .
Description		<p>The function <code>convert-from-foreign-string</code>, 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 <i>external-format</i>.</p> <p>The <i>external-format</i> argument is interpreted as by <code>with-foreign-string</code>. The names of available external formats are listed in the section "External formats" in the <i>LispWorks User Guide and Reference Manual</i>.</p> <p>Either <i>length</i> or <i>null-terminated-p</i> must be non-<code>nil</code>. If <i>null-terminated-p</i> is true and <i>length</i> is not specified, it is assumed that the foreign string to be converted is terminated with a null character.</p> <p>If <i>allow-null</i> is true, then if a null pointer <i>pointer</i> is passed, <code>nil</code> is returned.</p>
See also		<p><code>convert-to-foreign-string</code> <code>set-locale</code> <code>set-locale-encodings</code> <code>with-foreign-string</code> Section "External formats" in the <i>LispWorks User Guide and Reference Manual</i> "Modifying a string in a C function" on page 41 "Mapping Nil to a Null Pointer" on page 54</p>

convert-integer-to-dynamic-foreign-object

Function

Summary	Converts a Lisp integer to foreign bytes.	
Package	fli	
Signature	<code>convert-integer-to-dynamic-foreign-object</code> <i>integer</i> => <i>pointer</i> , <i>length</i>	
Arguments	<i>integer</i>	An integer.
Values	<i>pointer</i>	A foreign pointer.
	<i>length</i>	An integer.
Description	The function <code>convert-integer-to-dynamic-foreign-object</code> makes a dynamic foreign object containing the bytes of <i>integer</i> and returns <i>pointer</i> pointing to the first byte of that object and <i>length</i> 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 <i>integer</i> by calling <code>make-integer-from-bytes</code> .	
See also	“Lisp integers” on page 57 <code>with-integer-bytes</code> <code>make-integer-from-bytes</code>	

convert-to-foreign-string

Function

Summary	Converts a Lisp string to a foreign string.	
Package	fli	
Signature	<code>convert-to-foreign-string</code> <i>string</i> &key <i>external-format</i> <i>null-terminated-p</i> <i>allow-null</i> <i>into</i> <i>limit</i> <i>allocation</i> => <i>pointer</i>	

	<code>convert-to-foreign-string</code> <i>string</i> &key <i>external-format</i> <i>null-terminated-p</i> <i>allow-null</i> <i>into</i> <i>limit</i> <i>allocation</i> => <i>pointer</i> , <i>length</i> , <i>byte-count</i>	
Package	<code>fli</code>	
Arguments	<i>string</i>	A Lisp string.
	<i>external-format</i>	An external format specification.
	<i>null-terminated-p</i>	If <code>t</code> , the foreign string terminates with a null character. The default value is <code>t</code> .
	<i>allow-null</i>	A boolean. The default is <code>nil</code> .
	<i>into</i>	A foreign array, a foreign pointer or <code>nil</code> . The default is <code>nil</code> .
	<i>limit</i>	A non-negative fixnum, or <code>nil</code> . The default is <code>nil</code> .
	<i>allocation</i>	A keyword, either <code>:dynamic</code> or <code>:static</code> . The default is <code>:static</code> .
Values	<i>pointer</i>	A FLI pointer to the foreign string.
	<i>length</i>	The length of the foreign string (including the terminating null character if there is one).
	<i>byte-count</i>	The number of bytes in the foreign string.
Description	<p>The function <code>convert-to-foreign-string</code> converts a Lisp string to a foreign string, and returns a pointer to the string.</p> <p>The <i>external-format</i> argument is interpreted as by <code>with-foreign-string</code>. The names of available external formats are listed in the section "External formats" in the <i>LispWorks User Guide and Reference Manual</i>.</p> <p>The <i>null-terminated-p</i> argument specifies whether the foreign string is terminated with a null character. It defaults to <code>t</code>.</p>	

If *allow-null* is non-`nil`, then if *string* is `nil` a null pointer *pointer* is returned.

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*.

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

If *allocation* is `:dynamic`, then `convert-to-foreign-string` allocates memory for the string and pointer within the scope of the body of `with-dynamic-foreign-objects` and additional values, *length* and *byte-count* are returned. This is equivalent to using `convert-to-dynamic-foreign-string`. Otherwise, the allocation is static.

See also

`convert-from-foreign-string`

`set-locale`

`set-locale-encodings`

`with-foreign-string`

Section "External formats" in the *LispWorks User Guide and Reference Manual*

"Calling a C function that takes an array of strings" on page 46

convert-to-dynamic-foreign-string

Function

Summary

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

Package	<code>fli</code>
Signature	<code>convert-to-dynamic-foreign-string</code> <i>string</i> &key <i>external-format</i> <i>null-terminated-p</i> <i>allow-null</i> => <i>pointer</i> , <i>length</i> , <i>byte-count</i>
Arguments	<p><i>string</i> A Lisp string.</p> <p><i>external-format</i> An external format specification.</p> <p><i>null-terminated-p</i> If <code>t</code>, the foreign string terminates with a null character. The default value is <code>t</code>.</p> <p><i>allow-null</i> A boolean. The default is <code>nil</code>.</p>
Values	<p><i>pointer</i> A FLI pointer to the foreign string.</p> <p><i>length</i> The length of the string (including the terminating null character if there is one).</p> <p><i>byte-count</i> The number of bytes in the converted string.</p>
Description	<p>The function <code>convert-to-dynamic-foreign-string</code> 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 <code>with-dynamic-foreign-objects</code> command.</p> <p>The <i>external-format</i> argument is interpreted as by <code>with-foreign-string</code>. The names of available external formats are listed in the section "External formats" in the <i>LispWorks User Guide and Reference Manual</i>.</p> <p>The <i>null-terminated-p</i> keyword specifies whether the foreign string is terminated with a null character. It defaults to <code>t</code>.</p> <p>If <i>allow-null</i> is non-<code>nil</code>, then if <i>string</i> is <code>nil</code> a null pointer <i>pointer</i> is returned.</p>
See also	<p><code>allocate-dynamic-foreign-object</code></p> <p><code>convert-from-foreign-string</code></p> <p><code>convert-to-foreign-string</code></p>

`set-locale`
`set-locale-encodings`
`with-dynamic-foreign-objects`
`with-foreign-string`
 Section "External formats" in the *LispWorks User Guide and Reference Manual*
 "Calling a C function that takes an array of strings" on page 46

copy-pointer

Function

Summary Returns a copy of a pointer object.

Package `fli`

Signature `copy-pointer pointer &key type pointer-type => copy`

Arguments

<i>pointer</i>	A pointer to copy.
<i>type</i>	The type of the object pointer to by <i>pointer</i> .
<i>pointer-type</i>	The type of <i>pointer</i> .

Values *copy* A copy of *pointer*.

Description The function `copy-pointer` returns a copy of *pointer*.

Example 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.

```

(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 object; a pointer to a `char`, whereas `point3` contains a copy of the foreign pointer object.

See also `make-pointer`
“Copying pointers” on page 22

decf-pointer

Function

Summary Decreases the address held by a pointer.

Package `fli`

Signature `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 the pointer. If *delta* is not given the address is decreased by the size of the type pointed to by the pointer. The address can be decreased by a multiple of the size of the type by specifying a value for *delta*.

The function `decf-pointer` is often used to move a pointer through an array of values.

Example 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.

```
(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)
  (fli:incf-pointer point1)
  (print (fli:dereference point1)))

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

See also `incf-pointer`
“An example of dynamic pointer allocation” on page 27

define-c-enum *Macro*

Summary Defines a FLI enumerator type specifier corresponding to the C `enum` type.

Package `fli`

Signature `define-c-enum` *name-and-options* &rest *enumerator-list* => *list*
name-and-options ::= *name* | (*name* *option**)
option ::= (:foreign-name *string*)
enumerator-list ::= {*entry-name* | (*entry-name* *entry-value*)}*

Arguments *name* A symbol naming the new enumeration type specifier
string A string specifying the foreign name of the type

	<i>enumerator-list</i>	A sequence of symbols, possibly with integer values, constituting the enumerator type
	<i>entry-name</i>	A symbol
	<i>entry-value</i>	An integer value for an <i>entry-name</i>
Values	<i>list</i>	The list <code>(:enum name)</code>
Description		<p>The macro <code>define-c-enum</code> is used to define a FLI enumerator type specifier, which corresponds to the C <code>enum</code> type. It is a convenience function, as an enumerator type could also be defined using <code>define-foreign-type</code>.</p> <p>Each entry in the <i>enumerator-list</i> can either consist of a symbol, in which case the first entry has an integer value of 0, or of a list of a symbol and its corresponding integer value.</p>
Example		<p>In the following example a FLI enumerator type specifier is defined, and the corresponding definition for a C enumerator type follows.</p> <pre>(define-c-enum colors red green blue) enum colors { red, green, blue};</pre> <p>The next example illustrates how to start the enumerator value list counting from 1, instead of from the default start value of 0.</p> <pre>(define-c-enum half_year (jan 1) feb mar apr may jun) enum half_year { jan = 1, feb, mar, apr, may, jun }</pre>
See also		<p><code>define-c-struct</code> <code>define-c-typedef</code> <code>define-c-union</code> <code>define-foreign-type</code> <code>enum-symbol-value</code> “FLI Types” on page 11</p>

define-c-struct

Macro

Summary	Defines a FLI structure type specifier corresponding to the C <code>struct</code> type.	
Package	<code>fli</code>	
Signature	<pre>define-c-struct <i>name-and-options</i> &rest <i>descriptions</i> => <i>list</i> <i>name-and-options</i> ::= <i>name</i> (<i>name</i> <i>option</i>*) <i>option</i> ::= (:foreign-name <i>string</i>) <i>descriptions</i> ::= {<i>slot-description</i> <i>byte-packing</i> <i>aligned</i>}* <i>slot-description</i> ::= {<i>slot-name</i> (<i>slot-name</i> <i>slot-type</i>)} <i>byte-packing</i> ::= (:byte-packing <i>nbytes</i>) <i>aligned</i> ::= (:aligned <i>nbytes</i>) <i>nbytes</i> ::= integer</pre>	
Arguments	<i>name</i>	A symbol naming the new structure type specifier
	<i>string</i>	A string specifying the foreign name of the structure.
	<i>slot-description</i>	A symbol, or a list of symbol and type description, naming a slot in the structure
	<i>slot-name</i>	A symbol naming the slot
	<i>slot-type</i>	The foreign type of the slot
	<i>byte-packing</i>	A list specifying byte packing for the subsequent slots
	<i>nbytes</i>	The number of 8-bit bytes to pack
Values	<i>list</i>	The list <code>(:struct <i>name</i>)</code>
Description	The macro <code>define-c-struct</code> is used to define a FLI structure type specifier, which corresponds to the C <code>struct</code> 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.

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.

- | | |
|---------|--|
| Notes | <i>string</i> , specifying the foreign name, is supported only for documentation purposes. |
| Example | 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))

(fli:define-foreign-function fn ()
  :result-type (:pointer (:struct 3dvector)))

(let ((vector (fn)))
  (fli:with-foreign-slots (x y z) vector
    (values x y z)))

```

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:

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

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

```
(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
“FLI Types” on page 11
```

define-c-typedef

Macro

Summary	Defines FLI type specifiers corresponding to type specifiers defined using the C <code>typedef</code> command.	
Package	<code>fli</code>	
Signature	<pre>define-c-typedef <i>name-and-options</i> <i>type-description</i> => <i>name</i> <i>name-and-options</i> ::= <i>name</i> (<i>name</i> <i>option</i>*) <i>option</i> ::= (:foreign-name <i>string</i>)</pre>	
Arguments	<i>name</i>	A symbol naming the new FLI type
	<i>string</i>	A string specifying the foreign name of the type
	<i>type-description</i>	A symbol or list defining the new type

Values *name* The name of the new FLI type

Description The `define-c-typedef` macro 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`.

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

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

These are the corresponding C `typedef` definitions:

```
typedef int *intptr;
typedef struct (int one;) bar;
```

See also `define-c-enum`
 `define-c-struct`
 `define-c-union`
 `define-foreign-type`
 “FLI Types” on page 11

define-c-union *Macro*

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

Package `fli`

Signature `define-c-union` *name-and-options* &rest *slot-descriptions* => *list*
name-and-options ::= *name* | (*name* *option**)
option ::= (:foreign-name *string*)
slot-descriptions ::= {*slot-name* | (*slot-name* *slot-type*)}*

Arguments	<i>name</i>	A symbol naming the new union type descriptor
	<i>string</i>	A string specifying the foreign name of the type
	<i>slot-descriptions</i>	A sequence of symbols, possibly with type descriptions, naming the slots of the union.
	<i>slot-name</i>	A symbol naming the slot.
	<i>slot-type</i>	The FLI type of the slot.
Values	<i>list</i>	The list (<code>:union name</code>).
Description	<p>The macro <code>define-c-union</code> is used to define a FLI union type specifier, which corresponds to the C <code>union</code> type. It is a convenience function, as a union type could also be defined using <code>define-foreign-type</code>.</p> <p>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 <code>foreign-slot-type</code> and <code>foreign-slot-value</code> functions.</p>	
Example	<p>In the following example a union is defined using <code>define-c-union</code>, and the corresponding C code is given.</p> <pre>(fli:define-c-union a-point (x :int) (color :byte) (ident :char)) union a-point { int x; byte color; char ident; };</pre>	
See also	<p><code>define-c-enum</code> <code>define-c-struct</code> <code>define-c-typedef</code></p>	

`define-foreign-type`
“FLI Types” on page 11

`define-foreign-block-callable-type`

Macro

Summary	Defines a type for foreign blocks, in LispWorks for Macintosh.
Package	<code>fli</code>
Signature	<code>define-foreign-block-callable-type</code> <i>name</i> <i>result-type</i> <i>arg-types</i> => <i>name</i>
Arguments	<i>name</i> A symbol. <i>result-type</i> A foreign type specifier. <i>arg-types</i> A list of foreign type specifiers.
Description	<p>The macro <code>define-foreign-block-callable-type</code> defines a type for foreign blocks.</p> <p><i>name</i> specifies the name of the type. It must not be the same as the name of a <code>define-foreign-callable</code>.</p> <p><i>result-type</i> specifies the type of the result of the foreign block.</p> <p><i>arg-types</i> specifies the types of the arguments that a block of type <i>name</i> takes. These must correspond to the argument types with which the block is called from the foreign call.</p> <p>Note that the <i>arg-types</i> specify the type for a call from foreign code into Lisp, which affects the way <code>:reference-return</code> and <code>:reference-pass</code> 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 <code>:reference-return</code> (like <code>define-foreign-callable</code> does). See the <code>qsort_b</code> example in</p> <pre>(example-edit-file "fli/foreign-blocks")</pre>

`define-foreign-block-callable-type` returns *name*.

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`
 “Block objects in C (foreign blocks)” on page 60

`define-foreign-block-invoker` *Macro*

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

Package `ffi`

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

Arguments *the-name* A symbol.
args Arguments.

The other arguments are as for `define-foreign-function`

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`, and all the remaining arguments are interpreted in the same way.

Notes

The lambda list of the invoker is *(block . args)*. When *lambda-list* is supplied, `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 `define-foreign-funcallable` should not include an argument for the function.

`define-foreign-block-invoker` returns *the-name*.

`define-foreign-block-invoker` is implemented in Lisp-Works for Macintosh only.

Examples

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

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

See also

`define-foreign-funcallable`

`define-foreign-function`

`foreign-block-pointer`

“Block objects in C (foreign blocks)” on page 60

define-foreign-callable

Macro

Summary

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

Package

`fli`

Signature

