libwheel Documentation

Release 0.2.0

Adrián Pérez de Castro

Aug 05, 2018

Contents

Contents:

CHAPTER 1

Error Reporting & Debugging

The debug-print macros *W_DEBUG* and *W_DEBUGC* produce messages on standard error when _DEBUG_PRINT is defined before including the wheel.h header., By default their expansion are empty statements. The following builds program.c with the debug-print statements turned on:

```
cc -D_DEBUG_PRINT -o program program.c
```

The *W_FATAL* and *W_BUG* macros can be used to produce fatal errors which abort execution of the program.

The *W_WARN* macro can be used to produce non-fatal warnings, which can optionally be converted into fatal warnings at run-time when programs are run with the W_FATAL_WARNIGS environment variable defined to a non-zero value in their environment.

The *w_die()* function can be used to exit a program gracefully after printing an error message to standard error.

1.1 Macros

W_DEBUG (const char **format*, ...)
: Produces a debug-print on standard error with the given *format*. The macro automatically adds the function name, file name and line number to the message.

  If _DEBUG_PRINT is not defined (the default), this macro expands to an empty statement, causing no overhead.

  See *Formatted output* for the available formatting options.

W_DEBUGC (const char **format*, ...)
: Produces a "continuation" debug-print on standard error with the given *format*. The macro **does not** add the function name, file name and line number to the message (hence "continuation").

  If _DEBUG_PRINT is not defined (the default), this macro expands to an empty statement, causing no overhead.

  See *Formatted output* for the available formatting options.

W_FATAL (const char **format*, ...)
: Writes a fatal error message to standard error with the given *format*, and aborts execution of the program. The macro automatically adds the function name, file name and line number where the fatal error is produced.

See *Formatted output* for the available formatting options.

W_BUG (const char **message*)
Marks the line where the macro is expanded as a bug: if control ever reaches the location, a fatal error is produced using *W_FATAL* instructing the user to report the issue. The date and time of the build are included in the error message.

It is possible to supply an optional *message* to be printed next to the supplied text. Note that, if supplied, this should be a hint to help developers of the program.

Note that, as the *message* is optional, both these macro expansions produce valid code:

```
W_BUG();
W_BUG("This is a bug");
```

W_WARN (format, ...)
Writes a warning to standard error with the given *format*. The macro automatically adds the function name, file name and line number where the warning is produced.

See *Formatted output* for the available formatting options.

If the W_FATAL_WARNINGS environment variable is defined and its value is non-zero, warnings are converted into *W_FATAL* errors and execution of the program will be aboerted.

1.2 Functions

void **w_die** (const char **format*, ...)
Prints a message to standard error with a given *format* and exits the program with the EXIT_FAILURE status code.

See *Formatted output* for the available formatting options.

CHAPTER 2

Memory Utilities

The functions *w_malloc()*, *w_realloc()* and the function-like macros *w_new()*, *w_new0()*, and *w_free()* can be used to allocate and release chunks of memory from the heap.

The function-like macros *w_alloc()*, *w_alloc0()*, and *w_resize()* can be used to allocate and resize chunks of memory which contain arrays of elements of the same type.

It is highly encourages to use these instead of any other dynamic allocation facilities, as they do additional checks and will take care of aborting the running program with a meaningful message in case of out-of-memory conditions.

2.1 Macros

w_lmem

Marks a variable as being pointer to a heap-allocated chunk of memory which must be freed by calling *w_free()* on it when the variable goes out of scope.

For example:

```
if (show_frob ()) {
    // frob_message() allocates memory for the message using w_malloc()
    w_lmem char *frob_msg = get_frob_message ();

    w_print ("Frob: $s\n", frob_msg);

    // After this point, "frob_msg" goes out of scope, so w_free()
    // is called automatically on it to free the memory.
}
```

Note that this macro uses variable attributes supported by GCC and Clang to implement this behavior. When building your program with any other compiler a compilation error will be generated if this macro is used.

w_lobj

Marks a variable as being a pointer which holds a reference to to a heap-allocated object, and that *w_obj_unref()* will be called on it when the variable goes out of scope.

Usage example:

```
if (do_frob ()) {
    // Keep a reference to a frob_t, increasing the reference counter.
    w_lobj frob_t *frob_obj = w_obj_ref (get_frob_object ());

    handle_frob (frob_obj);

    // After this point, "frob_obj" goes out of scope, so w_obj_unref()
    // is called automatically on it to decrease the reference counter.
}
```

2.2 Functions

void* **w_malloc** (size_t *size*)
Allocates a chunk of memory from the heap of a given *size* and returns a pointer to it. The returned pointer is guaranteed to be valid.

If it was not possible to allocate memory, a fatal error is printed to standard error and execution aborted.

void* **w_realloc** (void **address*, size_t *new_size*)
Resizes a chunk of memory from the heap at *address* from its current size to a *new_size*.

Note that:

- Passing `NULL` is valid, and it will allocate memory if the *new_size* is non-zero. This means that the following two expressions are equivalent:

```
void *addr = w_realloc (NULL, 42);
void *addr = w_malloc (42);
```

- Requesting a *new_size* of zero causes memory to be deallocated. This means that the following two expressions are equivalent:

```
addr = w_realloc (addr, 0);
w_free (addr);
```

void **w_free** (void **address*)
Frees the chunk of heap memory at *address* and sets the pointer to `NULL`.

type* **w_new** (type)
Allocates a new chunk of memory from the heap suitable to hold a value of a certain *type*. Note that the pointer is returned casted to the appropriate *type* pointer, and no additional casts are needed:

```
int *value = w_new (int);
```

type* **w_new0** (type)
Allocates a zero-filled chunk of memory from the heap suitable to hold a value of a certain *type*. This is equivalent to using *w_new()*, clearing the memory with `memset()`, and then casting to the appropriate type:

```
int *value = w_new0 (int);
w_print ("$i\n", *value);  // Prints "0"
```

type* **w_alloc** (type, size_t *size*)
Allocates a chunk of memory suitable to hold an array of a given *size* of values of a *type*. Note that the returned pointer is casted to the appropriate *type*, and no additional casts are needed:

```
int *point = w_alloc (int, 2);
point[0] = 42;
point[1] = 14;
```

type* **w_alloc0** (type, size_t *size*)
Allocates a zero-filled chunk of memory suitable to hold an array of a given *size* of values of a *type*. This is equivalent to using *w_alloc()*, clearing the memory with `memset()`, and then casting to the appropriate type:

```
int *point = w_alloc0 (int, 2);
w_print ("($i, $i)\n", point[0], point[1]); // Prints "(0, 0)"
```

type* **w_resize** (type* *address*, type, size_t *new_size*)
Resizes a chunk of memory at *address* which contains an array of elements of a given *type* to a *new_size*.

```
int *values = w_alloc (int, 10);
if (need_more_values ())
    values = w_resize (values, int, 20);
```

CHAPTER 3

Objects

The object system allows to do object oriented programming in C99. The object model has the following features:

- Simple inheritance. All object types must "inherit" from `w_obj_t`. Of course, composition of objects is also possible.
- Objects are typically allocated in the heap, but it is possible to allocate them statically, or in the stack with the aid of the `W_OBJ_STATIC` macro.
- Objects keep a reference counter, which can be manipulated using `w_obj_ref()`, and `w_obj_unref()`. Objects are deallocated when their reference counter drops to zero.
- It is possible to assign a "destructor function" to any object using `w_obj_dtor()`.
- Minimal overhead: objects do not have a *vtable* by default, and dynamic method dispatching is not done unless explicitly added by the user.
- Uses only C99 constructs, and it does not require any special compiler support.
- Optionally, when using GCC or Clang, the reference count for an object can be automatically decreased when a pointer to it goes out of scope, by marking it with the `w_lobj` macro.