```
define-foreign-callable (foreign-name &key encode language result-type result-pointer no-check calling-convention) ({args}*)  
&body body => lisp-name
```

```
args ::= {arg-name} | (arg-name arg-type)
```

	<i>language</i>	<code>::= :c :ansi-c</code>
Arguments	<i>foreign-name</i>	A string or symbol naming the Lisp callable function created.
	<i>encode</i>	By default, LispWorks performs automatic name encoding to translate <i>foreign-name</i> . If you want to explicitly specify an encoding, the <i>encode</i> option can be one of the following: <ul style="list-style-type: none"> <code>:source</code> tells LispWorks that <i>foreign-name</i> is the function name to call from the foreign source code. This is the default value of <i>encode</i> if <i>foreign-name</i> is a string. <code>:object</code> tells LispWorks that <i>foreign-name</i> is the literal name to be called in the foreign object code. <code>:lisp</code> tells LispWorks that if <i>foreign-name</i> is a Lisp symbol, it must be translated and encoded. This is the default value of <i>encode</i> if <i>foreign-name</i> is a symbol. <code>:dbs</code> modifies the function name on Windows, as described for <code>define-foreign-function</code>.
	<i>language</i>	The language in which the foreign calling code is written. The default is <code>:ansi-c</code> .
	<i>result-type</i>	The FLI type of the Lisp foreign callable function's return value which is passed back to the calling code.
	<i>result-pointer</i>	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.

	<i>no-check</i>	If <code>nil</code> , the result of the foreign callable function, produced by <i>body</i> , is checked to see if matches the <i>result-type</i> , and an error is raised if they do not match. Setting <i>no-check</i> to <code>t</code> overrides this check.
	<i>calling-convention</i>	Specifies the calling convention used on Windows and ARM.
	<i>args</i>	The arguments of the Lisp foreign callable function. Each argument can consist either of an <i>arg-name</i> , in which case LispWorks assumes it is an <code>:int</code> , or an <i>arg-name</i> and an <i>arg-type</i> , which is a FLI type.
	<i>body</i>	A list of forms which make up the Lisp foreign callable function.
Values	<i>lisp-name</i>	A string or symbol naming the Lisp callable function created.
Description		<p>The macro <code>define-foreign-callable</code> 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.</p> <p>When you use <code>:reference</code> with <code>:lisp-to-foreign-p t</code> as an <i>arg-type</i>, you need to set <i>arg-name</i> 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 <code>define-foreign-callable</code> form returns.</p>

calling-convention is ignored on platforms other than Windows and ARM, where there is no calling convention issue. On 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"`.

On ARM platforms the value of *calling-convention* can also be `:soft-float` or `:hard-float` (see “Foreign callables and foreign functions” on page 31 for details). Any other value produces bi-compatible code. Note that for variadic functions *calling-convention* `:soft-float` is required.

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 the *result-pointer* argument 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")
```

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 *LispWorks Delivery User Guide* for details, and take care to specify the package). However it is simpler to name the foreign call-

able 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`.

Example

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.

```
(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`.

```
(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.

```
(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`:

```
(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
   "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
```

Chapter 4, “Defining foreign functions and callables”
“Operations on foreign blocks” on page 62

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 <i>type-name lambda-list object-names &key foreign-type foreign-to-lisp lisp-to-foreign predicate tested-value error-form documentation => type-name</i>	
Arguments	<i>type-name</i>	A symbol naming the new FLI type.
	<i>lambda-list</i>	A lambda list which is the argument list of the new FLI type.
	<i>object-names</i>	A symbol or a list of two symbols

<i>foreign-type</i>	A macro expansion form that evaluates to a FLI type descriptor
<i>foreign-to-lisp</i>	A macro expansion form to convert between Lisp and the FLI.
<i>lisp-to-foreign</i>	A macro expansion form to convert between the FLI and Lisp.
<i>predicate</i>	A macro expansion form to check whether a Lisp object is of this type.
<i>tested-value</i>	A macro expansion form to give an error if a Lisp object is not of this type.
<i>error-form</i>	A macro expansion form to give an error if <i>predicate</i> returns false.
<i>documentation</i>	A string.
<i>object-names ::=</i>	<i>object-name</i> (<i>lisp-object-name</i> <i>foreign-object-name</i>)

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 the *foreign-type*, *foreign-to-lisp*, *lisp-to-foreign*, *predicate* and *tested-value* forms.

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 the *lisp-to-foreign*, *predicate* and *tested-value* forms and *foreign-object-name* provides the name of a variable for use in the *foreign-to-lisp* form.

When the new FLI type is used, the *foreign-type* form 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`), the *foreign-to-lisp* form 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*:

```
(fli:define-foreign-converter real-double (lisp-type)
  object
  :foreign-type :double
  :foreign-to-lisp `(coerce ,object ',lisp-type)
  :lisp-to-foreign `(coerce ,object 'double-float)
  :predicate `(realp ,object))
```

This defines a FLI type `int-signum`, which uses -1, 0 and 1 for values on the foreign side. There is no *foreign-to-lisp* form specified, so it will return these values to Lisp too:

```
(fli:define-foreign-converter int-signum () object
  :foreign-type :int
  :lisp-to-foreign `(signum ,object))
```

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

```
(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))

(fli:with-dynamic-foreign-objects ((x :int 10))
  (fli:dereference x :type '(bigger-in-lisp 2))) => 12
```

See also

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

:wrapper

“Parameterized types” on page 16

define-foreign-forward-reference-type*Macro*

Summary	Defines a FLI type specifier if it is not already defined.
Package	<code>fli</code>
Signature	<code>define-foreign-forward-reference-type <i>type-name</i> <i>lambda-list</i> &body <i>forms</i> => <i>type-name</i></code>
Arguments	These are interpreted as in <code>define-foreign-type</code> .
Values	<i>type-name</i> The name of the FLI type.
Description	The macro <code>define-foreign-forward-reference-type</code> defines a new FLI type called <i>type-name</i> , unless <i>type-name</i> is already defined. This macro is useful when a type declaration is needed but the full definition is not yet available.
See also	<code>define-foreign-type</code> <code>define-opaque-pointer</code>

define-foreign-funcallable*Macro*

Summary	Defines a Lisp function which, when passed a pointer to a foreign function, calls it.
Package	<code>fli</code>
Signature	<code>define-foreign-funcallable <i>the-name</i> <i>args</i> &key <i>lambda-list</i> <i>documentation</i> <i>result-type</i> <i>language</i> <i>no-check</i> <i>calling-convention</i> => <i>the-name</i></code> <code><i>args</i> ::= ({<i>arg</i>}*)</code>

Arguments	<i>the-name</i> A symbol naming the Lisp function. The other arguments are interpreted as by <code>define-foreign-function</code> .
Description	This is like <code>define-foreign-function</code> , but creates a function with an extra argument at the start of the argument list for the address to call.
Example	Define a caller for this shape: <pre>(fli:define-foreign-funcallable call-with-string-and-int ((string (:reference-pass :ef-mb-string)) (value :int)))</pre> <code>printf</code> is defined here: <pre>(fli:register-module "msvcrt")</pre> Call <code>printf</code> . Note that the output goes to console output which is hidden by default: <pre>(let ((printf-func (fli:make-pointer :symbol-name "printf"))) (call-with-string-and-int printf-func "printf called with %d" 1234))</pre>
See also	<code>define-foreign-function</code>

define-foreign-function

Macro

Summary	Defines a Lisp function which acts as an interface to a foreign function.
Package	<code>fli</code>
Signature	<code>define-foreign-function</code> <i>name</i> (<i>{arg}</i> *) &key <i>lambda-list</i> <i>documentation</i> <i>result-type</i> <i>language</i> <i>no-check</i> <i>calling-convention</i> <i>module</i> => <i>lisp-name</i> <i>name</i> ::= <i>lisp-name</i> (<i>lisp-name</i> <i>foreign-name</i> [<i>encoding</i>])

encoding ::= `:source` | `:object` | `:lisp` | `:dbcs`

arg ::= `arg-name` | (`arg-name arg-type`) | (`:constant value value-type`) | `&optional` | `&key` | ((`arg-name default`) `arg-type`) | (`:ignore arg-type`)

language ::= `:c` | `:ansi-c`

calling-convention ::= `:stdcall` | `:cdecl`

Arguments	<i>lisp-name</i>	A symbol naming the defined Lisp function.
	<i>foreign-name</i>	A string or a symbol specifying the foreign name of the function.
	<i>arg-name</i>	A variable.
	<i>arg-type</i>	A foreign type name.
	<i>value</i>	A Lisp object.
	<i>value-type</i>	A foreign type name.
	<i>lambda-list</i>	The lambda list to be used for the defined Lisp function.
	<i>documentation</i>	A documentation string for the foreign function.
	<i>result-type</i>	A foreign type.
	<i>result-pointer</i>	The name of the keyword argument that is added to the lambda-list of the Lisp function when the result-type is an aggregate type.
	<i>language</i>	The language in which the foreign source code is written. The default is <code>:ansi-c</code> .
	<i>no-check</i>	If <code>nil</code> , the types of the arguments provided when the Lisp function is called are compared with the expected types and an error is raised if they do not match. Setting <i>no-check</i> to <code>t</code> overrides this check. If the compilation safety level is set to 0 then <i>no-check</i> is automatically set to <code>t</code> . The default value for <i>no-check</i> is <code>nil</code> .

calling-convention

Specifies the calling convention used on Windows and ARM.

module

A symbol or string naming the module in which the foreign symbol is defined.

Values

lisp-name

A symbol naming the defined Lisp function.

Description

The macro `define-foreign-function` defines a Lisp function *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.

encoding specifies how the Lisp function name is translated into the function name in the foreign object code. Its values are interpreted as follows:

`:source`

foreign-name is the name of the function in the foreign source code. This is the default value of *encoding* when *foreign-name* is a string.

`:object`

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

`:lisp`

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.

`:dbcs`

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.

The number and types of the arguments of *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 *lambda-list*.

If the *arg-name* syntax of *arg* is used, then `define-foreign-function` assumes that it is of type `:int`. Otherwise *arg-type* or *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.

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*.

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 the *result-pointer* argument 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 lifetime 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 platforms other than Windows and ARM, where there is no calling convention issue. On 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"`.

On ARM platforms the value of *calling-convention* can also be `:soft-float` or `:hard-float` (see “Foreign callables and foreign functions” on page 31 for details). Any other value produces bi-compatible code. Note that for variadic functions *calling-convention* `:soft-float` is required.

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.

Notes The *module* argument is not accepted in LispWorks for UNIX. This restriction applies to LispWorks for SPARC Solaris.

Compatibility note 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`.

Example A simple example of the use of `define-foreign-function` is given in “Defining a FLI function” on page 6. More detailed examples are given in Chapter 5, “Advanced Uses of the FLI”.

Here is an example using the `:reference-return` type.

Unix/Linux/Macintosh version:

```
int cfloor(int x, int y, int *remainder)
{
    int quotient = x/y;
    *remainder = x - y*quotient;
    return quotient;
}
```

Windows version:

```

__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:

```

(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:

```

(define-foreign-function one-or-two-ints
  ((arg-one :int)
   &optional
   ((arg-two 42) :int)))

```

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

```

Chapter 4, “Defining foreign functions and callables”

define-foreign-pointer

Macro

Summary Defines a new FLI pointer type.

Package	<code>fli</code>
Signature	<pre>define-foreign-pointer <i>name-and-options</i> <i>points-to-type</i> &rest <i>slots</i> => <i>type-name</i> <i>name-and-options</i> ::= <i>type-name</i> (<i>type-name</i> (<i>option*</i>)) <i>option</i> ::= (<i>option-name</i> <i>option-value</i>)</pre>
Arguments	<p><i>type-name</i> A symbol naming the new FLI type.</p> <p><i>option-name</i> <code>:allow-null</code> or a <code>defstruct</code> option.</p> <p><i>option-value</i> A symbol.</p> <p><i>points-to-type</i> A foreign type.</p> <p><i>slots</i> Slots of the new type.</p>
Values	<i>type-name</i> The name of the new FLI pointer type.
Description	<p>The macro <code>define-foreign-pointer</code> defines a new FLI pointer type called <i>type-name</i>.</p> <p><i>type-name</i> is a subtype of <code>pointer</code>.</p> <p>The option <code>:allow-null</code> takes an <i>option-value</i> of either <code>t</code> or <code>nil</code>, defaulting to <code>nil</code>. It controls whether the type <i>type-name</i> accepts <code>nil</code>.</p> <p>The other allowed options are the <code>defstruct</code> options <code>:conc-name</code>, <code>:constructor</code>, <code>:predicate</code>, <code>:print-object</code>, <code>:print-function</code>. In each case the symbol supplied as <i>option-value</i> provides the corresponding operator for <i>type-name</i>.</p>
See also	“Creating pointers” on page 21

define-foreign-type*Macro*

Summary Defines a new FLI type specifier.

Package	<code>fli</code>
Signature	<pre> define-foreign-type <i>name-and-options</i> <i>lambda-list</i> &body <i>forms</i> => <i>name</i> <i>name-and-options</i> ::= <i>name</i> (<i>name</i> <i>option</i>*) <i>option</i> ::= (:foreign-name <i>string</i>) </pre>
Arguments	<p><i>name</i> A symbol naming the new FLI type</p> <p><i>string</i> A string specifying the foreign name of the type</p> <p><i>lambda-list</i> A lambda list which is the argument list of the new FLI type</p> <p><i>forms</i> One or more Lisp forms which provide a definition of the new type</p>
Values	<i>name</i> The name of the new FLI type
Description	The macro <code>define-foreign-type</code> defines a new FLI type called <i>name</i> . The <i>forms</i> in the definition can be used to determine the behavior of the type, depending on the arguments supplied to the <i>lambda-list</i> .
Example	<p>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.</p> <pre> (fli:define-foreign-type :int-array (dimensions) `(:c-array :int ,@dimensions)) (setq number-array (fli:allocate-foreign-object :type '(:int-array (2 2)))) </pre> <p>In the next example a boolean type, called <code>:bool</code>, with the same size as an integer is defined.</p> <pre> (fli:define-foreign-type :bool () `(:boolean :int)) (fli:size-of :bool) </pre>

See also `define-c-typedef`
`define-foreign-converter`
`define-foreign-forward-reference-type`
`foreign-type-equal-p`
“FLI Types” on page 11
“Defining new types” on page 57

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` | `:dbcs`

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.

lisp-name A symbol naming the Lisp accessor.

foreign-name A string or a symbol specifying the foreign name of the variable.

encoding An option controlling how the Lisp variable name is translated to match the foreign variable name in the foreign DLL. The *encoding* option 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.

:dbcs modifies the variable name on Windows, as described for **define-foreign-function**.

type

The FLI type corresponding to the type of the foreign variable to which Lisp is interfacing.

accessor

An option specifying what kind of accessor is generated for the variable. It can be one of the following:

:value gets the value of the foreign variable directly. This is the default value when *type* is a non-aggregate type. (There is no default *accessor* for aggregate types.)

:address-of returns a FLI pointer to the foreign variable.

:read-only ensures that no **setf** method 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.

	<i>language</i>	The language in which the foreign source code for the variable is written. The default is <code>:ansi-c</code> .
	<i>no-check</i>	If <code>nil</code> , the types of the arguments provided when the Lisp function is called are compared with the expected types and an error is raised if they do not match. Setting <i>no-check</i> to <code>t</code> overrides this check.
	<i>module</i>	A string or symbol naming the module in which the foreign variable is defined.
Values	<i>lisp-name</i>	A symbol naming the Lisp accessor.
Description		<p>The macro <code>define-foreign-variable</code> defines a Lisp accessor which can be used to get and set the value of a variable defined in foreign code.</p> <p>If the foreign variable has a type corresponding to an FLI aggregate type, then <i>accessor</i> must be supplied (there is no default). If <i>accessor</i> is <code>:value</code>, then a copy of the object is allocated using <code>allocate-foreign-object</code>, and the copy is returned. In general, it is more useful to use <i>accessor</i> <code>:address-of</code> for aggregate types, to allow the original aggregate to be updated.</p>
Notes		<p>If you specify any of the FLI float types <code>:float</code>, <code>:double</code>, <code>:lisp-float</code>, <code>:lisp-single-float</code> and so on, then the value of <i>language</i> should be <code>:ansi-c</code>.</p> <p><i>module</i> is processed as for <code>define-foreign-function</code>.</p>
Example		<p>The following example illustrates how to use the FLI to define a foreign variable, given the following C variable in a DLL:</p> <pre>int num;</pre>

The first example defines a Lisp variable, `num1`, to interface with the C variable `num`.

```
(fli:define-foreign-variable (num1 "num") :type :int)
```

The following commands return the value of `num`, and increase its value by 1:

```
(num1)
(incf (num1))
```

In the next example, the Lisp variable `num2` interfaces with `num` in a read-only manner.

```
(fli:define-foreign-variable (num2 "num")
  :type :int :accessor :READ-ONLY)
```

In this case, the next command still returns the value of `num`, but the second command raises an error, because `num2` is read-only.

```
(num2)
(incf (num2))
```

The final example defines a Lisp variable, `num3`, which accesses `num` through pointers.

```
(fli:define-foreign-variable (num3 "num")
  :type :int :accessor :address-of)
```

As a result, the next command returns a pointer to `num`, and to obtain the actual value stored by `num`, `num3` needs to be dereferenced.

```
(num3)
(fli:dereference (num3))
```

See also

```
define-foreign-callable
define-foreign-function
```

define-opaque-pointer*Macro*

Summary	Defines an opaque foreign pointer type.
Package	<code>ffi</code>
Signature	<code>define-opaque-pointer <i>pointer-type</i> <i>structure-type</i></code>
Arguments	<p><i>pointer-type</i> A symbol.</p> <p><i>structure-type</i> A symbol.</p>
Description	<p>The macro <code>define-opaque-pointer</code> defines an opaque foreign pointer type and foreign 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</p> <pre>typedef struct <i>structure-type</i> *<i>pointer-type</i>;</pre> <p>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.</p>
Example	Using the C standard <code>file*</code> 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`

dereference

Function

Summary Accesses and returns the value of a foreign object.

Package `fli`

Signature `dereference pointer &key index type copy-foreign-object => value`

(**setf dereference**) *value pointer &key index type copy-foreign-object => value*

Arguments	<i>pointer</i>	An instance of a FLI pointer.
	<i>index</i>	An integer. If <i>index</i> is supplied, dereference assumes that <i>pointer</i> points to one element in an array of object, and returns the element at the <i>index</i> position in the array.
	<i>type</i>	The foreign object type that <i>pointer</i> points to. If the specified type is different to the actual type, dereference returns the value of the object in the format of <i>type</i> where possible.
		<p><i>copy-foreign-object</i></p> <p>This option is only important when dealing with aggregate FLI types, which cannot be returned by value.</p> <p>If set to t, dereference makes a copy of the aggregate object pointed to by <i>pointer</i> and returns the copy.</p> <p>If set to nil, dereference returns the aggregate object directly.</p> <p>If set to :error then dereference signals an error. This is the default value for <i>copy-foreign-object</i>.</p>
Values	<i>value</i>	The value of the dereferenced object at <i>pointer</i> .
Description		<p>The function dereference accesses and returns the value of the FLI object pointed to by <i>pointer</i>, unless <i>pointer</i> points to an aggregate type. In the case of aggregates, the return value is specified by using the <i>copy-foreign-object</i> option.</p> <p>An error is signaled if <i>value</i> is an aggregate type and <i>copy-foreign-object</i> is not set accordingly.</p>

The value of an object at *pointer* can be changed using `(setf 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`.

Example

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 `(setf 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`

“FLI Types” on page 11

“Pointer dereferencing and coercing” on page 24

“Calling a C function that takes an array of strings” on page 46

disconnect-module*Function*

Summary Disconnects the DLL associated with a registered module.

Package `fli`

Signature `disconnect-module name &key verbose remove => result`

Arguments *name* A symbol or string.
 verbose `nil`, `t` or an output stream.
 remove A boolean.

Values *result* A boolean.

Description The function `disconnect-module` disconnects the DLL associated with a registered module specified by *name* and registered with `register-module`.

When disconnecting, if *verbose* is a stream, then `disconnect-module` will send disconnection information to that stream. If *verbose* is `t`, this is interpreted as standard output. The default value of *verbose* is `nil`.

`disconnect-module` returns `t` if the module was found. If the module was not found, `disconnect-module` merely returns `nil`.

If *remove* is `nil` then after disconnection the module will be in the same state as it was when first registered by `register-module`, that is, lookups for foreign symbols can still automatically reconnect the DLL. If *remove* is `t` then *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 *remove* is `nil`.

See also `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`

Signature `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 *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 the *enum-type*, where a pair is a cons of symbol and value.

enum-type must be defined by `define-c-enum`.

Example

```
(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.
- `byte` 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.

Example

```
(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 *Function*

Summary	Accesses and returns the value at a specified point in an array.	
Package	fli	
Signature	<code>foreign-aref array &rest <i>subscripts</i> => value</code> <code>(setf foreign-aref) value array &rest <i>subscripts</i> => value</code>	
Arguments	<i>array</i>	A FLI array or a pointer to a FLI array.
	<i>subscripts</i>	A list of valid array indices for <i>array</i> .
Values	<i>value</i>	An element of <i>array</i> .
Description	<p>The function <code>foreign-aref</code> accesses a specified element in an array and returns its value if the element is an immediate type. If it is an aggregate type, such as a <code>:struct</code>, <code>:union</code>, or <code>:c-array</code>, an error is signaled. The function <code>foreign-array-pointer</code> should be used to access such embedded aggregate data.</p> <p>The value of an element in an array can be changed using <code>(setf foreign-aref)</code>. See the examples section for an example of this.</p>	
Example	<p>In the first example, a 3 by 3 integer array is created, and <code>(setf foreign-aref)</code> is used to set all the elements to 42.</p> <pre>(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)))</pre> <p>Next, <code>foreign-aref</code> is used to dereference the value at position 2 2 in <code>array1</code>. Remember that the count for the indices of an array start at 0.</p>	

```
(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.

```
(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.

```
(setq array3 *)

(fli:foreign-aref array3 1)
```

See also

“FLI Types” on page 11
`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	<code>fli</code>	
Signature	<code>foreign-array-dimensions</code> <i>array-or-type</i> => <i>dimensions</i>	
Arguments	<i>array-or-type</i>	A FLI array, a pointer to a FLI array or the name of a FLI array type.
Values	<i>dimensions</i>	A list containing the dimensions of <i>array-or-type</i> .

Description	The function <code>foreign-array-dimensions</code> takes a FLI array, a pointer to a FLI array or the name of a FLI array type as its argument and returns a list containing the dimensions of the array.
Examples	In the following example an instance of a 3 by 4 array is created, and these dimensions are returned using the <code>foreign-array-dimensions</code> function. <pre>(setq array1 (fli:allocate-foreign-object :type '(:c-array :int 3 4))) (fli:foreign-array-dimensions array1)</pre>
See also	<code>foreign-aref</code> <code>foreign-array-element-type</code> <code>foreign-array-pointer</code>

foreign-array-element-type*Function*

Summary	Returns the type of the elements of an array.
Package	<code>fli</code>
Signature	<code>foreign-array-element-type <i>array-or-type</i> => <i>type</i></code>
Arguments	<i>array-or-type</i> A FLI array, a pointer to a FLI array or the name of a FLI array type.
Values	<i>type</i> The type of the elements of <i>array-or-type</i> .
Description	The function <code>foreign-array-element-type</code> takes a FLI array, a pointer to a FLI array or the name of a FLI array type as its argument and returns the type of the elements of that array.
Examples	In the following example a 3 by 4 array with integer elements is defined, and the <code>foreign-array-element-type</code> 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

Function

Summary	Returns a pointer to a specified element in an array.
Package	<code>fli</code>
Signature	<code>foreign-array-pointer array &rest <i>subscripts</i> => <i>pointer</i></code>
Arguments	<i>array</i> A FLI array or a pointer to a FLI array. <i>subscripts</i> A list of valid array indices for <i>array</i> .
Values	<i>pointer</i> A pointer to the element at position <i>subscripts</i> in <i>array</i> .
Description	The function <code>foreign-array-pointer</code> returns a pointer to a specified element in an array. The value pointed to can then be obtained by dereferencing the pointer returned, or set to a specific value using <code>(setf dereference)</code> .
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 <code>foreign-array-pointer</code> . <pre>(setq array1 (fli:allocate-foreign-object :type '(:c-array :int 3 2))) (setq array-ptr (fli:foreign-array-pointer array1 2 0))</pre>

The (`setf dereference`) function 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* A foreign block pointer.

Results *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`
 “Block objects in C (foreign blocks)” on page 60

foreign-block-release *Function*

Summary Releases a foreign block, like `_Block_release`, in LispWorks for Macintosh.

Package `fl1`

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`
 “Block objects in C (foreign blocks)” on page 60

foreign-function-pointer

Function

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`
 “Creating pointers” on page 21

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*.

Description The `foreign-slot-names` function returns a list containing the slot names of a foreign object defined by `define-c-struct`. If *object* is not a structure, an error is signaled.

Example 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.

```
(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 “Structures and unions” on page 15
`define-c-struct`
`foreign-slot-value`

foreign-slot-offset

Function

Summary	Returns the offset of a slot in a FLI object.
Package	<code>fli</code>
Signature	<code>foreign-slot-offset <i>object-or-type slot-name</i> => <i>offset</i></code>
Arguments	<p><i>object-or-type</i> A foreign object, a pointer to a foreign object, or a foreign structure or union type.</p> <p><i>slot-name</i> A symbol or a list of symbols identifying the slot to be accessed, as described for <code>foreign-slot-value</code>.</p>
Values	<i>offset</i> The offset, in bytes, of the slot <i>slot-name</i> in the FLI object <i>object</i> .
Description	The function <code>foreign-slot-offset</code> returns the offset, in bytes, of a slot in a FLI object. 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.
Example	The following example defines a structure, creates an instance of the structure pointed to by <code>dir</code> , and then finds the offset of the third slot in the object.

```
(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)
```

See also `foreign-slot-value`
`foreign-slot-pointer`
`size-of`

foreign-slot-pointer*Function*

Summary	Returns a pointer to a specified slot of an object.	
Package	<code>fli</code>	
Signature	<code>foreign-slot-pointer</code> <i>object slot-name</i> &key <i>type object-type</i> => <i>pointer</i>	
Arguments	<i>object</i>	A foreign object, or a pointer to a foreign object.
	<i>slot-name</i>	A symbol or a list of symbols identifying the slot to be accessed, as described for <code>foreign-slot-value</code> .
	<i>type</i>	The type of the slot <i>slot-name</i> .
	<i>object-type</i>	The FLI structure type that contains <i>slot-name</i> . 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 <i>object</i> .
Values	<i>pointer</i>	A pointer to the slot identified by <i>slot-name</i> .
Description	The function <code>foreign-slot-pointer</code> takes a foreign object, a slot within the object, and optionally the type of the slot, and returns a pointer to the slot.	
Example	In the following example a structure type called <code>COMPASS</code> is defined. An instance of the structure is allocated using <code>allocate-foreign-object</code> , pointed to by <code>point1</code> . Then	

`foreign-slot-pointer` is used to get a pointer, called `point2`, to the second slot of the foreign object.

```
(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.

```
(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`.

```
(setf (fli:dereference point2) 0)

(fli:dereference point3)

(setf (fli:dereference point2) 1)

(fli:dereference point3)
```

See also

“Structures and unions” on page 15

```
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	<code>fli</code>	
Signature	<code>foreign-slot-type</code>	<i>object-or-type slot-name => type</i>
Arguments	<i>object-or-type</i>	A foreign object, a pointer to a foreign object, or a foreign structure or union type.
	<i>slot-name</i>	A symbol or a list of symbols identifying the slot whose type is to be returned. The value is interpreted as described for <code>foreign-slot-value</code> .
Values	<i>type</i>	The type of <i>slot-name</i> .
Description	The function <code>foreign-slot-type</code> returns the type of a slot of a foreign object.	
Example	<p>In the following example two new types, <code>EAST</code> and <code>WEST</code> are defined. Then a new structure, <code>COMPASS</code>, is defined, with two slots. An instance of the structure is created, and <code>foreign-slot-type</code> is used to get the type of the first slot of the structure.</p> <pre>(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)</pre>	
See also	<p>“Structures and unions” on page 15 <code>foreign-slot-names</code> <code>foreign-slot-value</code></p>	

foreign-slot-value

Function

Summary	Returns the value of a slot in a foreign object.
Package	<code>fli</code>
Signature	<pre>foreign-slot-value <i>object slot-name</i> &key <i>type object-type</i> copy-foreign-object => <i>value</i> (setf foreign-slot-value) <i>value object slot-name</i> &key <i>type</i> <i>object-type copy-foreign-object</i> => <i>value</i></pre>
Arguments	<p><i>object</i> Either an instance of or a pointer to a FLI structure.</p> <p><i>slot-name</i> A symbol or a list of symbols identifying the slot to be accessed.</p> <p><i>type</i> The type of <i>value</i>. Specifying <i>type</i> makes accessing the object faster. If the specified type is different to the actual type, <code>foreign-slot-value</code> returns the <i>value</i> in the format of <i>type</i> where possible.</p> <p><i>object-type</i> The FLI structure type that contains <i>slot-name</i>. 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 <i>object</i>.</p> <p><i>copy-foreign-object</i> This option is only important when dealing with slots which are aggregate FLI types, and cannot be returned by value. The recognized values are <code>t</code>, <code>nil</code> and <code>:error</code>: If <i>copy-foreign-object</i> is <code>t</code>, <code>foreign-slot-value</code> makes a copy of the aggregate slot of the object pointed to by <i>pointer</i> and returns the copy.</p>

		<p>If <i>copy-foreign-object</i> is <code>nil</code>, <code>foreign-slot-value</code> returns the aggregate slot of the object directly.</p> <p>If <i>copy-foreign-object</i> is <code>:error</code> then <code>foreign-slot-value</code> signals an error. This is the default value for <i>copy-foreign-object</i>.</p>
Value	<i>value</i>	The value of the slot <i>slot-name</i> in the FLI object <i>object</i> is returned.
Description		<p>The function <code>foreign-slot-value</code> returns the value of a slot in a specified object. An error is signaled if the slot is an aggregate type and <i>copy-foreign-object</i> is not set accordingly. Use <code>foreign-slot-pointer</code> to access such aggregate slots.</p> <p>If <i>slot-name</i> is a symbol then it names the slot of <i>object</i> to be accessed. If <i>slot-name</i> is a list of symbols, then these symbols name slots in nested structures starting with the outermost structure <i>object</i>, as in the <code>inner/middle/outer</code> example below.</p> <p>The function <code>(setf foreign-slot-value)</code> can be used to set the value of a slot in a structure, as shown in the example below.</p>
Compatibility note		64-bit integer types such as <code>(:long :long)</code> , <code>:int64</code> and <code>:uint64</code> are now supported for <i>type</i> in <code>foreign-slot-value</code> in 32-bit LispWorks. In 32-bit LispWorks 6.1 and earlier versions, these types could only be used by <code>define-foreign-function</code> .
Example		In the following example a foreign structure is defined, an instance of the structure is made with <code>my-pos</code> pointing to the instance, and <code>foreign-slot-value</code> is used to set the <code>y</code> slot of the object to 10.

```
(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.