A number of features included in `libwheel` make use of the object system (for example, the *Input/Output Streams*), or includes support to seamlessly integrate with the object system (for example, the *List Container* can update the reference counter when objects are added to it).

3.1 Usage

This example shows how to define a base "shape" object type: `shape_t`; and two derived types for squares (`square_t`) and rectangles (`rectangle_t`).

In order to have methods which work on any object derived from the shape type, a "vtable" is added manually to perform dynamic dispatch using a shared `struct` which contains function pointers to the actual implementations for each shape. Another valid approach would be to add the function pointers directly in `shape_t` to avoid the extra indirection. This second approach would be better if the function pointers to method implementations could change at runtime, at the cost of each instance of a shape occupying some extra bytes of memory.

Header:

```
// Objects have no vtable by default, so one is defined manually.
typedef struct {
    double (*calc_area)      (void*);
    double (*calc_perimeter) (void*);
} shape_vtable_t;

// Base object type for shapes.
W_OBJ (shape_t) {
    w_obj_t         parent;  // Base object type.
    shape_vtable_t *vtable;  // Pointer to vtable.
};

// A square shape.
W_OBJ (square_t) {
    shape_t parent;  // Inherits both base object and vtable.
    double  side_length;
};

// A rectangular shape.
W_OBJ (rectangle_t) {
    shape_t parent;  // Inherits both base object and vtable.
    double  width;
    double  height;
};

// Functions used to create new objects.
extern shape_t* square_new (double side_length);
extern shape_t* rectangle_new (double width, double height);

// Convenience functions to avoid having to manually make the
// dynamic dispatch through the vtable manually in client code.
static inline double shape_calc_area (shape_t *shape) {
    return (*shape->vtable->calc_area) (shape);
}
static inline double shape_calc_perimeter (shape_t *shape) {
    return (*shape->vtable->calc_perimeter) (shape);
}
```

Implementation:

```
// Methods and vtable for squares.
static double square_calc_area (void *obj) {
    double side_length = ((square_t*) obj)->side_length;
    return side_length * side_length;
}
static double square_calc_perimeter (void *obj) {
    return 4 * ((square_t*) obj)->side_length;
}
static const shape_vtable_t square_vtable = {
    .calc_area      = square_calc_area,
    .calc_perimeter = square_calc_perimeter,
};

shape_t* square_new (double side_length) {
    square_t *square = w_obj_new (square_t);  // Make object.
    square->parent.vtable = &square_vtable;   // Set vtable.
```

(continues on next page)

(continued from previous page)

```
    square->side_length = side_length;
    return (shape_t*) square;
}

// Methods and vtable for rectangles.
static double rectangle_calc_area (void *obj) {
    rectangle_t *rect = (rectangle_t*) obj;
    return rect->width * rect->height;
}
static double rectangle_calc_perimeter (void *obj) {
    rectangle_t *rect = (rectangle_t*) obj;
    return 2 * (rect->width + rect->height);
}
static const shape_vtable_t rectangle_vtable = {
    .calc_area      = rectangle_calc_area,
    .calc_perimeter = rectangle_calc_perimeter,
};

shape_t*
rectangle_new (double width, double height) {
    rectangle_t *rect = w_obj_new (rectangle_t);  // Make object.
    rect->parent.vtable = &rectangle_vtable;      // Set vtable.
    rect->width = width;
    rect->height = height;
    return (shape_t*) rect;
}
```

Using shapes:

```
// Uses the generic shape_* functions.
static void print_shape_infos (shape_t *shape) {
    w_print ("Shape area: $F\n", shape_calc_area (shape));
    w_print ("Shape perimeter: $F\n", shape_calc_perimeter (shape));
}

int main (void) {
    w_lobj shape_t *s = square_new (10);
    w_lobj shape_t *r = rectangle_new (10, 20);
    print_shape_infos (s);  // Works on any object derived from shape_t.
    print_shape_infos (r);  // Ditto.
    return 0;
}
```

3.2 Types

w_obj_t

Base type for objects.

All other object types must "derive" from this type for the objects system to work properly. This is achieved by having a member of this type as first member of object types — either explicitly or by "inheriting" it from another object type:

```
W_OBJ (my_type) {
    // Explicitly make the first member be an "w_obj_t"
```

(continues on next page)

(continued from previous page)

```
    w_obj_t parent;
};

W_OBJ (my_subtype) {
    // The first member itself has an "w_obj_t" as first member.
    my_type parent;
};
```

3.3 Macros

W_OBJ_DECL (type)
Makes a forward declaration of a object class of a certain *type*.

See also *W_OBJ_DEF*.

W_OBJ_DEF (type)
Defines the structure for an object class of a certain *type*.

This macro should be used after the *type* has been declared using the *W_OBJ_DECL* macro.

Typical usage involves declaring the *type* in a header, and the actual layout of it in an implementation file, to make the internals opaque to third party code:

```
// In "my_type.h"
W_OBJ_DECL (my_type);

// In "my_type.c"
W_OBJ_DEF (my_type) {
    w_obj_t parent;
    int     value;
    // ...
};
```

W_OBJ (type)
Declares *and* defines the structure for an object class of a certain *type*. This is equivalent to using *W_OBJ_DECL* immediately followed by *W_OBJ_DEF*.

For example:

```
W_OBJ (my_type) {
    w_obj_t parent;
    int     value;
    // ...
};
```

This is used instead of a combination of *W_OBJ_DECL* and *W_OBJ_DEF* when a forward declaration is not needed, and it does not matter that the internals of how an object class is implemented are visible in headers:

W_OBJ_STATIC (destructor)
Initializes a statically-allocated object, and sets *destructor* to be called before the object is deallocated by *w_obj_destroy()*.

Similarly to *w_obj_mark_static()*, this macro allows to initialize objects for which the memory they occupy will not be deallocated.