```
(fli:foreign-slot-value my-pos 'y)

(fli:foreign-slot-value my-pos 'y :object-type 'pos)
```

See the *LispWorks User Guide and Reference Manual* section "Optimizing your code" for an example showing how to inline foreign slot access.

This example accesses a slot in nested structures:

```
(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

“Structures and unions” on page 15

`foreign-slot-pointer`

`foreign-slot-offset`

`dereference`

`with-foreign-slots`

foreign-type-equal-p*Function*

Summary	Determines whether two foreign types are the same underlying foreign type.	
Package	<code>fli</code>	
Signature	<code>foreign-type-equal-p <i>type1</i> <i>type2</i> => <i>result</i></code>	
Arguments	<i>type1</i>	A foreign type.
	<i>type2</i>	A foreign type.
Values	<i>result</i>	A boolean.
Description	The function <code>foreign-type-equal-p</code> returns true if <i>type1</i> and <i>type2</i> are the same underlying foreign type, and false otherwise.	
Example	<pre>(fli:define-foreign-type aa () '(:signed :byte)) => aa (fli:define-foreign-type bb () '(:signed :char)) => bb (fli:foreign-type-equal-p 'aa 'bb) => t (fli:foreign-type-equal-p 'bb :char) => nil</pre>	
See also	Chapter 2, “FLI Types” <code>define-foreign-type</code>	

foreign-type-error

Condition Class

Summary	The class of errors signaled when an object does not match a foreign type.
Package	<code>fli</code>
Superclasses	<code>type-error</code>
Description	The condition class <code>foreign-type-error</code> is used for errors signaled when an object does not match a foreign type.

foreign-typed-aref

Function

Summary	Accesses a foreign array and can be compiled to efficient code.
Package	<code>fli</code>
Signature	<code>foreign-typed-aref type array index => value</code> <code>(setf foreign-typed-aref) value type array index => value</code>
Arguments	<i>type</i> A type specifier. <i>array</i> A foreign pointer. <i>index</i> A non-negative integer.
Values	<i>value</i> An element of <i>array</i> .
Description	The function <code>foreign-typed-aref</code> accesses a foreign array and is compiled to efficient code when compiled at safety 0. It corresponds to <code>sys:typed-aref</code> which accesses Lisp vectors. <i>type</i> must evaluate to a supported element type for foreign arrays. In 32-bit LispWorks these types are <code>double-float</code> , <code>single-float</code> , <code>(unsigned-byte 32)</code> , <code>(signed-byte 32)</code> , <code>(unsigned-byte 16)</code> , <code>(signed-byte 16)</code> , <code>(unsigned-</code>

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:

```
(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, `allocate-foreign-object`) assume `fixnum` sizes so it is best to use `fixnums` only.

Notes Efficient access to a Lisp vector object is also available. See `sys:typed-aref` in the *LispWorks User Guide and Reference Manual*.

See also “FLI Types” on page 11
`foreign-aref`

free *Function*

Summary A synonym for `free-foreign-object`.

Package `fli`

Signature `free pointer => null-pointer`

Description The function `free` is a synonym for `free-foreign-object`.

See also `free-foreign-object`

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 “Scope of invocation” on page 62.

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`
 “Block objects in C (foreign blocks)” on page 60

free-foreign-object*Function*

Summary Deallocates the space in memory pointed to by a pointer.

Package `fli`

Signature `free-foreign-object 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.

Example 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`
“An example of dynamic memory allocation” on page 7
“Allocation of FLI memory” on page 23

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

<i>name</i>	A symbol.
<i>filename</i>	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.
2. The effect of `get-embedded-module` persists after `save-image` and `deliver`.
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`

“Incorporating a foreign module into a LispWorks image” on page 60

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	<code>ffi</code>	
Signature	<code>get-embedded-module-data <i>filename</i> => <i>data</i></code>	
Arguments	<i>filename</i>	A pathname specifier for a file containing a dynamic foreign module.
Values	<i>data</i>	A Lisp object containing the data of the foreign module.
Description	The function <code>get-embedded-module-data</code> returns the foreign module in <i>filename</i> as a Lisp object suitable as argument to <code>setup-embedded-module</code> , but also externalizable, that is the compiler can put it in a fasl file.	
Notes	<ol style="list-style-type: none"> 1. <code>get-embedded-module-data</code> 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 `#.` :

```
(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 by the compiler:

```
(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
setup-embedded-module
```

“Incorporating a foreign module into a LispWorks image” on page 60

incf-pointer *Function*

Summary	Increases the address held by a pointer.	
Package	<code>fli</code>	
Signature	<code>incf-pointer <i>pointer</i> &optional <i>delta</i> => <i>pointer</i></code>	
Arguments	<i>pointer</i>	A FLI pointer.
	<i>delta</i>	An integer. The default value is 1.
Values	<i>pointer</i>	The pointer passed.
Description	<p>The function <code>incf-pointer</code> increases the address held by the pointer. If <i>delta</i> is not given the address is increased by the size of the type pointed to by the pointer. The address can be increased by a multiple of the size of the type by specifying a <i>delta</i>.</p> <p>The function <code>incf-pointer</code> is often used to move a pointer through an array of values.</p>	
Example	<p>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.</p>	

```
(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)
  (fli:incf-pointer point1)
  (print (fli:dereference point1)))
```

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

See also `decf-pointer`
“An example of dynamic pointer allocation” on page 27

install-embedded-module

Function

Summary Installs an embedded dynamic module.

Package `fli`

Signature `install-embedded-module` *name*

Arguments *name* A symbol.

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.

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`
 “Incorporating a foreign module into a LispWorks image” on
 page 60

locale-external-formats*Variable*

Summary Provides a mapping from locale names to encodings

Package `fli`

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 Linux and Unix 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

Function

Summary	Converts foreign bytes back to a Lisp integer.	
Signature	<code>make-integer-from-bytes <i>pointer</i> <i>length</i> => <i>integer</i></code>	
Arguments	<i>pointer</i>	A foreign pointer.
	<i>length</i>	An integer.
Values	<i>integer</i>	An integer.
Description	The function <code>make-integer-from-bytes</code> converts <i>length</i> bytes starting at <i>pointer</i> into the Lisp integer <i>integer</i> . The bytes and <i>length</i> must have been generated by <code>with-integer-bytes</code> or <code>convert-integer-to-dynamic-foreign-object</code> .	
See also	“Lisp integers” on page 57 <code>with-integer-bytes</code> <code>convert-integer-to-dynamic-foreign-object</code>	

make-pointer

Function

Summary	Creates a pointer to a specified address.	
Package	<code>ffi</code>	
Signature	<code>make-pointer &key <i>address</i> <i>type</i> <i>pointer-type</i> <i>symbol-name</i> <i>functionp</i> <i>module</i> <i>encoding</i> => <i>pointer</i></code>	
Arguments	<i>address</i>	The address pointed to by the pointer to be created.
	<i>type</i>	The type of the object pointed to by the pointer to be created.
	<i>pointer-type</i>	The type of the pointer to be made.

	<i>symbol-name</i>	A string or a symbol.
	<i>functionp</i>	If <i>type</i> or <i>pointer-type</i> are not specified, then <i>functionp</i> can be used. If <code>t</code> , the pointer made is a pointer to type <code>:function</code> . This is the default value. If <code>nil</code> , the pointer made is a pointer to type <code>:void</code> .
	<i>module</i>	A symbol or string naming a module, or <code>nil</code> .
	<i>encoding</i>	One of <code>:source</code> , <code>:object</code> , <code>:lisp</code> or <code>:dbs</code> .
Values	<i>pointer</i>	A pointer to <i>address</i> .
Description		<p>The function <code>make-pointer</code> creates a pointer of a specified type pointing to a given address <i>address</i>, or optionally to a function or foreign callable.</p> <p><i>symbol-name</i> 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 <code>define-foreign-callable</code>.</p> <p>Either <i>address</i> or <i>symbol-name</i> must be supplied, otherwise <code>make-pointer</code> signals an error.</p> <p>Note that in many cases, especially when <code>:symbol-name</code> is used with a symbol defined by <code>define-foreign-callable</code>, <code>foreign-function-pointer</code> would be better than using <code>make-pointer</code> with <code>:symbol-name</code>.</p> <p><i>encoding</i> controls how <i>symbol-name</i> is processed. The values are interpreted like the <i>encode</i> argument of <code>define-foreign-callable</code>. The default value of <i>encoding</i> is <code>:source</code> if <i>symbol-name</i> is a string and <code>:lisp</code> if <i>symbol-name</i> is a symbol.</p> <p>In the case of a pointer to a foreign callable or foreign function, the <i>module</i> keyword can be used to ensure that the pointer points to the function in the correct DLL if there are</p>

other DLLs containing functions with the same name. *module* is processed as by `define-foreign-function`.

Example 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))
```

See also Chapter 3, “FLI Pointers”
“Foreign callables and foreign functions” on page 31

```
copy-pointer  
define-foreign-callable  
foreign-function-pointer  
register-module  
with-coerced-pointer
```

malloc

Function

Summary A synonym for `allocate-foreign-object`.

Package `fli`

Signature `malloc &key type pointer-type initial-element initial-contents
nelems => pointer`

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

See also `allocate-foreign-object`
“FLI Pointers” on page 21

module-unresolved-symbols*Function*

Summary	Returns foreign symbol names that cannot be resolved. Note: This function is not defined in LispWorks for UNIX.
Package	<code>fli</code>
Signature	<code>module-unresolved-symbols &key <i>module</i> => <i>list</i></code>
Arguments	<i>module</i> <code>nil</code> , <code>:all</code> , or a string. The default is <code>:all</code> .
Values	<i>list</i> A list of strings.
Description	The function <code>unresolved-module-symbols</code> returns a list of foreign symbol names, each of which cannot be resolved in the currently known modules. If <i>module</i> is <code>nil</code> , then <i>list</i> includes only those names not associated with a module. If <i>module</i> is <code>:all</code> , then <i>list</i> includes the unresolved names in all modules and those not associated with a module. If <i>module</i> is a string, then it names a module and <i>list</i> contains only the unresolved symbols associated with that module.
See also	“Testing whether a function is defined” on page 59 <code>register-module</code>

null-pointer*Variable*

Summary	A null pointer.
Package	<code>fli</code>
Description	The variable <code>*null-pointer*</code> contains a <code>(:pointer :void)</code> with address 0.

This provides a simple way to pass a null pointer when needed.

Example

```
(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.

Example 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`.

```
(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 “Pointer testing functions” on page 23
 “Testing whether a function is defined” on page 59
null-pointer
pointer-address
pointer-eq

pointer-address*Function*

Summary	Returns the address of a pointer.	
Package	<code>fli</code>	
Signature	<code>pointer-address</code> <i>pointer</i> => <i>address</i>	
Arguments	<i>pointer</i>	A FLI pointer.
Values	<i>address</i>	The address pointed to by <i>pointer</i> .
Description	The function <code>pointer-address</code> returns the address of a pointer.	
Example	In the following example a pointer is defined, and its address is returned using <code>pointer-address</code> .	
	<pre>(setq point (fli:allocate-foreign-object :type :int)) (fli:pointer-address point)</pre>	
See also	“Pointer testing functions” on page 23 null-pointer-p pointer-eq	

pointer-element-size

Function

Summary	Returns the size in bytes of a foreign object or a foreign type.
Package	<code>fli</code>
Signature	<code>pointer-element-size <i>pointer-or-type</i> => <i>size</i></code>
Arguments	<i>pointer-or-type</i> A FLI pointer to a foreign object or the name of a FLI pointer type.
Values	<i>size</i> A non-negative integer.
Description	<p>The function <code>pointer-element-size</code> returns the size, in bytes, of the object or type specified.</p> <p>If <i>pointer-or-type</i> is an FLI pointer, <i>size</i> is the size, in bytes, of the object pointed to by <i>pointer-or-type</i>.</p> <p>If <i>pointer-or-type</i> is the name of a FLI pointer type, <i>size</i> is the size, in bytes, of the elements of that type.</p>
Example	<p>In the following example a pointer to an integer is created. Then the size in bytes of the integer is returned using <code>pointer-element-size</code>.</p> <pre>(setq point (fli:allocate-foreign-object :type :int)) (fli:pointer-element-size point)</pre>
See also	<p>“Pointer testing functions” on page 23</p> <p><code>pointer-element-type</code> <code>size-of</code></p>

pointer-element-type

Function

Summary	Returns the type of the foreign object pointed to by a FLI pointer.
---------	---

Package	<code>fli</code>
Signature	<code>pointer-element-type <i>pointer-or-type</i> => <i>type</i></code>
Arguments	<i>pointer-or-type</i> A FLI pointer to a foreign object or the name of a FLI pointer type.
Values	<i>type</i> The name of a FLI pointer type.
Description	<p>The function <code>pointer-element-type</code> returns the type of the foreign object specified, or the element type of the foreign type specified.</p> <p>If <i>pointer-or-type</i> is a FLI pointer, <i>type</i> is the type of the foreign object pointed to by <i>pointer-or-type</i>.</p> <p>If <i>pointer-or-type</i> is the name of a FLI pointer type, <i>type</i> is the type of the elements of that FLI pointer type.</p>
Example	<p>In the following example a pointer to an integer is defined, and <code>pointer-element-type</code> is used to confirm that the pointer points to an integer.</p> <pre>(setq point (fli:allocate-foreign-object :type :int)) (fli:pointer-element-type point)</pre> <p>In the next example a new type, <code>HAPPY</code>, is defined. The pointer <code>point</code> is set to point to an instance of <code>HAPPY</code>, and <code>pointer-element-type</code> is used to find the type of the object pointed to by <code>point</code>.</p> <pre>(fli:define-c-typedef HAPPY :long) (setq point (fli:allocate-foreign-object :type 'HAPPY)) (fli:pointer-element-type point)</pre>
See also	<p>“Pointer testing functions” on page 23</p> <p><code>foreign-slot-type</code></p> <p><code>pointer-element-size</code></p> <p><code>pointer-element-type-p</code></p>

pointer-element-type-p

Function

Summary	Tests whether a FLI pointer matches a given element type.
Package	<code>fli</code>
Signature	<code>pointer-element-type-p <i>pointer type</i> => <i>result</i></code>
Arguments	<i>pointer</i> A FLI pointer to a foreign object. <i>type</i> A foreign type.
Values	<i>result</i> A boolean.
Description	The function <code>pointer-element-type-p</code> returns true if the element type of the foreign object pointed to by <i>pointer</i> has the same underlying type as <i>type</i> .
Example	<pre>(setq point (fli:allocate-foreign-object :type :int)) => => #<Pointer to type :INT = #x007F3970> (fli:pointer-element-type-p point :signed) -> t</pre>
See also	“Pointer testing functions” on page 23 <code>pointer-element-type</code>

pointer-eq

Function

Summary	Test whether two pointers point to the same memory address.
Package	<code>fli</code>
Signature	<code>pointer-eq <i>pointer1 pointer2</i> => boolean</code>

Arguments	<i>pointer1</i>	A FLI pointer.
	<i>pointer2</i>	A FLI pointer.
Values	<code>boolean</code>	If <i>pointer1</i> points to the same address as <i>pointer2</i> , <code>pointer-eq</code> returns <code>t</code> , otherwise it returns <code>nil</code> .
Description	The function <code>pointer-eq</code> tests whether two pointers point to the same address, and returns <code>t</code> if they do, and <code>nil</code> if they do not.	
Example	<p>In the following example a pointer, <code>point1</code>, is defined, and <code>point2</code> is set equal to it. Both are then tested to see if they are equal to each other using <code>pointer-eq</code>. Then <code>point2</code> is defined to point to a different object, and the two pointers are tested for equality again.</p> <pre>(setq point1 (fli:allocate-foreign-object :type :int)) (setq point2 point1) (fli:pointer-eq point1 point2) (setq point2 (fli:allocate-foreign-object :type :int)) (fli:pointer-eq point1 point2)</pre>	
See also	<p>“Pointer testing functions” on page 23</p> <p><code>null-pointer-p</code></p> <p><code>pointerp</code></p>	

pointer-pointer-type

Function

Summary	Returns the pointer type of a FLI pointer.
Package	<code>fli</code>
Signature	<code>pointer-pointer-type</code> <i>pointer</i> => <i>pointer-type</i>

Arguments	<i>pointer</i>	A FLI pointer.
Values	<i>pointer-type</i>	The pointer type of <i>pointer</i> .
Description	The function <code>pointer-pointer-type</code> returns the pointer type of the foreign pointer <i>pointer</i> .	
Example	<pre>(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></pre>	
See also	“Pointer dereferencing and coercing” on page 24 <code>make-pointer</code>	

pointerp

Function

Summary	Tests whether an object is a pointer or not.	
Package	<code>fli</code>	
Signature	<code>pointerp <i>pointer</i> => <i>result</i></code>	
Arguments	<i>pointer</i>	An object that may be a FLI pointer.
Values	<i>result</i>	A boolean.
Description	The function <code>pointerp</code> tests whether the argument <i>pointer</i> is a pointer. <i>result</i> is <code>t</code> if <i>pointer</i> is a pointer, otherwise <code>nil</code> is returned.	

Example 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 “Pointer testing functions” on page 23
`null-pointer-p`
`pointer-address`
`pointer-eq`

print-collected-template-info

Function

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.

See the *LispWorks Delivery User Guide* for further details.

See also `start-collecting-template-info`

print-foreign-modules

Function

Summary	Prints the foreign modules loaded into the image by <code>register-module</code> .
Package	<code>fli</code>
Signature	<code>print-foreign-modules &optional <i>stream</i> <i>verbose</i> => nil</code>
Arguments	<i>stream</i> An output stream. <i>verbose</i> A generalized boolean.
Description	<p>The function <code>print-foreign-modules</code> prints a list of the foreign modules loaded via <code>register-module</code>, to the stream <i>stream</i>.</p> <p>The default value of <i>stream</i> is the value of <code>*standard-output*</code>.</p> <p>If <i>verbose</i> is true, more information is printed if possible. Currently this only has an effect in LispWorks for Unix. The default value of <i>verbose</i> is <code>nil</code>.</p>
See also	<code>register-module</code>

register-module

Function

Summary	Informs LispWorks of the presence of a dynamic library.
Package	<code>fli</code>
Signature	<code>register-module <i>name</i> &key <i>connection-style</i> <i>lifetime</i> <i>real-name</i> <i>dlopen-flags</i> => <i>name</i></code>
Arguments	<i>name</i> A symbol or string specifying the Lisp name the module will be registered under.

	<i>connection-style</i>	A keyword determining when the connection to the dynamic library is made. One of <code>:automatic</code> , <code>:manual</code> or <code>:immediate</code> . The default value is <code>:automatic</code> .
	<i>lifetime</i>	A keyword specifying the lifetime of the connection. One of <code>:indefinite</code> or <code>:session</code> . The default value is <code>:indefinite</code> .
	<i>real-name</i>	Overrides the <i>name</i> for identifying the actual dynamic library to connect to.
	<i>dlopen-flags</i>	Controls use of <code>dlopen</code> on Unix-based systems. One of <code>t</code> , <code>nil</code> , <code>:local-now</code> , <code>:global-now</code> , <code>:global-lazy</code> , <code>:local-lazy</code> , or a fixnum. The default value is <code>nil</code> on Darwin, and <code>t</code> on other platforms.
Values	<i>name</i>	The <i>name</i> argument.
Description		<p>The function <code>register-module</code> explicitly informs LispWorks of the presence of a DLL or shared object file, referred to here as a dynamic library. Functions such as <code>make-pointer</code> and <code>define-foreign-function</code> have a <i>module</i> keyword which can be used to specify which module the function refers to.</p> <p>The main use of modules is to overcome ambiguities that can arise when two different dynamic libraries have functions with the same name.</p> <p>If an application is delivered after calling <code>register-module</code>, then the application attempts to reload the module on startup but does not report any errors. Therefore it is strongly recommended that you call <code>register-module</code> during initialization of your application, rather than at compile time or build time. Loading the module at run time allows you to:</p> <ul style="list-style-type: none"> • 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 *real-name* should also be passed to provide a filename. If *real-name* is not specified then *name* must be a string and 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 path-name such as

```
#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, `.a` on AIX and `.dylib` on Macintosh.

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.
4. The Windows directory (as specified by `GetWindowsDirectory`).

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, the search is conducted in this order:

1. Directories on the user's `LD_LIBRARY_PATH` environment variable.
2. The list of libraries specified in `/etc/ld.so.cache`.
3. `/usr/lib`, followed by `/lib`.

On Mac OS X, 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`.

Note: on LispWorks for SPARC Solaris this value `:manual` for *connection-style* is not supported.

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. The only supported value of *lifetime* in LispWorks for UNIX is `:indefinite`.

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: In LispWorks for UNIX the loader function `link-load:read-foreign-modules` is now deprecated in favor of `register-module`.

Note: static libraries are not supported except on UNIX. For example, on Linux evaluating this form:

```
(fli:register-module "libc.a"
                    :real-name "/usr/lib/libc.a"
                    :connection-style :immediate)
```

results in an error:

```
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:

```
(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 Unix-based systems. It controls the value that is passed to `dlopen` as second argument when the module is connected, and on Darwin it also controls whether `dlopen` is used at all.

The keyword values of `dlopen-flags` correspond to combinations of `RTLD_*` constants (see `/usr/include/dlfcn.h`). The value `t` means the same as `:local-lazy`. The value `nil` means the same as `t` except on Darwin. On Darwin the value `nil` means do not use `dlopen`, and use the older interfaces instead.

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.

The default value of `dlopen-flags` is `nil` on Darwin, because it seems `dlopen` does not work properly on this platform.

Notes

1. It is strongly recommended that you call `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 "Compiling and Loading Foreign Code with the Editor" on page 232

Example

In the following example on Windows, the `user32` DLL is registered, and then a foreign function called `set-cursor-pos` is defined to explicitly reference the `SetCursorPos` function in the `user32` DLL.

```
(fli:register-module :user-dll :real-name "user32")
```

```
(fli:define-foreign-function (set-cursor-pos
                             "SetCursorPos")
  ((x :long)
   (y :long))
  :module :user-dll)
```

This example on Linux loads the shared library even though its symbols are not yet needed. An error is signalled if loading fails:

```
(fli:register-module "libX11.so"
                    :connection-style :immediate)
```

This example loads a module from the same directory as the Lisp executable, by executing this code at runtime:

```
(fli:register-module
  modulename
  :file-name
  (merge-pathnames "modulefilename.dylib"
    (lisp-image-name)))
```

In this last example a program which runs in both 32-bit LispWorks and 64-bit LispWorks loads the correct library for each architecture:

```
(fli:register-module #+:lispworks-32bit "mylib32"
                    #+:lispworks-64bit "mylib64")
```

See also

“Incorporating a foreign module into a LispWorks image” on page 60

```
connected-module-pathname
define-foreign-function
make-pointer
module-unresolved-symbols
print-foreign-modules
```

replace-foreign-array

Function

Summary

Copies the contents of one foreign or Lisp array into another.

Package	<code>ffi</code>	
Signature	<code>replace-foreign-array to from &key start1 start2 end1 end2 allow-sign-mismatch => to</code>	
Arguments	<i>to</i>	A foreign array, foreign pointer or a Lisp array.
	<i>from</i>	A foreign array, foreign pointer or a Lisp array.
	<i>start1, start2, end1, end2</i>	Integers.
	<i>allow-sign-mismatch</i>	A boolean, default value <code>nil</code> .
Values	<i>to</i>	A foreign array, foreign pointer or a Lisp array.
Description	<p>The function <code>replace-foreign-array</code> copies the contents of the array specified by <i>from</i> into another array specified by <i>to</i>. The arrays element types must have the same size and both be either signed or unsigned. When <i>allow-sign-mismatch</i> is <code>nil</code> (the default), the array element types must also match for sign, that is they must be either both signed or both unsigned. When <i>allow-sign-mismatch</i> is non-<code>nil</code>, the array element types do not need to match.</p> <p>The argument <i>to</i> is destructively modified by copying successive elements into it from <i>from</i>. Elements of the subsequence of <i>from</i> bounded by <i>start2</i> and <i>end2</i> are copied into the subsequence of <i>to</i> bounded by <i>start1</i> and <i>end1</i>. 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.</p> <p>Each of <i>to</i> and <i>from</i> can be one of the following:</p>	

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 `c1:base-char`, `lw:bmp-char`, `c1:single-float` and `c1: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 `c1: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 `c1:character`, and thus arrays of `lw:simple-char` (that is, `lw:text-string`) cannot be used in `replace-foreign-array`.

Example

This example demonstrates copying from a foreign pointer to a Lisp array.

An initial array filled with 42:

```
(setq lisp-array
      (make-array 10
                  :element-type '(unsigned-byte 8)
                  :initial-element 42))
```

A foreign pointer to 10 consecutive unsigned chars:

```
(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:

```
(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:

```
(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:

```
(fli:replace-foreign-array
 lisp-array foreign-array :start1 7)
=>
#(42 42 42 6 7 8 42 1 2 3)
```

See also

```
allocate-foreign-object
copy-pointer
```

```
make-pointer
replace-foreign-object
```

replace-foreign-object

Function

Summary	Copies the contents of one foreign object into another.	
Package	fli	
Signature	<code>replace-foreign-object to from &key nelems => pointer</code>	
Arguments	<i>to</i>	A foreign object or a pointer to a foreign object.
	<i>from</i>	A foreign object or a pointer to a foreign object.
	<i>nelems</i>	An integer.
Values	<i>pointer</i>	A pointer to the object specified by <i>from</i> .
Description	The function <code>replace-foreign-object</code> copies the contents of the foreign object specified by <i>from</i> into another foreign object specified by <i>to</i> . Block copying on an array of elements can also be performed by specifying the number of elements to copy using the <i>nelems</i> argument.	
Example	In the following object two sets of ten integers are defined. The object <code>from-obj</code> contains the integers from 0 to 9. The object <code>to-obj</code> contains random values. The <code>replace-foreign-object</code> function is then used to copy the contents of <code>from-obj</code> into <code>to-obj</code> .	

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

```
(setf to-obj
      (fli:allocate-foreign-object
        :type :int
        :nelems 10 ))

(fli:replace-foreign-object to-obj from-obj :nelems 10)
```

See also “Modifying a string in a C function” on page 41

```
allocate-foreign-object
fill-foreign-object
copy-pointer
make-pointer
replace-foreign-array
```

set-locale*Function*

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 This function 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. The *locale* argument 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.

Example On a Windows system:

```
(fli:set-locale "English_UK")
=>
"English_United Kingdom.1252"
```

On a Linux system:

```
(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 `set-locale-encodings mb wc => mb`

Arguments *mb* An external format specification.

wc An external format specification, or `nil`.

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.

`set-locale` calls `set-locale-encodings` after successfully setting the C locale.

See also `convert-from-foreign-string`
`convert-to-foreign-string`
`:ef-mb-string`
`:ef-wc-string`
`set-locale`
`with-foreign-string`

setup-embedded-module*Function*

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.
2. The effect of `setup-embedded-module` persists after `save-image` and `deliver`.
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`
“Incorporating a foreign module into a LispWorks image” on page 60

size-of *Function*

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*.

Example 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 [Chapter 2, “FLI Types”](#)
[allocate-foreign-object](#)
[free-foreign-object](#)

start-collecting-template-info

Function

Summary Nullifies the FLI Template information in the image.

Package `fli`

Signature `start-collecting-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-collecting-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 *LispWorks Delivery User Guide* for further details.

See also `print-collected-template-info`

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 `with-coerced-pointer (coerced-pointer &key type pointer-type)
pointer &body body => last`

Arguments	<i>coerced-pointer</i>	A variable bound to a copy of <i>pointer</i> .
	<i>type</i>	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.
	<i>pointer-type</i>	The pointer type of the temporary pointer.
	<i>pointer</i>	A FLI pointer of which a copy is made. The lifetime of the copy is across the scope of the <code>with-coerced-pointer</code> macro.
	<i>body</i>	A list of forms to be executed across the scope of the temporary pointer binding.

Values	<i>last</i>	The value of the last form in <i>body</i> .
--------	-------------	---

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 *body* with *coerced-pointer* bound to a dynamic-extent copy of the FLI pointer *pointer*.

coerced-pointer points to the same foreign object as *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 `decf-pointer`, without affecting *pointer*.

Note that *coerced-pointer* has dynamic-extent, so you should not use it after returning from *body*.

Example 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 “An example of dynamic pointer allocation” on page 27
`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*)
body ::= *form**

Arguments *var* A symbol to be bound to a foreign type.