Typical usage involves initializing static global objects, or objects allocated in the stack, e.g.:

```
W_OBJ (my_type) {
    w_obj_t parent;
    int     value;
};

static my_type static_object = {
    .parent = W_OBJ_STATIC (NULL),
    .value  = 42,
};

void do_foo (void) {
    my_type stack_object = {
        .parent = W_OBJ_STATIC (NULL),
        .value  = 32,
    };

    use_object (&stack_object);
}
```

3.4 Functions

void* **w_obj_ref** (void **object*)
Increases the reference counter of an *object*.

The *object* itself is returned, to allow easy chaining of other function calls.

void* **w_obj_unref** (void **object*)
Decreases the reference counter of an *object*.

Once the reference count for an object reaches zero, it is destroyed using *w_obj_destroy()*.

The *object* itself is returned, to allow easy chaining of other function calls.

void **w_obj_destroy** (void **object*)
Destroys an *object*.

If a destructor function was set for the *object* using *w_obj_dtor()*, then it will be called before the memory used by the object being freed.

void* **w_obj_dtor** (void **object*, void (**destructor*)(void*))
Registers a *destructor* function to be called when an *object* is destroyed using *w_obj_destroy()*.

The *object* itself is returned, to allow easy chaining of other function calls.

void **w_obj_mark_static** (void **object*)
Marks an *object* as being statically allocated.

When the last reference to an object marked as static is lost, its destructor will be called, but the area of memory occupied by the object **will not** be freed. This is the same behaviour as for objects initialized with the *W_OBJ_STATIC* macro. The typical use-case for this function to mark objects that are allocated as part of others, and the function is called during their initialization, like in the following example:

```
W_OBJ (my_type) {
    w_obj_t     parent;
    w_io_unix_t unix_io;
};

void my_type_free (void *objptr) {
```

(continues on next page)

(continued from previous page)

```
    w_obj_destroy (&self->unix_io);
}

my_type* my_type_new (void) {
    my_type *self = w_obj_new (my_type);
    w_io_unix_init_fd (&self->unix_io, 0);
    w_obj_mark_static (&self->unix_io);
    return w_obj_dtor (self, _my_type_free);
}
```

type* **w_obj_new**(type)
Creates a new instance of an object of a given *type*.

Freshly created objects always have a reference count of `1`.

type* **w_obj_new_with_priv_sized**(type, size_t *size*)
Creates a new instance of an object of a given *type*, with additional space of *size* bytes to be used as instance private data.

A pointer to the private data of an object can be obtained using *w_obj_priv()*.

type* **w_obj_new_with_priv**(type)
Creates a new instance of an object of a given *type*, with additional space to be used as instance private data. The size of the private data will be that of a type named after the gicen *type* with a `_p` suffix added to it.

A pointer to the private data of an object can be obtained using *w_obj_priv()*.

Typical usage:

```
// In "my_type.h"
W_OBJ (my_type) {
    w_obj_t parent;
};

extern my_type* my_type_new ();


// In "my_type.c"
typedef struct {
    int private_value;
} my_type_p;


my_type* my_type_new (void) {
    my_type *obj = w_obj_new_with_priv (my_type);
    my_type_p *p = w_obj_priv (obj, my_type);
    p->private_value = 42;
    return obj;
}
```

void* **w_obj_priv**(void **object*, type)
Obtains a pointer to the private instance data area of an *object* of a given *type*.

Note that only objects created using *w_obj_new_with_priv_sized()* or *w_obj_new_with_priv()* have a private data area. The results of using this function on objects which do not have a private data area is undefined.

CHAPTER 4

Buffers

Buffers provide a variable-length area of memory in which data may be held and manipulated. Contained data is not interpreted, and length of it is tracked, so it is possible to add null bytes to a buffer.

Allocating a buffer is done in the stack using the *W_BUF* macro. to initialize it. After initialization, all buffer functions can be used, and when the buffer is not needed anymore, its contents can be freed using *w_buf_clear()*.

4.1 Usage

```
// Initialize the buffer.
w_buf_t b = W_BUF;

// Append some string pieces.
w_buf_append_str (&b, "Too much work ");
w_buf_append_str (&b, "and no joy makes");
w_buf_append_str (&b, " Jack a dull boy");

// Buffer contents can be printed directly using the $B format.
w_print ("$B\n", &b);

// Free the memory used by the contents of the buffer.
w_buf_clear (&b);
```

4.2 Types

w_buf_t
: Buffer type.

4.3 Macros

W_BUF
Initializer for buffers. It can be used to initialize buffers directly on the stack:

```
w_buf_t buffer = W_BUF;
```

4.4 Functions

void **w_buf_resize**(*w_buf_t** *buffer*, size_t *size*)
Adjust the size of a buffer keeping contents. This is mostly useful for trimming contents, when shrinking the buffer. When a buffer grows, random data is liklely to appear at the end.

Parameters

- **buffer** – A w_buf_t buffer
- **size** – size New size of the buffer

void **w_buf_set_str**(*w_buf_t *buffer*, const char **string*)
Set the contents of a buffer to a C string.

Parameters

- **buffer** – A w_buf_t buffer
- **string** – String to set the buffer to.

void **w_buf_append_mem**(*w_buf_t *buffer*, const void **address*, size_t *length*)
Appends the contents of a chunk of memory of *length* bytes starting at *address* to a *buffer*.

Parameters

- **buffer** – A w_buf_t buffer.
- **address** – Pointer to the memory block of memory.
- **length** – Length of the memory block.

void **w_buf_append_str**(*w_buf_t *buffer*, const char **string*)
Appends a *string* to a *buffer*.

void **w_buf_append_char**(*w_buf_t *buffer*, int *character*)
Appends a *character* to a *buffer*.

void **w_buf_append_buf**(*w_buf_t *buffer*, const *w_buf_t *other*)
Appends the contents of *other* buffer to another *buffer*.

char* **w_buf_str**(*w_buf_t *buffer*)
Obtains the contents of a *buffer* as a `NULL`-terminated C string.

> **Warning:** If the buffer contains embedded null characters, functions like `strlen()` will not report the full length of the buffer.

The returned pointer is owned by the *buffer*, and there two ways in which the memory region can be freed:

- Clearing the *buffer* with *w_buf_clear()*. The returned pointer will be invalid afterwards.
- Calling *w_free()* on the returned pointer. The *buffer* will be invalid and must not be used afterwards.

The second way is useful to assemble a string which is returned from a function, for example:

```
char* concat_strings (const char *s, ...)
{
  w_buf_t buffer = W_BUF;
  w_buf_set_str (&buffer, s);

  va_list args;
  va_start (args, s);
  while ((s = va_args (args, const char*)))
    w_buf_append_str (&buffer, s);
  va_end (args);

  return w_buf_str (&buffer);
}
```

void **w_buf_clear** (*w_buf_t *buffer*)
Clears a *buffer*, freeing any used memory.

w_io_result_t **w_buf_format** (*w_buf_t *buffer*, const char *format*, ...)
Appends text with a given *format* into a *buffer*, consuming additional arguments as needed by the *format*.