	<i>foreign-type</i>	A foreign type descriptor to be bound to the variable <i>var</i> .
	<i>form</i>	A form to be executed with <i>bindings</i> in effect.
Values	<i>last</i>	The value of the last <i>form</i> in <i>body</i> .
Description	<p>The macro <code>with-dynamic-foreign-objects</code> binds variables according to the list <i>bindings</i>, and then executes <i>body</i>. Each element of <i>bindings</i> is a list which binds a symbol to a pointer which points to a locally allocated instance of a foreign type.</p> <p><i>initial-element</i>, <i>initial-contents</i>, <i>fill</i>, <i>nelems</i> and <i>size-slot</i> initialize the allocated instance as if by <code>allocate-foreign-object</code>.</p> <p>The lifetime of the bound foreign objects, and hence the allocation of the memory they take up, is within the scope of the <code>with-dynamic-foreign-objects</code> function.</p> <p>Any object created with <code>allocate-dynamic-foreign-object</code> within <i>body</i> will automatically be deallocated once the scope of the <code>with-dynamic-foreign-objects</code> function has been left.</p>	
Compatibility note	<p>There is an alternative syntax for <i>binding</i> with an optional <i>initial-element</i> which is the only way to supply an initial element in LispWorks 5.0 and previous versions. Like this:</p> <pre><i>binding</i> ::= (var <i>foreign-type</i> &optional <i>initial-element</i>)</pre> <p>This alternative syntax is deprecated in favor of the keyword syntax for <i>binding</i> defined in “Signature” above which is supported in LispWorks 5.1 and later.</p>	
Example	<p>This example shows the use of <code>with-dynamic-foreign-objects</code> with an implicitly created pointer.</p> <p>Windows version:</p>	

```

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;
};

```

Unix/Linux/Macintosh version:

```

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 “An example of dynamic memory allocation” on page 7.

See also “Modifying a string in a C function” on page 41
`allocate-dynamic-foreign-object`
`free-foreign-object`
`with-coerced-pointer`

with-dynamic-lisp-array-pointer*Macro*

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

<i>pointer-var</i>	A variable to be bound to the foreign pointer.
<i>lisp-array</i>	A static Lisp array (a string or a byte/single-float/double-float array).
<i>start</i>	An index into the Lisp array.
<i>type</i>	A foreign type. The default is <code>:void</code> .
<i>body</i>	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, 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.

Example

```
(let ((vector
      (make-array 3 :element-type '(unsigned-byte 8)
                  :initial-contents '(65 77 23)
                  :allocation :static)))
    (fli:with-dynamic-lisp-array-pointer
      (ptr vector :start 1 :type '(:unsigned :byte))
      (fli:dereference ptr)))
=>
77
```

See also

```
:lisp-array
:lisp-simple-1d-array
```

with-foreign-block

Macro

Summary

Allocates a foreign block, executes code and frees the block, in LispWorks for Macintosh.

Package

```
fli
```

Signature	<code>with-foreign-block</code> (<i>foreign-block-var</i> <i>type</i> <i>function</i> &rest <i>extra-args</i>) &body <i>body</i> => <i>results</i>
Arguments	<p><i>foreign-block-var</i> A symbol</p> <p><i>type</i> A symbol naming a foreign block type defined using <code>define-foreign-block-callable-type</code>.</p> <p><i>extra-args</i> Arguments.</p>
Values	<i>results</i> The results of <i>body</i> .
Description	<p>The macro <code>with-foreign-block</code> allocates a foreign block using <i>type</i>, <i>function</i> and <i>extra-args</i> in the same way as <code>allocate-foreign-block</code>. It then binds <i>foreign-block-var</i> to the foreign block, execute the code of <i>body</i> and frees the foreign block using <code>free-foreign-block</code>, using <code>unwind-protect</code>.</p> <p><code>with-foreign-block</code> is a convenient way to ensure that you do not forget to free the foreign block.</p>
Notes	<p>If the foreign block is copied in the code of <i>body</i>, the copy may be invoked, and hence the function called, after exiting this macro. See the discussion in “Scope of invocation” on page 62.</p> <p><code>with-foreign-block</code> returns the results of <i>body</i>.</p> <p><code>with-foreign-block</code> is implemented in LispWorks for Macintosh only.</p>
See also	<p><code>allocate-foreign-block</code></p> <p><code>free-foreign-block</code></p> <p><code>with-local-foreign-block</code></p> <p>“Block objects in C (foreign blocks)” on page 60</p>

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 *variable-name* A symbol
slot-name A symbol
object-type A FLI structure type
form A form evaluating to an instance of (or a pointer to) a FLI structure
body Forms to be executed

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 the *copy-foreign-object* option 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*.

Example

```
(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 “Structures and unions” on page 15
`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.

Package `fli`

Signature	<code>with-foreign-string</code> (<i>pointer</i> <i>element-count</i> <i>byte-count</i> &key <i>external-format</i> <i>null-terminated-p</i> <i>allow-null</i>) <i>string</i> &body <i>body</i> => <i>last</i>	
	<i>body</i> ::= <i>form</i> *	
Arguments	<i>pointer</i>	A symbol bound to a pointer to the foreign string.
	<i>element-count</i>	A symbol bound to the number of elements in the foreign string.
	<i>byte-count</i>	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 <i>byte-count</i> will be the same as <i>element-count</i> .
	<i>external-format</i>	An external format specification.
	<i>null-terminated-p</i>	If <i>t</i> , the foreign string is terminated by a null character. The null character is included in the <i>element-count</i> of the string.
	<i>allow-null</i>	A boolean. The default is <code>nil</code> .
	<i>string</i>	The Lisp string to convert.
	<i>body</i>	A list of forms to be executed.
	<i>form</i>	A form to be executed.
Values	<i>last</i>	The value of the last <i>form</i> in <i>body</i> .
Description	The macro <code>with-foreign-string</code> 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 <i>string</i> , a Lisp string, into a foreign string. The symbol <i>pointer</i> is bound to a pointer to the start of the string, the symbol <i>element-count</i> is set equal to the number of elements in the string, and the symbol <i>byte-count</i> is set equal to the number of bytes the string occupies. Then the list of forms specified by <i>body</i> is executed. Finally, the memory allocated for the foreign string and pointer is de-allocated.	

external-format is used to specify the encoding of the foreign string. It defaults to a format appropriate for C string of type `char*`. For Unicode encoded strings, specify `:unicode`. If you want to pass a string to the Win32 API, known as `STR` in the Win32 API terminology, specify `*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`. The names of available external formats are listed in the section "External formats" 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 the string terminates in a null character, it is included in the *element-count*.

If *allow-null* is non-nil, then if *string* is `nil` a null pointer is passed.

See also "Modifying a string in a C function" on page 41
 "Passing a string to a Windows function" on page 39
 Section "External formats" in the *LispWorks User Guide and Reference Manual*

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

with-integer-bytes

Macro

Summary Converts a Lisp integer to foreign bytes while executing a body of code.

Package `ffi`

Signature `with-integer-bytes (pointer length) integer &body body => last`

Arguments	<i>pointer</i>	A variable to be bound to the foreign pointer.
	<i>length</i>	A variable to be bound to the length in bytes.
	<i>integer</i>	An integer.
	<i>body</i>	Forms to be executed.
Values	<i>last</i>	The value of the last form in <i>body</i> .
Description	The macro <code>with-integer-bytes</code> evaluates the forms in <i>body</i> with <i>pointer</i> bound to a dynamic foreign object containing the bytes of <i>integer</i> and <i>length</i> 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 <i>integer</i> by calling <code>make-integer-from-bytes</code> .	
See also	“Lisp integers” on page 57 <code>convert-integer-to-dynamic-foreign-object</code> <code>make-integer-from-bytes</code>	

with-local-foreign-block *Macro*

Summary	Allocates a foreign block, executes code and frees the block, in LispWorks for Macintosh.	
Package	<code>fli</code>	
Signature	<code>with-local-foreign-block</code> (<i>foreign-block-var</i> <i>type</i> <i>function</i> &rest <i>extra-args</i>) &body <i>body</i> => <i>results</i>	
Arguments	<i>foreign-block-var</i>	A symbol.
	<i>type</i>	A symbol naming a foreign block type defined using <code>define-foreign-block-callable-type</code> .

	<code>extra-args</code>	Arguments.
Values	<code>results</code>	The results of <i>body</i> .
Description	<p>The macro <code>with-local-foreign-block</code> allocates a foreign block using <i>type</i>, <i>function</i> and <i>extra-args</i> in the same way as <code>allocate-foreign-block</code>, but with dynamic extent. It then binds <i>foreign-block-var</i> to the foreign block and executes the code of <i>body</i>.</p> <p><code>with-local-foreign-block</code> can be used only if the code in <i>body</i> can be guaranteed not to invoke the block or a copy of it either outside the scope of <code>with-local-foreign-block</code> or in another thread. Unless you can be sure of that, you need to use <code>with-foreign-block</code>.</p> <p><code>with-local-foreign-block</code> returns the results of <i>body</i>.</p> <p><code>with-local-foreign-block</code> can be a little faster than <code>with-foreign-block</code>.</p>	
Notes	<code>with-local-foreign-block</code> is implemented in LispWorks for Macintosh only.	
See also	<code>allocate-foreign-block</code> <code>free-foreign-block</code> <code>with-foreign-block</code> “Block objects in C (foreign blocks)” on page 60	

8

Type Reference

:boolean

FLI type descriptor

Summary

Converts between a Lisp boolean value and a C representation of a boolean value.

Syntax

`:boolean &optional encapsulates`

Arguments

encapsulates An integral type.

Description

The FLI `:boolean` type converts between a Lisp boolean value and a C representation of a boolean value. The *encapsulates* option is used to specify the size of the value from which the boolean value is obtained. 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.

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.

Example

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`
“Boolean types” on page 13

:byte *FLI type descriptor*

Summary Converts between a Lisp integer with a C `signed char`.

Syntax `:byte`

Arguments None.

Description The FLI `:byte` type converts between a Lisp integer type and a C `signed char` type.

See also `:char`
`:short`
“Integral types” on page 13

:c-array *FLI type descriptor*

Summary Converts between a FLI array and a C array type.

Syntax `:c-array type &rest dimensions`

Arguments *type* The type of the elements of the array.
dimensions A sequence of the dimensions of the new array.

Description The FLI `:c-array` type 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.

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.

Example

The following code defines a 3 by 3 array of integers.

```
(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.

```
(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`
 “Arrays” on page 14

:char *FLI type descriptor*

Summary Converts between a Lisp `character` type and a C `char` type.

Syntax `:char`

Arguments None.

Description The FLI `:char` type 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`
 “Character types” on page 13

:const

FLI type descriptor

Summary	Corresponds to the C <code>const</code> type.
Syntax	<code>:const</code> &optional <i>type</i>
Arguments	<i>type</i> The type of the constant. The default is <code>:int</code> .
Description	The FLI <code>:const</code> type corresponds to the C <code>const</code> type qualifier. The behavior of a <code>:const</code> is exactly the same as the behavior of its <i>type</i> , and it is only included to ease the readability of FLI code and for naming conventions.
Example	In the following example a constant is allocated and set equal to 3.141. <pre>(setq pi1 (fli:allocate-foreign-object :type '(:const :float))) (setf (fli:dereference pi1) 3.141))</pre>
See also	<code>:volatile</code> “Immediate types” on page 12

:double

FLI type descriptor

Summary	Converts a Lisp double float to a C <code>double</code> .
Syntax	<code>:double</code>
Arguments	None.
Description	The FLI <code>:double</code> type converts between a Lisp double float and the C <code>double</code> type.
Compatibility Note	In LispWorks 4.4 and previous on Windows and Linux platforms, all Lisp floats are doubles. In LispWorks 7.0, there are three disjoint Lisp float types, on all platforms.

See also `:float`
 “Floating point types” on page 13

`:ef-mb-string` *FLI type descriptor*

Summary Converts between a Lisp string and a C multi-byte string.

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 `:ef-mb-string` type converts between a Lisp string and a C multi-byte string. The C string may have a maximum length of *limit* bytes. The *limit* can be omitted in cases where a new foreign string is being allocated.

The *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 `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`

“Strings” on page 14

`:ef-wc-string`

FLI type descriptor

Summary Converts between a Lisp string and a C wide-character string.

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 `:ef-wc-string` type converts between a Lisp string and a C wide-character string. The C string may have a maximum length of *limit* characters. The *limit* can be omitted in cases where a new foreign string is being allocated.

The *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`
 “Strings” on page 14

:enum *FLI type descriptor*

Summary Converts between a Lisp symbol and a C `enum`.

Syntax `:enum &rest enum-constants`
`enum-constants ::= {symbol | (symbol value)}*`

Arguments

- `enum-constants` A sequence of one or more symbols naming the elements of the enumeration.
- `symbol` A symbol naming an element of the enumeration.
- `value` An integer specifying the value of `symbol`.

Description The FLI `:enum` type converts between a Lisp symbol and the C `enum` type. Each entry in the `enum-constants` can either consist of a symbol, in which case the first entry has a value 0, or of a list of a symbol and its corresponding integer value.

Example See `define-c-enum`, page 83, for an example using the `:enum` type.

See also `define-c-enum`
 “Integral types” on page 13

:enumeration

FLI type descriptor

Summary	A synonym for <code>:enum</code>
Syntax	<code>:enumeration &rest <i>enum-constants</i></code>
Description	The FLI <code>:enumeration</code> type is the same as the FLI <code>:enum</code> type.
See also	<code>:enum</code> “Integral types” on page 13

:fixnum

FLI type descriptor

Summary	Converts between a Lisp fixnum and a 32 bit raw integer.
Syntax	<code>:fixnum</code>
Arguments	None.
Description	The FLI <code>:fixnum</code> type converts between a Lisp fixnum and a 32 bit integer in C.
See also	“Integral types” on page 13

:float

FLI type descriptor

Summary	Converts a Lisp single float to a C <code>float</code> .
Syntax	<code>:float</code>
Arguments	None.
Description	The FLI <code>:float</code> type converts between a Lisp single float and the C <code>float</code> type.

Compatibility note In LispWorks 4.4 and previous on Windows and Linux platforms, all Lisp floats are doubles. In LispWorks 7.0, there are three disjoint Lisp float types, on all platforms.

See also `:double`
 “Floating point types” on page 13

:foreign-array *FLI type descriptor*

Summary Converts between a FLI array and a foreign array type.

Syntax `:foreign-array type dimensions`

Arguments *type* The type of the elements of the array.
dimensions A list containing the dimensions of the array.

Description The FLI `: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.

Example The following code defines a 3 by 4 foreign array with elements of type `:byte`.

```
(setq farray (fli:allocate-foreign-object
              :type '(:foreign-array :byte (3 4))))
```

The type of this is equivalent to the C declaration

```
signed char array2[3][4];
```

See also `:c-array`
 `foreign-aref`
 `foreign-array-pointer`
 “Arrays” on page 14

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`

Arguments None

Description The foreign 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 “Block objects in C (foreign blocks)” on page 60.

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`
 “Block objects in C (foreign blocks)” on page 60

:function

FLI type descriptor

Summary

Converts between Lisp and the C `function` type.

Syntax

`:function` &optional *args-spec* *return-spec* &key *calling-convention*

Arguments

args-spec A list of function argument types.

return-spec A list of function return types.

calling-convention

One of `:stdcall` or `:cdecl`.

Description

The FLI `:function` type allows for conversion from the C `function` type. It is typically used in conjunction with the `:pointer` type to reference an existing foreign function.

The default value of *calling-convention* is as described for `define-foreign-function`.

Example

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 * )
```

See also `:pointer`

:int *FLI type descriptor*

Summary Converts between a Lisp integer and a C `int` type.

Syntax `:int`

Arguments None.

Description The `:int` type converts between an Lisp integer and a C `int` type. It is equivalent to the `:signed` and `(:signed :int)` types.

See also `:signed`
“Integral types” on page 13

:int8

:int16

:int32

:int64

:intmax

:intptr

FLI type descriptors

Summary The signed sized integer types.

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:

<code>:int8</code>	8-bit signed integer
<code>:int16</code>	16-bit signed integer
<code>:int32</code>	32-bit signed integer
<code>:int64</code>	64-bit signed integer
<code>:intmax</code>	The largest type of signed integer available
<code>:intptr</code>	A signed integer the same size as a pointer

See also `:uint8`
 “Integral types” on page 13

`:lisp-array` *FLI type descriptor*

Summary A foreign type which passes the address of a Lisp array direct to C.

Syntax `:lisp-array &optional type`

Arguments *type* A list. The default is `nil`.

Description `:lisp-array` is a foreign type which 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.

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.

Example

This C function fills an array of doubles from an array of single floats.

Windows version:

```
__declspec(dllexport) void __cdecl ProcessFloats(int
count, float * fvec, double * dvec)
{
    for(--count ; count >= 0 ; count--) {
        dvec[count] = fvec[count] * fvec[count];
    }
}
```

Linux/Unix/Macintosh version:

```
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`:

```

(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

```
(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
single-array[2] = 2.0; double-array[2] = 4.0

```

See also

```

:lisp-simple-1d-array
with-dynamic-lisp-array-pointer

```

:lisp-double-float

FLI type descriptor

Summary

A synonym for `:double`.

Syntax `:lisp-double-float`

Description The FLI `:lisp-double-float` type is the same as the FLI `:double` type.

See also `:double`
“Floating point types” on page 13

:lisp-float *FLI type descriptor*

Summary Converts between any Lisp float and the C `double` type or the C `float` type.

Syntax `:lisp-float &optional float-type`
`float-type ::= :single | :double`

Arguments *float-type* Determines the C type to convert to. The default is `:single`.

Description The FLI `:lisp-float` type 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`
“Floating point types” on page 13

:lisp-simple-1d-array *FLI type descriptor*

Summary A foreign type which passes the address of a Lisp simple vector direct to C.

Syntax `:lisp-simple-1d-array &optional type`

Arguments	<i>type</i>	A list. The default is <code>nil</code> .
Description	<code>:lisp-simple-1d-array</code>	is a foreign type which 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. The Lisp vector as subject to the same memory management restrictions as the array passed with <code>:lisp-array</code> . It must be statically allocated, and may need to be retained explicitly in Lisp. The argument <i>type</i> , if non-nil, is a list (<i>element-type</i> & <i>rest dimensions</i>) and is used to check the element type and dimensions of the Lisp array passed.
See also	<code>:lisp-array</code> <code>with-dynamic-lisp-array-pointer</code>	

`:lisp-single-float` *FLI type descriptor*

Summary	A synonym for <code>:float</code> .
Syntax	<code>:lisp-single-float</code>
Description	The FLI <code>:lisp-single-float</code> type is the same as the FLI <code>:float</code> type.
See also	<code>:float</code> “Floating point types” on page 13

`:long` *FLI type descriptor*

Summary	Converts between a Lisp integer and a C long.
---------	---

Syntax `:long &optional integer-type`
 `integer-type ::= :int | :double | :long`

Arguments `integer-type` One of `:int`, `:double`, or `:long`.

Description The FLI `:long` type converts between the Lisp `integer` type and the C `long` type. See Table 8.1 for comparisons between Lisp and C long types.

Table 8.1 A comparison between Lisp and C long types

Lisp type	FLI type	C type
<code>integer</code>	<code>:long</code>	<code>long</code>
<code>integer</code>	<code>:long :int</code>	<code>long</code>
<code>integer</code>	<code>:long :double</code>	<code>long double</code>
<code>integer</code>	<code>:long :long</code> <code>:long-long</code>	<code>long long</code>

See also `:int`
 `:long-long`
 `:short`
 “Integral types” on page 13

`:long-long` *FLI type descriptor*

Summary Converts between a Lisp `integer` and a signed C `long long`.

Syntax `:long-long`

Arguments None.

Description The FLI `:long-long` type converts between the Lisp `integer` type and the C `long long` type.

method for determining the type of `thing`, and therefore the first element in the list of `:one-of` is used.

```
(fli:dereference thing)
```

See also `:union`

:pointer *FLI type descriptor*

Summary Defines a C-style FLI pointer to an object of a specified type.

Syntax `:pointer type`

Arguments `type` The type of FLI object pointed to by the pointer.

Description The FLI `:pointer` type 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)

For more details on pointers, including examples on pointer coercion, dereferencing, making, and copying see Chapter 3, “FLI Pointers”.

See also `copy-pointer`
`dereference`
`make-pointer`
`*null-pointer*`
“Pointer types” on page 14

:ptr *FLI type descriptor*

Summary A synonym for `:pointer`.

Syntax `:ptr type`

Description The FLI `:ptr` type is the same as the FLI `:pointer` type.

See also `:pointer`

`:ptrdiff-t` *FLI type descriptor*

Summary Converts between a Lisp integer and an ISO C `ptrdiff_t`.

Syntax `:ptrdiff-t`

Arguments None.

Description The FLI `:ptrdiff-t` type 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.

Syntax `:reference type &key allow-null lisp-to-foreign-p foreign-to-lisp-p`

Arguments *type* The type of the object to pass by reference.

allow-null If non-`nil`, if the input argument is `nil` a null pointer is passed instead of a reference to an object containing `nil`.

lisp-to-foreign-p If non-`nil`, allow conversion from Lisp to the foreign language. The default value is `t`.

foreign-to-lisp-p If non-`nil`, allow conversion from the foreign language to Lisp. The default value is `t`.

Description The FLI `:reference` type 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`.

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.

Example 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.

Syntax `:reference-pass type &key allow-null`

Arguments

<i>type</i>	The type of the object to pass by reference.
<i>allow-null</i>	If non- <code>nil</code> , if the input argument is <code>nil</code> a null pointer is passed instead of a reference to an object containing <code>nil</code> .

Description The FLI type `:reference-pass` is equivalent to:

```
(:reference :lisp-to-foreign-p t
           :foreign-to-lisp-p nil)
```

See `:reference` for the details.

See also `:reference`
`:reference-return`

`:reference-return`

FLI type descriptor

Summary Passes an object from the foreign language to Lisp by reference.

Syntax `:reference-return type &key allow-null`

Arguments	<i>type</i>	The type of the object to return by reference.
	<i>allow-null</i>	If non-nil, if the input argument is <code>nil</code> a null pointer is passed instead of a reference to an object containing <code>nil</code> .
Description	The FLI type <code>:reference-return</code> is equivalent to: <pre>(:reference :lisp-to-foreign-p nil :foreign-to-lisp-p t)</pre> See <code>:reference</code> for the details.	
See also	<code>:reference</code> <code>:reference-pass</code>	

released-foreign-block-pointer

FLI type descriptor

Summary	The type of foreign blocks that have been released.	
Package	<code>fli</code>	
Syntax	<code>released-foreign-block-pointer</code>	
Description	The FLI type <code>released-foreign-block-pointer</code> is the type of released foreign blocks. The system marks foreign blocks that have been released by <code>foreign-block-release</code> as being of foreign type <code>released-foreign-block-pointer</code> .	
See also	<code>foreign-block-pointer</code> <code>foreign-block-release</code>	

:short

FLI type descriptor

Summary	Converts between a Lisp <code>fixnum</code> type and a C <code>short</code> type.	
---------	---	--

Syntax	<code>:short &optional <i>integer-type</i></code> <code><i>integer-type</i> ::= :int</code>
Arguments	<code><i>integer-type</i></code> If specified, must be <code>:int</code> , which associates a Lisp <code>fixnum</code> with a C <code>int</code> .
Description	The FLI <code>:short</code> type associates a Lisp <code>fixnum</code> with a C <code>short</code> . The FLI types <code>:short</code> , <code>(:short :int)</code> , <code>(:signed :short)</code> , and <code>(:signed :short :int)</code> are equivalent.
See also	<code>:int</code> <code>:signed</code> “Integral types” on page 13

:signed *FLI type descriptor*

Summary	Converts between a Lisp integer and a foreign signed integer.
Syntax	<code>:signed &optional <i>integer-type</i></code> <code><i>integer-type</i> ::= :byte :char :short :int :long :long :int :short :int</code>
Arguments	<code><i>integer-type</i></code> The type of the signed integer.

Description

The `:signed` type converts between a Lisp integer and a foreign signed integer. The optional `integer-type` argument specifies other kinds of signed integer types. See Table 8.2 for a comparison between Lisp and C signed types.

Table 8.2 A comparison of Lisp and C signed types

Lisp type	FLI type	C type
<code>integer</code>	<code>:signed</code>	<code>signed int</code>
<code>fixnum</code>	<code>:signed :byte</code>	<code>signed char</code>
<code>fixnum</code>	<code>:signed :char</code>	<code>signed char</code>

Table 8.2 A comparison of Lisp and C signed types

Lisp type	FLI type	C type
<code>fixnum</code>	<code>:signed :short</code>	<code>signed short</code>
<code>integer</code>	<code>:signed :int</code>	<code>signed int</code>
<code>integer</code>	<code>:signed :long</code>	<code>signed long</code>
<code>fixnum</code>	<code>:signed :short :int</code>	<code>signed short</code>
<code>integer</code>	<code>:signed :long :int</code>	<code>signed long</code>

See also `cast-integer`
`:unsigned`
 “Integral types” on page 13

`:size-t` *FLI type descriptor*

Summary Converts between a Lisp integer and an ISO C `size_t`.

Syntax `:size-t`

Arguments None.

Description The FLI `:size-t` type 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 `:ssize-t`

`:ssize-t` *FLI type descriptor*

Summary Converts between a Lisp integer and the platform-specific `ssize_t` type.

Syntax `:ssize-t`

Arguments	None.
Description	The FLI <code>:ssize-t</code> type converts between a Lisp integer and a platform-specific <code>ssize_t</code> type, which is a signed integer representing the size of an object in bytes.
See also	<code>:size-t</code>

:struct *FLI type descriptor*

Summary	Converts between a FLI structure and a C <code>struct</code> .	
Syntax	<pre><code>:struct &rest <i>slots</i> <i>slots</i> ::= {<i>symbol</i> (<i>symbol</i> <i>slot-type</i>)*} <i>slot-type</i> ::= <i>type</i> (:bit-field <i>integer-type</i> <i>size</i>)</code></pre>	
Arguments	<i>slots</i>	A sequence of one or more slots making up the structure.
	<i>symbol</i>	A symbol naming the slot.
	<i>type</i>	The slot type. If no type is given it defaults to an <code>:int</code> .
	<i>integer-type</i>	An integer type. Only <code>:int</code> , <code>(:unsigned :int)</code> and <code>(:signed :int)</code> are guaranteed to work on all platforms.
	<i>size</i>	An integer specifying a number of bits for the field.
Description	The FLI <code>:struct</code> type is an aggregate type, and converts between a FLI structure and a C <code>struct</code> type. The FLI structure consists of a collection of one or more slots. Each slot has a name and a type. A structure can also contain bit fields, which are integers with a specified number of 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.

Example

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)
```

See also

```
define-c-struct  
foreign-slot-names  
foreign-slot-offset  
foreign-slot-pointer  
foreign-slot-type  
foreign-slot-value  
“Structures and unions” on page 15
```

:time-t

FLI type descriptor

Summary

Converts between a Lisp integer and the platform-specific `time_t` type.

Syntax

```
:time-t
```

Arguments

None.

Description The FLI `:time-t` type 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.

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:

<code>:uint8</code>	8-bit unsigned integer
<code>:uint16</code>	16-bit unsigned integer
<code>:uint32</code>	32-bit unsigned integer
<code>:uint64</code>	64-bit unsigned integer
<code>:uintmax</code>	The largest type of unsigned integer available
<code>:uintptr</code>	An unsigned integer the same size as a pointer

See also `:int8`
 “Integral types” on page 13

:union

FLI type descriptor

Summary

Converts between a FLI union and a C union type.

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 `:union` type 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 and a 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.

Example

In the following example a union type with two slots is defined.

```
(fli:define-c-union my-number  
                  (small :byte) (large :int))
```

An instance of the union is allocated and bound to the Lisp variable `length`.

```
(setq length  
      (fli:allocate-foreign-object :type 'my-number))
```

Finally, the `small` slot of the union is set equal to 24.

```
(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
```

“Structures and unions” on page 15

:unsigned*FLI type descriptor*

Summary

Converts between a Lisp integer and a foreign unsigned integer.

Syntax

```
:unsigned &optional integer-type
integer-type ::= :byte | :char | :short | :int |
                  :long | :long :int | :short :int
```

Arguments

integer-type The type of the unsigned integer.

Description

The **:unsigned** type converts between a Lisp integer and a foreign unsigned integer. The optional *integer-type* argument specifies other kinds of unsigned integer types. See Table 8.3 for a comparison between Lisp and C unsigned types.

Table 8.3 A comparison of Lisp and C unsigned types

Lisp type	FLI type	C type
integer	:unsigned	unsigned int
fixnum	:unsigned :byte	unsigned char
fixnum	:unsigned :char	unsigned char
fixnum	:unsigned :short	unsigned short
integer	:unsigned :int	unsigned int
integer	:unsigned :long	unsigned long
fixnum	:unsigned :short :int	unsigned short

Table 8.3 A comparison of Lisp and C unsigned types

Lisp type	FLI type	C type
<code>integer</code>	<code>:unsigned :long</code> <code>:int</code>	<code>unsigned long</code>

See also `cast-integer`
`:signed`
“Integral types” on page 13

:void

FLI type descriptor

Summary Represents the C `void` type.

Syntax `:void`

Arguments None.

Description The FLI `:void` type 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`
“The void type” on page 18

:volatile *FLI type descriptor*

Summary Corresponds to the C `volatile` type.

Syntax `:volatile &optional type`

Arguments `type` The type of the volatile. The default is `:int`.

Description The FLI `:volatile` type 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`.

Syntax `:wchar-t`

Arguments None.

Description The FLI `:wchar-t` type 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.

Syntax	<code>:wrapper</code>	<code>foreign-type</code>	&key	<code>lisp-to-foreign</code>	<code>foreign-to-lisp</code>
Arguments	<code>foreign-type</code>	The underlying type to wrap.			
	<code>lisp-to-foreign</code>	Code specifying how to convert between Lisp and the FLI.			
	<code>foreign-to-lisp</code>	Code specifying how to convert between the FLI and Lisp.			

Description

The FLI `:wrapper` type allows for an extra level of conversion between Lisp and a foreign language through the FLI. With the `:wrapper` type you can specify conversion functions from and to an instance of another 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.

Example

In the following example an `:int` is allocated with a wrapper to allow the `:int` to be accessed as a string.

```
(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.

```
(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:

```
(fli:dereference wrap :type :int)
```

See also “Encapsulated types” on page 16

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 `c1.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 `c1.exe`, download the VC++ toolkit from Microsoft.

Preprocessors known to work with LispWorks are:

- Microsoft Visual Studio's `c1.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 “Modifying a string in a C function” on page 41. The header file `test.h` needs to be slightly different depending on the platform.

Windows version:

```
__declspec(dllexport) void __cdecl modify(char *string)
```

Linux/Unix/Macintosh version:

```
void modify(char *string)
```

1. Load the Foreign Parser:

```
(require "foreign-parser")
```

2. Now generate prototype FLI definitions:


```
(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:

```
(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`:

```
(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

```
(fli:register-module "test.dll")
```

or

```
(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 *LispWorks 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 (usually that given by the value of the environment variable `TEMP`) and deleted after `register-module` is called on it.

9.5 Foreign Parser Reference

preprocessor *Variable*

Package	<code>foreign-parser</code>
Initial Value	<code>"cc"</code> on Unix, Linux and Mac OS X. <code>"cl"</code> on Windows
Description	The variable <code>*preprocessor*</code> provides the default value for the <i>preprocessor</i> used by <code>process-foreign-file</code> .
See also	<code>*preprocessor-options*</code> <code>process-foreign-file</code>

preprocessor-format-string *Variable*

Package	<code>foreign-parser</code>
Initial Value	On Windows: <code>"~A" /nologo /E ~A ~{/D~A ~}~{/I"~A" ~}/Tc "~A"</code> On Unix, Linux and Mac OS X: <code>"~A -E ~A ~{-D~A~ ~}~{-I~A ~}~A"</code>
Description	The variable <code>*preprocessor-format-string*</code> provides the default value for the <i>preprocessor-format-string</i> used by <code>process-foreign-file</code> .