See *Formatted output* for the available formatting options.

w_io_result_t **w_buf_formatv** (*w_buf_t *buffer*, const char *format*, va_list *arguments*)
Appends text with a given *format* into a *buffer*, consuming additional *arguments* as needed by the *format*.

See *Formatted output* for the available formatting options.

bool **w_buf_is_empty** (const *w_buf_t *buffer*)
Checks whether a *buffer* is empty.

size_t **w_buf_size** (const *w_buf_t *buffer*)
Obtains the size of a *buffer*.

char* **w_buf_data** (*w_buf_t *buffer*)
Obtains a pointer to the internal data stored by a *buffer*.

> **Warning:** The returned value may be `NULL` when the buffer is empty.

const char* **w_buf_const_data** (const *w_buf_t *buffer*)
Obtains a pointer to the internal data stored by a *buffer*, returning it as a `const` pointer. This may be used instead of *w_buf_data()* when the data is not going to be modified.

> **Warning:** The returned value may be `NULL` when the buffer is empty.

CHAPTER 5

List Container

The implementation uses a doubly-linked list, meaning that most operations are very efficient, and the list can be traversed both forward and backward.

The functions which use numeric indexes to refer to elements in the list can be slow, and should be avoided if possible: *w_list_at()*, *w_list_insert_at()*, and *w_list_del_at()*. Negative numeric indexes can be passed to functions, with the same meaning as in Python: `-1` refers to the last element, `-2` to the element before the last, and so on.

If a list is meant to contain objects (see *Objects*), it is possible to let the list reference-count the objects (using *w_obj_ref()* and *w_obj_unref()*) by passing `true` when creating a list with *w_list_new()*. If enabled, whenever an item is added to the list, its reference count will be increased, and it will be decreased when the item is removed from the list.

5.1 Usage

```
w_list_t *fruits = w_list_new (false);

w_list_append (fruits, "apples");
w_list_append (fruits, "bananas");

w_list_foreach (item, fruits)  // Prints "apples bananas "
    w_print ("$s ", (const char*) *item);

w_list_insert_after (fruits, w_list_first (fruits), "pears");
// fruits = {"apples", "pears", "bananas"}

w_list_del_head (fruits);
// fruits = {"pears", "bananas"}

w_list_foreach_reverse (item, fruits)  // Prints "bananas pears "
    w_print ("$s ", (const char*) *item);
```

(continues on next page)

(continued from previous page)

```
w_obj_unref (fruits);  // Decrease the reference counter, frees the list.
```

5.2 Types

w_list_t
Object type of a list container.

5.3 Macros

w_list_foreach (iterator, *w_list_t *list*)
Defines a loop over all items in a *list*.

Typical usage:

```
w_list_t *list = make_string_list ();
w_list_foreach (i, list) {
    w_io_format (w_stdout, "$s\n", (const char*) *i);
}
```

w_list_foreach_reverse (iterator, *w_list_t *list*)
Defines a loop over all items in a *list*, in reverse order.

Typical usage:

```
w_list_t *list = make_string_list ();
w_list_foreach_reverse (i, list) {
    w_io_format (w_stdout, "$s\n", (const char*) *i);
}
```

5.4 Functions

*w_list_t** **w_list_new** (bool *reference_counted*)
Creates a new list, in which elements are optionally *reference_counted*.

void **w_list_clear** (*w_list_t *list*)
Clears a *list*, removing all of its elements.

void **w_list_push_tail** (*w_list_t *list*, void *element*)
Appends an *element* to the end of a *list*.

void **w_list_push_head** (*w_list_t *list*, void *element*)
Inserts an *element* at the beginning of a *list*.

void* **w_list_pop_head** (*w_list_t *list*)
emoves the element at the beginning of a *list* and returns it.

Note that this **will not** decrease the reference counter when reference counting is enabled: it is assumed that the caller will use the returned item.

void* **w_list_pop_tail** (*w_list_t *list*)
Removes the element at the end of a *list* and returns it.

Note that this **will not** decrease the reference counter when reference counting is enabled: it is assumed that the caller will use the returned item.

void* **w_list_at** (const *w_list_t *list*, long *index*)
Obtains the value stored in a *list* at a given *index*. Negative indexes count from the end of the list.

void* **w_list_head** (const *w_list_t *list*)
Obtains the element at the first position of a *list*.

void* **w_list_tail** (const *w_list_t *list*)
Obtains the element at the last position of a *list*.

w_iterator_t **w_list_first** (const *w_list_t *list*)
Obtains an iterator pointing to the first element of a *list*.

w_iterator_t **w_list_last** (const *w_list_t *list*)
Obtains an iterator pointing to the last element of a *list*

w_iterator_t **w_list_next** (const *w_list_t *list*, w_iterator_t *iterator*)
Makes an *iterator* to an element of a *list* point to the next element in the list, and returns the updated iterator.

w_iterator_t **w_list_prev** (const *w_list_t *list*, w_iterator_t *iterator*)
Makes an *iterator* to an element of a *list* point to the previous element in the list, and returns the updated iterator.

void **w_list_insert_before** (*w_list_t *list*, w_iterator_t *position*, void *element*)
Inserts an *element* in a *list* before a particular *position*.

void **w_list_insert_after** (*w_list_t *list*, w_iterator_t *position*, void *element*)
Inserts an *element* in a *list* after a particular *position*.

void **w_list_insert_at** (*w_list_t *list*, long *index*, void *element*)
Inserts an *element* in a *list* at a given *index*..

Note that the operation is optimized for some particular indexes like `0` (first position) and `-1` (last position), bu in general this function runs in *O(n)* time depending on the size of the list.

void **w_list_del** (*w_list_t *list*, w_iterator_t *position*)
Deletes the element at a given *position* in a *list*.

void **w_list_del_at** (*w_list_t *list*, long *index*)
Deletes the element at a given *index* in a *list*.

size_t **w_list_size** (const *w_list_t *list*)
Obtains the number of elements in a *list*.

bool **w_list_is_empty** (const *w_list_t *list*)
Checks whether a *list* is empty.

void **w_list_del_head** (*w_list_t *list*)
Deletes the element at the beginning of a *list*.

Contrary to *w_list_pop_head()*, the element is **not** returned, and the reference counter is decreased (if reference counting is enabled).

void **w_list_del_tail** (*w_list_t *list*)
Deletes the element at the end of a *list*.

Contrary to *w_list_pop_tail()*, the element is **not** returned, and the reference counter is decreased (if reference counting is enabled).

void **w_list_insert** (*w_list_t* *list, w_iterator_t *position*, void *element*)
Alias for *w_list_insert_before()*.

void **w_list_append** (*w_list_t* *list, void *element*)
Alias for *w_list_push_tail()*.

void **w_list_pop** (*w_list_t* *list, void *element*)
Alias for *w_list_pop_tail()*.

CHAPTER 6

Input/Output Streams

The following kinds of streams are provided:

- *Input/Output on Buffers*
- *Input/Output on memory*
- *Input/Output on FILE* streams*
- *Input/Output on Unix file descriptors*
- *Input/Output on Sockets*

6.1 Formatted output

A number of functions support specifying a *format string*, and will accept a variable amount of additional function arguments, depending on the *format specifiers* present in the *format string*. All those functions use the same formatting mechanism, as described here.

Format specifiers are sequences of characters started with a dollar symbol (`$`), followed by at a character, which determines the type and amount of the additional function arguments being consumed.

The recognized *format specifiers* are:

Specifier	Type(s)	Output format.
`$c`	int	Character.
`$l`	long int	Decimal number.
`$L`	unsigned long int	Decimal number.
`$i`	int	Decimal number.
`$I`	unsigned int	Decimal number.
`$X`	unsigned long int	Hexadecimal number.
`$O`	unsigned long int	Octal number.
`$p`	void*	Pointer, as hexadecimal number.
`$f`	float	Floating point number.
`$F`	double	Floating point number.
`$s`	const char*	A `\0`-terminated string.
`$B`	w_buf_t*	Arbitrary data (usually a string).
`$S`	size_t, const char*	String of a particular length.
`$e`		Last value of `errno`, as an integer.
`$E`		Last value of `errno`, as a string.
`$R`	w_io_result_t	String representing the return value of an input/output operation (see *Output for w_io_result_t* below).

6.1.1 Output for `w_io_result_t`

The `$R` *format specifier* will consume a *w_io_result_t* value, and format it as a string, in one of the following ways:

IO<EOF> End-of-file marker. This is printed when *w_io_eof()* would return `true` for the value.

IO<string> This format is used for errors, the *string* describes the error. Most of the time the error strings correspond to the values obtained by using `strerror (w_io_result_error (value))` on the value.

IO<number> This format is used for successful operations, the *number* is the amount of bytes that were handled during the input/output operation, and it can be zero, e.g. for the result of *w_io_close()*.

6.1.2 Reusing

The main entry point of the formatting mechanism is the *w_io_format()* function, which works for any kind of output stream. To allow for easier integration with other kinds of output, an alternate version of the function, *w_io_formatv()*, which accepts a `va_list`, is provided as well.

By providing a custom output stream implementation, it is possible to reuse the formatting mechanism for your own purposes. The *w_buf_format()* function, which writes formatted data to a buffer, is implemented using this technique.

The following example implements a function similar to `asprintf()`, which allocates memory as needed to fit the formatted output, using *w_io_formatv()* in combination with a *w_io_buf_t* stream:

```
char*
str_format (const char *format, ...)
{
```

(continues on next page)

(continued from previous page)

```
    // Using NULL uses a buffer internal to the w_io_buf_t.
    w_io_buf_t buffer_io;
    w_io_buf_init (&buffer_io, NULL, false);

    // Writing to a buffer always succeeds, the result can be ignored.
    va_list args;
    va_start (args, format);
    W_IO_NORESULT (w_io_formatv ((w_io_t*) &buffer_io, format, args));
    va_end (args);

    // The data area of the buffer is heap allocated, so it is safe to
    // return a pointer to it even when the w_io_buf_t is in the stack.
    return w_io_buf_str (&buffer_io);
}
```

6.2 Types

w_io_result_t
Represents the result of an input/output operation, which is one of:

- An error, which signals the failure of the operation. The *w_io_failed()* can be used to check whether an input/output operation failed. If case of failure, the error code can be obtained using *w_io_result_error()*; otherwise the operation succeeded and it can have one of the other values.
- An indication that the end-of-file marker has been reached — typically used when reading data from a stream. The *w_io_eof()* function can be used to check for the end-of-file marker.
- A successful operation, indicating the amount of data involved. This is the case when an operation neither failed, neither it is the end-of-file marker. The amount of bytes handled can be retrieved using *w_io_result_bytes()*.

It is possible to obtain a textual representation of values of this type by using the `$R` *format specifier* with any of the functions that use the *Formatted output* system.

w_io_t
Represents an input/output stream.

6.3 Macros

W_IO_RESULT (bytes)
Makes a *w_io_result_t* value which indicates a successful operation which handled the given amount of *bytes*.

W_IO_RESULT_ERROR (error)
Makes a *w_io_result_t* value which indicates a failure due to given *error*.

W_IO_RESULT_EOF
Makes a *w_io_result_t* value which indicates a successful operation that reached the end-of-file marker.

W_IO_RESULT_SUCCESS
Makes a *w_io_result_t* value which indicates a successful operation.

W_IO_CHAIN (result, expression)
This macro expands to code that evaluates an *expression*, which must return a *w_io_result_t* value. If the

operation failed, it will cause the current function to return the error, otherwise the amount of bytes returned by *w_io_result_bytes()* are added to the amount in the variable of type *w_io_result_t()* passed as the *result* parameter.

Typical usage involves using the macro is inside a function that performs a number of reads (or writes), and the overall status of a "chain" of input/output operations is to be reported back as a result.

The expanded macro is roughly equivalent to:

```
w_io_result_t expr_result = expression;
if (w_io_failed (expr_result))
    return expr_result;
result.bytes += w_io_result_bytes (expr_result);
```

As an example, consider a data stream which contains "messages" of the form `SIZE:DATA`, with the *SIZE* of the message encoded as a plain text integer, followed by a colon, and *SIZE* bytes of arbitrary data. The following function reads such messages, one at a time, returning the result of reading from the *input* as a *w_io_result_t* value —which can be checked for end-of-file or errors—, and returning the *DATA* in a *buffer* and its expected *size*:

```
w_io_result_t
read_message (w_io_t *input, w_buf_t *message, unsigned long *size)
{
    w_io_result_t bytes = W_IO_RESULT (0);

    // Read the length of the record. If reading fails, the macro
    // causes the function to return early with an error result.
    // The amount of bytes read is added to "bytes".
    W_IO_CHAIN (bytes, w_io_fscan_ulong (input, size));

    // Read the separator. Again: the macro causes an early error
    // return on failure, or incrementing "bytes" on success.
    char separator = '\0';
    W_IO_CHAIN (bytes, w_io_read (input, &separator, 1));
    if (separator != ':')
        return W_IO_RESULT_ERROR (EBADMSG);

    // Read the message contents. Again: the macro causes an early
    // error on failure, or incrementing "bytes" on success.
    w_buf_resize (message, record_size);
    W_IO_CHAIN (bytes, w_io_read (input, w_buf_data (message), *size);

    return bytes;  // Returns the total amount of bytes read.
}
```

W_IO_CHECK (expression)

This macro expands to code that evaluates an *expression*, which must return a *w_io_result_t* value. If the operation failed, it will cause the current function to return the error. It is roughly equivalent to:

```
w_io_result_t expr_result = expression;
if (w_io_failed (expr_result))
    return expr_result;
```

W_IO_CHECK_RETURN (expression, value)

Similar to *W_IO_CHECK*, but instead of returning a *w_io_result_t* if the *expression* fails to perform input/output, the given *value* is returned instead. It is roughly equivalent to:

```
if (w_io_failed (expression))
    return value;
```

W_IO_NORESULT (expression)
This macro expands to code that evaluates an *expression*, which must return a *w_io_result_t* value, and ignoring that result value. This is typically used when a an input/output operation is known to never fail, and the result status will not be checked, or when the operation might fail, but it does not matter whether it did.