See also `process-foreign-file`

preprocessor-include-path *Variable*

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*

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*

Package `foreign-parser`

Syntax `process-foreign-file source &key dff language preprocess
preprocessor preprocessor-format-string preprocessor-options
preprocessor-include-path case-sensitive package =>`

Arguments	<i>source</i>	One or more filenames.
	<i>dff</i>	A filename.
	<i>language</i>	A keyword.
	<i>preprocess</i>	A boolean.
	<i>preprocessor-format-string</i>	A string.
	<i>preprocessor</i>	A string.
	<i>preprocessor-options</i>	A string.
	<i>include-path</i>	A list.
	<i>case-sensitive</i>	See description.
	<i>package</i>	A package designator or <code>nil</code> .
Description	<p>The <code>process-foreign-file</code> function takes a file or files of foreign declarations — usually header files — and parses them, producing ‘dff’ files of Lisp definitions using <code>define-foreign-function</code>, <code>define-foreign-variable</code>, <code>define-foreign-type</code>, and so on, providing a Lisp interface to the foreign code.</p> <p><i>source</i> gives the name of the header files or file to be processed. The name of a file consists of <i>source-file-name</i> and <i>source-file-type</i> (typically <code>.h</code>).</p> <p><i>dff</i> is an output file which will contain the Lisp foreign function definitions. The default value is <code>nil</code>, in which case the dff file will be <i>source-file-name-dff.lisp</i>. (See <i>source</i>, above.)</p> <p><i>language</i> specifies the language the header files are written in. Currently the supported languages are <code>:c</code> (standard K&R C header files) and <code>:ansi-c</code>. The default value is <code>:ansi-c</code>.</p> <p><i>preprocess</i>, when non-<code>nil</code>, runs the preprocessor on the input files. The default value is <code>t</code>.</p>	

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`.
- `:prefix` — changes lowercase to uppercase and concatenates the string with the string held in `sys:*prefix-name-string*`. For example, `OneTwoTHREE` becomes `FOREIGN-ONETWOTHREE`.

- `(list :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, `nreverse`) 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.

coerced 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 `fl1` package used to interface Lisp with a foreign language.

FLI type

A data type specifier in the `fl1` 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.

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

- aggregate FLI types 12, 14–16
- aggregate types 12
- `align-of` function 67
- `alloca` function 68
- `allocate-dynamic-foreign-object` function 68
- `allocate-foreign-block` function 70
- `allocate-foreign-object` function 71
- allocating memory dynamically 7, 72
- Android 35

B

- Block object 61
- `:boolean` FLI type 193
- `:byte` FLI type 194

C

- C 61
 - calling from Lisp 2, 61, 103
 - calling from Lisp with a block 61
 - calling into Lisp 32, 93
- C++
 - calling from Lisp 2, 61, 103
 - calling from Lisp with a block 61
 - calling into Lisp 32, 93
- C code
 - declarations 229

- calling convention
 - hard-float 35
 - soft-float 35
- `:c-array` FLI type 194
- `cast-integer` function 74
- `:char` FLI type 196
- CLANG 61
- conditions
 - `foreign-type-error` 141
- `connected-module-pathname` function 74
- `:connection-style` argument to `register-module` 164
- `:const` FLI type 197
- `convert-from-foreign-string` function 75
- `convert-integer-to-dynamic-foreign-object` function 77
- `convert-to-dynamic-foreign-string` function 79
- `convert-to-foreign-string` function 77
- `copy-pointer` function 81

D

- `decf-pointer` function 82
- `define-c-enum` macro 83
- `define-c-struct` macro 85
- `define-c-typedef` macro 88
- `define-c-union` macro 89
- `define-foreign-block-callable-type` macro 91
- `define-foreign-block-invoker` macro 92
- `define-foreign-callable`

- macro 31, 35, 93
- define-foreign-converter**
 - macro 98
- define-foreign-forward-reference-type** macro 102
- define-foreign-funcallable**
 - macro 102
- define-foreign-function**
 - macro 31, 103
- define-foreign-pointer**
 - macro 22, 109
- define-foreign-type** macro 11, 110
- define-foreign-variable**
 - macro 112
- define-opaque-pointer**
 - macro 116
- defining FLI functions 6
- defining FLI types 4
- defining forms
 - ambiguity 237
 - automated generation 229
- defsystem** macro 60
- dereference** function 117
- disconnect-module** function 120
- DLLs
 - exporting functions from 58-60
- documentation strings 12
- :double** FLI type 197
- dynamic memory allocation 7

E

- :ef-mb-string** FLI type 198
- :ef-wc-string** FLI type 199
- Embedded dynamic modules 60
- :embedded-module** member
 - option 60
- :enum** FLI type 200
- :enumeration** FLI type 201
- enum-symbols** function 121
- enum-symbol-value** function 121
- enum-symbol-value-pairs**
 - function 121
- enum-values** function 121
- enum-value-symbol** function 121
- environment variable
 - DYLD_LIBRARY_PATH 166
 - LD_LIBRARY_PATH 166
 - PATH 166

F

- fill-foreign-object** function 123
- :fixnum** FLI type 201
- FLI functions
 - defining 6
- FLI templates 162, 178
- FLI type constructors 12
- FLI types
 - aggregate 12, 14-16
 - :boolean** 193
 - :byte** 194
 - :c-array** 194
 - :char** 196
 - :const** 197
 - defining 4
 - defining new types 57
 - :double** 197
 - :ef-mb-string** 198
 - :ef-wc-string** 199
 - :enum** 200
 - :enumeration** 201
 - :fixnum** 201
 - :float** 201
 - :foreign-array** 202
 - foreign-block-pointer** 61, 203
 - :function** 204
 - immediate 12-13
 - :int** 205
 - :int16** 205
 - :int32** 205
 - :int64** 205
 - :int8** 205
 - :intmax** 205
 - :intptr** 205
 - :lisp-array** 206
 - :lisp-double-float** 208
 - :lisp-float** 209
 - :lisp-simple-1d-array** 209
 - :lisp-single-float** 210
 - :long** 210
 - :long-long** 211
 - lpcstr** 53
 - lpctstr** 54
 - lpcwstr** 54
 - lptstr** 54
 - :one-of** 212
 - :pointer** 213
 - :ptr** 213
 - :ptrdiff-t** 214
 - :reference** 214
 - :reference-pass** 216
 - :reference-return** 216

- released-foreign-block-pointer 217
- :short 217
- :signed 218
- :size-t 219
- :ssize-t 219
- str 53
- :struct 220
- :time-t 221
- tstr 54
- :uint16 222
- :uint32 222
- :uint64 222
- :uint8 222
- :uintmax 222
- :uintptr 222
- :union 223
- :unsigned 224
- :void 225
- :volatile 226
- :wchar-t 226
- :wrapper 226
- wstr 54
- :float FLI type 201
- Foreign blocks 60
- foreign callable
 - defining 31
 - passing and returning strings 34
- Foreign Parser 229
- foreign-aref function 124
- :foreign-array FLI type 202
- foreign-array-dimensions
 - function 125
- foreign-array-element-type
 - function 126
- foreign-array-pointer
 - function 127
- foreign-block-copy function 128
- foreign-block-pointer FLI
 - type 61, 203
- foreign-block-release
 - function 129
- foreign-function-pointer
 - function 130
- foreign-parser package symbols
 - process-foreign-file 234
- foreign-slot-names function 132
- foreign-slot-offset function 133
- foreign-slot-pointer
 - function 134
- foreign-slot-type function 135
- foreign-slot-value function 137
- foreign-typed-aref function 141
- foreign-type-equal-p
 - function 140
- foreign-type-error condition 141
- free function 142
- free-foreign-block function 143
- free-foreign-object function 144
- :function FLI type 204
- functions
 - align-of 67
 - alloca 68
 - allocate-dynamic-foreign-object 68
 - allocate-foreign-block 70
 - allocate-foreign-object 71
 - cast-integer 74
 - connected-module-pathname 74
 - convert-from-foreign-string 75
 - convert-integer-to-dynamic-foreign-object 77
 - convert-to-dynamic-foreign-string 79
 - convert-to-foreign-string 77
 - copy-pointer 81
 - decf-pointer 82
 - dereference 117
 - disconnect-module 120
 - enum-symbols 121
 - enum-symbol-value 121
 - enum-symbol-value-pairs 121
 - enum-values 121
 - enum-value-symbol 121
 - fill-foreign-object 123
 - foreign-aref 124
 - foreign-array-dimensions 125
 - foreign-array-element-type 126
 - foreign-array-pointer 127
 - foreign-block-copy 128
 - foreign-block-release 129
 - foreign-function-pointer 130
 - foreign-slot-names 132
 - foreign-slot-offset 133
 - foreign-slot-pointer 134
 - foreign-slot-type 135
 - foreign-slot-value 137
 - foreign-typed-aref 141
 - foreign-type-equal-p 140
 - free 142

- free-foreign-block 143
- free-foreign-object 144
- get-embedded-module 145
- get-embedded-module-
data 146
- incf-pointer 148
- install-embedded-
module 149
- make-integer-from-
bytes 151
- make-pointer 151
- malloc 153
- module-unresolved-
symbols 154
- null-pointer-p 155
- pointer-address 156
- pointer-element-size 157
- pointer-element-type 157
- pointer-element-type-p 159
- pointer-eq 159
- pointerp 161
- pointer-pointer-type 160
- print-collected-template-
info 162
- print-foreign-modules 163
- process-foreign-file 234
- register-module 163
- replace-foreign-array 169
- replace-foreign-object 173
- set-locale 174
- set-locale-encodings 175
- setup-embedded-module 176
- size-of 177
- start-collecting-tem-
plate-info 178

G

- GCD 61
- gdi+ 64
- gdiplus 64
- get-embedded-module
function 145
- get-embedded-module-data
function 146
- Grand Central Dispatch 61
- graphics functions 64

I

- immediate FLI types 12-13
- incf-pointer function 148
- install-embedded-module

- function 149
- :int FLI type 205
- :int16 FLI type 205
- :int32 FLI type 205
- int32 type 142
- :int64 FLI type 205
- int64 type 142
- :int8 FLI type 205
- :intmax FLI type 205
- :intptr FLI type 205
- iOS 35

L

- languages supported 1
- :lifetime
argument to register-module 164
- Linux 35
- Lisp
 - calling from C 32, 93
 - calling from C++ 32, 93
 - calling into C 2, 61, 103
 - calling into C++ 2, 61, 103
 - calling into C with a block 61
 - calling into C++ with a block 61
- :lisp-array FLI type 206
- :lisp-double-float FLI type 208
- :lisp-float FLI type 209
- :lisp-simple-1d-array FLI
type 209
- :lisp-single-float FLI type 210
- *locale-external-formats*
variable 150
- :long FLI type 210
- :long-long FLI type 211
- lpctr FLI type 53
- lpctstr FLI type 54
- lpcwstr FLI type 54
- lptstr FLI type 54

M

- macros
 - define-c-enum 83
 - define-c-struct 85
 - define-c-typedef 88
 - define-c-union 89
 - define-foreign-block-call-
able-type 91
 - define-foreign-block-
invoker 92
 - define-foreign-callable 31,
35, 93

- define-foreign-converter 98
- define-foreign-forward-reference-type 102
- define-foreign-funcallable 102
- define-foreign-function 31, 103
- define-foreign-pointer 22, 109
- define-foreign-type 11, 110
- define-foreign-variable 112
- define-opaque-pointer 116
- defsystem 60
- with-coerced-pointer 178
- with-dynamic-foreign-objects 180
- with-dynamic-lisp-array-pointer 184
- with-foreign-block 185
- with-foreign-slots 187
- with-foreign-string 188
- with-integer-bytes 190
- with-local-foreign-block 191
- make-integer-from-bytes function 151
- make-pointer function 151
- malloc 23
- malloc function 153
- memory allocation 7, 23
- module-unresolved-symbols function 154

N

- New in LispWorks 7.0
 - 64-bit integer FLI types supported in 32-bit LispWorks 97, 119, 138
 - :allow-sign-mismatch argument to `replace-foreign-array` 170
 - `foreign-function-pointer` function 130
 - hard-float and soft-float calling conventions for ARM platforms 35
 - `released-foreign-block-pointer` FLI type descriptor 217
 - Store a foreign module in a Lisp image with `defsystem` member option
 - :`embedded-module` 60
 - null pointers 24
 - *`null-pointer`* variable 154
 - `null-pointer-p` function 155

O

- :`one-of` FLI type 212

P

- :`pointer` FLI type 213
- `pointer-address` function 156
- `pointer-element-size` function 157
- `pointer-element-type` function 157
- `pointer-element-type-p` function 159
- `pointer-eq` function 159
- `pointerp` function 161
- `pointer-pointer-type` function 160
- pointers 21–30
 - coercing 25
 - copying 23
 - creating 21
 - dereferencing 24–27
 - dynamically allocating 27
 - null pointers 24
 - test functions for 23–24
- *`preprocessor`* variable 230, 233
- *`preprocessor-format-string`* variable 233
- *`preprocessor-include-path`* variable 233, 234
- *`preprocessor-options`* variable 234
- `print-collected-template-info` function 162
- `print-foreign-modules` function 163
- `process-foreign-file` function 234
- :`ptr` FLI type 213
- :`ptrdiff-t` FLI type 214

R

- :`reference` FLI type 214
- :`reference-pass` FLI type 216
- :`reference-return` FLI type 216
- `register-module` function 163
- `released-foreign-block-pointer` FLI type 217
- `replace-foreign-array` function 169
- `replace-foreign-object` function 173

S

Self-contained examples
 foreign blocks 65
 miscellaneous examples 66
set-locale function 174
set-locale-encodings
 function 175
setup-embedded-module
 function 176
:short FLI type 217
:signed FLI type 218
size-of function 177
:size-t FLI type 219
:ssize-t FLI type 219
start-collecting-template-
 info function 178
str FLI type 53
strings
 modifying in C 41
 passing to C 39, 43
 returning from C 45
:struct FLI type 220

T

templates, FLI 162, 178
:time-t FLI type 221
tstr FLI type 54
type constructors 12
types
 int32 142
 int64 142

U

:uint16 FLI type 222
:uint32 FLI type 222
:uint64 FLI type 222
:uint8 FLI type 222
:uintmax FLI type 222
:uintptr FLI type 222
:union FLI type 223
:unsigned FLI type 224

V

variables
 ***locale-external-**
 formats* 150
 null-pointer 154
 preprocessor 230, 233
 ***preprocessor-format-**
 string* 233
 ***preprocessor-include-**

path* 233, 234

preprocessor-options 234
:void FLI type 225
:volatile FLI type 226

W

:wchar-t FLI type 226
with-coerced-pointer macro 178
with-dynamic-foreign-objects
 macro 180
with-dynamic-lisp-array-
 pointer macro 184
with-foreign-block macro 185
with-foreign-slots macro 187
with-foreign-string macro 188
with-integer-bytes macro 190
with-local-foreign-block
 macro 191
:wrapper FLI type 226
wstr FLI type 54