6.4 Functions

void **w_io_init** (*w_io_t* **stream*)
Initializes a base input/output *stream*.

w_io_result_t **w_io_close** (*w_io_t* **stream*)
Closes an input/output *stream*.

w_io_result_t **w_io_read** (*w_io_t* **stream*, void **buffer*, size_t *count*)
Reads up to *count* bytes from the an input *stream*, placing the data in in memory starting at *buffer*.

Passing a *count* of zero always succeeds and has no side effects.

If reading succeeds, the amount of bytes read may be smaller than the requested *count*. The reason may be that the end-of-file marker has been reached (and it will be notified at the next attempt of reading data), or because no more data is available for reading at the moment.

w_io_result_t **w_io_write** (*w_io_t* **stream*, const void **buffer*, size_t *count*)
Writes up to *count* bytes from the data in memory starting at *buffer* to an output *stream*.

Passing a *count* of zero always succeeds and has no side effects.

int **w_io_getchar** (*w_io_t* **stream*)
Reads the next character from a input *stream*.

If the enf-of-file marker is reached, `W_IO_EOF` is returned. On errors, negative values are returned.

w_io_result_t **w_io_putchar** (*w_io_t* **stream*, int *character*)
Writes a *character* to an output *stream*.

void **w_io_putback** (*w_io_t* **stream*, int *character*)
Pushes a *character* back into an input *stream*, making it available during the next read operation.

> **Warning:** Pushing more than one character is not supported, and only the last pushed one will be saved.

w_io_result_t **w_io_flush** (w_io_stream **stream*)
For an output *stream*, forces writing buffered data to the stream.

For in input *stream*, discards data that may have been fetched from the stream but still not consumed by the application.

int **w_io_get_fd** (*w_io_t* **stream*)
Obtains the underlying file descriptor used by a *stream*.

> **Warning:** Not all types of input/output streams have an associated file descriptor, and a negative value will be returned for those.

w_io_result_t **w_io_format** (*w_io_t *stream*, const char **format*, ...)
Writes data with a given *format* to an output *stream*. The amount of consumed arguments depends on the *format* string.

See *Formatted output* for more information.

w_io_result_t **w_io_formatv** (*w_io_t *stream*, const char **format*, va_list *arguments*)
Writes data with a given *format* to an output *stream*, consuming the needed additional *arguments* from the supplied `va_list`. The amount of consumed arguments depends on the *format* string.

See *Formatted output* for more information.

w_io_result_t **w_print** (format, ...)
Writes data in the given *format* to the standard output stream `w_stdout`. The amount of consumed arguments depends on the *format* string.

See *Formatted output* for more information.

w_io_result_t **w_printerr** (format, ...)
Writes data in the given *format* to the standard error stream `w_stderr`. The amount of consumed arguments depends on the *format* string.

See *Formatted output* for more information.

w_io_result_t **w_io_read_until** (*w_io_t *stream*, *w_buf_t *buffer*, *w_buf_t *overflow*, int *character*, unsigned *read_bytes*)
Reads data from a *stream* until a certain *character* is read. Data is read in chunks of *read_bytes* size, and placed in a *buffer*, with the excess characters placed in an *overflow* buffer, which can be passed back to continue scanning for the stop *character* in subsequent calls.

Passing zero for the *read_bytes* size will make the function use a default chunk size — usually the size of a memory page.

This function is intended to read records of data of variable size which are separated using a certain character as a delimiter. For example, usually fortune data files have items separated by `%` characters. The following function would read such a file:

```
void read_fortunes (w_io_t *input) {
    w_buf_t fortune = W_BUF;
    w_buf_t overflow = W_BUF;

    for (;;) {
        w_io_result_t r = w_io_read_line (input, &fortune, &overflow, 0);

        if (w_io_failed (r)) {
            w_printerr ("Error reading fortunes: $R\n" r);
            break;
        }
        if (w_io_eof (r))
            break;

        handle_fortune (&fortune);
    }

    w_buf_clear (&overflow);
    w_buf_clear (&fortune);
}
```

bool **w_io_failed** (*w_io_result_t result*)
Checks whether the *result* of an input/output operation was a failure.

If the *result* happens to be a failure, `w_io_result_error()` can be used to retrieve the error code.

bool **w_io_eof**(*w_io_result_t result*)
Checks whether the *result* of an input/output operation was the end-of-file marker.

int **w_io_result_error**(*w_io_result_t result*)
Obtains the code of the error that caused an input/output operation to return a failure *result*.

This function only returns meaningful values when *result* indicates a failed operation. This condition can be checked using `w_io_failed()`.

size_t **w_io_result_bytes**(*w_io_result_t result*)
Obtains the amount of bytes which were involved as the *result* of an input/output operation.

When the *result* signals a failed operation, or the end-of-file marker, the returned value is always zero.

w_io_result_t **w_io_read_line**(*w_io_t *stream*, *w_buf_t *line*, *w_buf_t *overflow*, unsigned *read_bytes*)
Reads a *line* of text from an input *stream*.

This is a convenience function that calls `w_io_read_until()` passing `'\n'` as delimiter character.

CHAPTER 7

Input/Output on Buffers

Provides support for using the *stream functions* to read and write to and from `w_buf_t` buffers.

7.1 Types

w_io_buf_t
Performs input/output on a `w_buf_t` buffer.

7.2 Functions

void **w_io_buf_init** (*w_io_buf_t *stream*, *w_buf_t *buffer*, bool *append*)
Initialize a *stream* object (possibly allocated in the stack) to be used with a *buffer*.

Passing a `NULL` *buffer* will create a new buffer owned by the stream object, which can be retrieved using `w_io_buf_get_buffer()`. The memory used by this buffer will be freed automatically when the stream object is freed. On the contrary, when a valid buffer is supplied, the caller is responsible for calling `w_buf_clear()` on it.

Optionally, the stream position can be setup to *append* data to the contents already present in the given *buffer*, insted of overwriting them.

*w_io_t** **w_io_buf_open** (*w_buf_t *buffer*)
Creates a stream object to be used with a *buffer*.

Passing a `NULL` *buffer* will create a new buffer owned by the stream object, which can be retrieved using `w_io_buf_get_buffer()`. The memory used by this buffer will be freed automatically when the stream object is freed. On the contrary, when a valid buffer is supplied, the caller is responsible for calling `w_buf_clear()` on it.

*w_buf_t** **w_io_buf_get_buffer** (*w_io_buf_t *stream*)
Obtain a pointer to the buffer being used by a *stream*.

char* **w_io_buf_str** (*w_io_buf_t* *stream)
Obtain a string representation of the contents of the buffer being used by a *stream*.

CHAPTER 8

Input/Output on memory

Provides support for using the *stream functions* to read and write to and from regions of memory of fixed sizes.

8.1 Types

w_io_mem_t
Performs input/output on a region of memory of a fixed size.

8.2 Functions

void **w_io_mem_init** (*w_io_mem_t* *stream, uint8_t *address, size_t *size*)
Initializes a *stream* object (possibly located in the stack) to be used with a region of memory of a given *size* located at *address*.

*w_io_t** **w_io_mem_open** (uint8_t *address, size_t *size*)
Creates a stream object to be used with a region of memory of a given *size* located at *address*.

uint8_t* **w_io_mem_data** (*w_io_mem_t* *stream)
Obtains the base address to the memory region on which a *stream* operates.

size_t **w_io_mem_size** (*w_io_mem_t* *stream)
Obtains the size of the memory region on which a *stream* operates.

CHAPTER 9

Input/Output on `FILE*` streams

Provides support for using the *stream functions* to read and write to and from `FILE*` streams as provided by the C standard library.

9.1 Types

`w_io_stdio_t`
Performs input/output on a `FILE*` stream.

9.2 Functions

void **`w_io_stdio_init`**(*w_io_stdio_t* **stream*, FILE **stdio_stream*)
Initializes a *stream* object (possibly allocated in the stack) to be used with a given *stdio_stream*.

*w_io_t** **`w_io_stdio_open`**(FILE **stdio_stream*)
Creates a stream object to be used with a given *stdio_stream*.

CHAPTER 10

Input/Output on Unix file descriptors

Once a Unix stream object has been initialized, they can be operated used the common *stream functions*.

10.1 Types

w_io_unix_t
Performs input/output using Unix file descriptors.

10.2 Functions

*w_io_t** **w_io_unix_open** (const char **path*, int *mode*, unsigned *permissions*)
Creates a stream object to be used with an Unix file descriptor by opening the file at *path* with the given *mode* and *permissions*.

This is a convenience function that calls `open()` and then uses *w_io_unix_open_fd()*.

If opening the file fails, `NULL` is returned.

*w_io_t** **w_io_unix_open_fd** (int *fd*)
Creates a stream object to be used with an Unix file descriptor.

bool **w_io_unix_init** (*w_io_unix_t *stream*, const char **path*, int *mode*, unsigned *permissions*)
Initializes a stream object (possibly allocated in the stack) to be used with an Unix file descriptor by opening the file at *path* with the given *mode* and *permissions*.

This is a convenience function that calls `open()` and then uses *w_io_unix_init_fd()*.

The return value indicates whether the file was opened successfully.

void **w_io_unix_init_fd** (*w_io_unix_t *stream*, int *fd*)
Initializes a stream object (possibly allocated in the stack) to be used with an Unix file descriptor.

CHAPTER 11

Input/Output on Sockets

Provides support for using the *stream functions* to read and write to and from sockets.

The following kinds of sockets are supported:

- TCP sockets.
- Unix sockets.

11.1 Usage

11.1.1 Client

To connect a socket to a remote server (and use it as a client), the usual procedure is as follows:

1. Create the socket with `w_io_socket_open()`.
2. Connect to the remote endpoint using `w_io_socket_connect()`.
3. Exchange data using `w_io_write()` and `w_io_read()`, or any other of the *stream functions*.
4. *(Optional)* Once no more data needs to be written, a half-close can be performed using `w_io_socket_send_eof()`. It will still be possible to read data from the stream.
5. Close the socket, using `w_io_close()` or calling `w_obj_unref()` on it and letting it be closed and destroyed when no more references to the socket are held.

The following code will make an HTTP request and read the response back into a `w_buf_t`:

```
w_buf_t
http_get (const char *ip_address, unsigned port, const char *resource)
{
    w_lobj w_io_t *stream = w_io_socket_open (W_IO_SOCKET_TCP4,
                                              ip_address,
                                              port);
    if (!stream || !w_io_socket_connect ((w_io_socket_t*) stream))
```

(continues on next page)

(continued from previous page)

```
        w_die ("Cannot create socket and connect to $s\n", ip_address);

    W_IO_NORESULT (w_io_format (stream, "GET $s HTTP/1.0\r\n", resource));
    w_io_socket_send_eof ((w_io_socket_t*) stream);

    w_buf_t response = W_BUF;
    char buf[512];

    for (;;) {
        w_io_result_t r = w_io_read (stream, buf, w_lengthof (buf));
        if (w_io_failed (r))
            w_die ("Error reading from $s: $R\n", ip_address, r);
        if (w_io_result_bytes (r) > 0)
            w_buf_append_mem (&response, buf, w_io_result_bytes (r));
        else if (w_io_eof (r))
            break;
    }

    return response;
}
```

11.1.2 Server

Using a socket to server requests is a bit more convoluted, but still much easier than using the sockets API directly. The overall procedure is:

1. Define a request handler function which conforms to the following signature:

   ```
   bool (*request_handler) (w_io_socket_t *socket)
   ```

2. Create the socket with *w_io_socket_open()*.

3. Start serving requests with *w_io_socket_serve()*. The function will bind to the address specified when creating the socket and start serving requests. For each request, the request handler function will be called with a socket that can be used to handle the request.

The following code implements a simple TCP echo server:

```
static bool
handle_echo_request (w_io_socket_t *socket)
{
    char buf[512];
    for (;;) {
        w_io_result_t r = w_io_read ((w_io_t*) socket, buf, w_lengthof (buf));
        if (w_io_failed (r))
            w_die ("Error reading from client: $R\n", r);
        if (w_io_eof (r))
            break;

        r = w_io_write ((w_io_t*) socket, buf, w_io_result_bytes (r));
        if (w_io_failed (r))
            w_die ("Error writing to client: $R\n", r);
    }
    w_io_socket_send_eof (socket);

    return true;  // Keep accepting more requests.
```

(continues on next page)

(continued from previous page)

```
}

int main (void)
{
    w_lobj w_io_t *socket = w_io_socket_open (W_IO_SOCKET_TCP4,
                                              "0.0.0.0",  // Address.
                                              4242);      // Port.
    if (!socket)
        w_die ("Cannot create socket: $E\n");

    if (!w_io_socket_serve ((w_io_socket_t*) socket,
                            W_IO_SOCKET_FORK,  // Handle each request in a child␣
↪process.
                            handle_echo_request))
        w_dir ("Cannot accept connections: $E\n");

    return 0;
}
```

11.2 Types

w_io_socket_t
Performs input/output on sockets.

11.3 Functions

bool **w_io_socket_init** (*w_io_socket_t* *stream*, w_io_socket_kind_t *kind*, ...)
Initializes a socket *stream* (possibly allocated in the stack).

For a description of the additional function arguments, please check the documentation for *w_io_socket_open()*.

*w_io_t** **w_io_socket_open** (w_io_socket_kind_t *kind*, ...)
Create a new socket of a given *kind*.

Sockets are created in a state in which they can be used both for client and server sockets:

- *w_io_socket_connect()* will put the socket in client mode and connect it to the specified address.
- *w_io_socket_serve()* will put the socket in server mode, and start listening for connections at the specified address.

The parameters that need to be passed to this function vary depending on the *kind* of the socket being created.

Unix sockets: Pass `W_IO_SOCKET_UNIX` as *kind*, and the path in the file system where the socket is to be created (or connected to).

TCP sockets: Pass `W_IO_SOCKET_TCP4` as *kind*, plus the IP address (as a string) and the port to use (or to connect to).

bool **w_io_socket_connect** (*w_io_socket_t* *socket*)
Connect a *socket* to a server.

This makes a connection to the host specified when creating the socket with *w_io_socket_open()*, and puts it in client mode. Once the socket is successfully connected, read and write operations can be performed in the socket.

The return value indicates whether the connection was successful.

bool **w_io_socket_serve**(*w_io_socket_t* **socket*, w_io_socket_serve_mode_t *mode*, bool (**handler*)(*w_io_socket_t**))

Serves requests using a *socket*. This function will start a loop accepting connections, and for each connection an open socket will be passed to the given *handler* function. The *mode* in which each request is served can be specified:

- `W_IO_SOCKET_SINGLE`: Each request is served by calling directly and waiting for it to finish. This makes impossible to serve more than one request at a time.
- `W_IO_SOCKET_THREAD`: Each request is served in a new thread. The handler is invoked in that thread.
- `W_IO_SOCKET_FORK`: A new process is forked for each request. The handler is invoked in the child process.

bool **w_io_socket_send_eof**(*w_io_socket_t* **socket*)

Half-close a *socket* on the write direction. This closes the socket, but only in writing one direction, so other endpoint will think that the end of the stream was reached (thus the operation is conceptually equivalent to sending and "end of file marker"). Read operations can still be performed in a socket which was half-closed using this function.

Note that for completely closing the socket, *w_io_close()* should be used instead.

The return value indicates whether the half-close was successful.

const char* **w_io_socket_unix_path**(*w_io_socket_t* **socket*)

Obtain the path in the filesystem for an Unix socket.

Note that the result is undefined if the socket is of any other kind than an Unix socket.

CHAPTER 12

Tasks

The tasks system provides "green threads" (also known as *lightweight cooperative threads*, or coroutines).

Additionally, the following utilities are provided to be used along with the tasks system:

- Functions to suspend a coroutine and wait for I/O to be completed: `w_task_yield_io_read()`, `w_task_yield_io_write()`.
- The `w_io_task_open()` and `w_io_task_init()` functions can be used to create a `w_io_task_t` wrapper to ease using asynchronous I/O with tasks.

12.1 Types

w_task_t
Type of a task.

Tasks are created by `w_task_prepare()`, and the resources used by a task (including the stack space used by the task and the `w_task_t` value itself) will be automatically freed when the tasks is exited. Never deallocate a task manually, or use it after it has been exited.

w_task_func_t
Type of task functions.

w_io_task_t
Stream wrapper for asynchronous input/output.

This is a subclass of :type:w_io_t; all the *stream functions* can be used on objects of this type. Reading and writing data uses `w_task_yield_io_read()` and `w_task_yield_io_write()`, so instead of blocking until completion the current task will be suspended automatically.

12.2 Functions

*w_task_t** **w_task_prepare**(*w_task_func_t function*, void *data, size_t *stack_size*)
Creates a task with a given *stack_size* and prepares it for running a *function*, passing a *data* pointer to the

function. The task will be in paused state upon creation.

To get tasks running, the scheduler must be running, see *w_task_run_scheduler()*.

The *stack_size* is always rounded up to the size of a memory page. It is possible to pass zero to get the smallest possible stack size (usually 4 kB).

*w_task_t** **w_task_current**()
Obtains the task currently running.

> **Warning:** This function **must** be called from inside a task, once the task scheduler has been started. Otherwise, calling this function is an error and the execution of the program will be aborted.

void **w_task_set_is_system**(*w_task_t *task*, bool *is_system*)
Set whether a *task* is a system task. System tasks are those which are always running.

System tasks do not prevent *w_task_run_scheduler()* from returning.

bool **w_task_get_is_system**(*w_task_t *task*)
Checks whether a task is a system task.

See also *w_task_set_is_system()*.

void **w_task_set_name**(*w_task_t *task*, const char *name*)
Sets the *name* of a *task*. The name of the tast is copied as-is, and it is not interpreted in any way. The ability of naming tasks is mainly provided as an aid for debugging client code.

It is possible to pass `NULL` as the *name*, which will clear any custom name previously set.

const char* **w_task_get_name**(*w_task_t *task*)
Obtains the name of a *task*.

If a name has not been set using *w_task_set_name()*, an autogenerated one of the form `Task<ID>` will be returned.

void **w_task_run_scheduler**()
Runs the task scheduler.

The scheduler will choose tasks in a round-robin fashion, and let each task run until it gives up the CPU explicitly using *w_task_yield()* or implicitly when waiting for input/output on a stream be means of *w_task_yield_io_read()* and *w_task_yield_io_write()*.

The scheduler will keep scheduling tasks until all non-system tasks have been exited.

This function **must** be called in the main function of a program. Typically:

```
extern void process_argument (void*);

int main (int argc, char **argv) {
    while (argc--)
        w_task_prepare (process_argument, *argv++, 0);
    w_task_run_scheduler ();
    return 0;
}
```

void **w_task_yield**()
Make the current task give up the CPU, giving control back to the task scheduler, which will give other other tasks the chance to run.

void **w_task_exit**()
Exits the current task. This can be used to exit from a task at any point, without needing to return from the task function.

w_io_result_t **w_task_yield_io_read**(*w_io_t* *stream, void *buffer, size_t count)
Reads *count* bytes into the memory block at *buffer* from an input *stream*, suspending the current task as needed.

If the *stream* has been set as non-blocking and reading from it results in an `EAGAIN` or `EWOULDBLOCK` error, the current task will give up the CPU and wait until the data is available for reading as many times as needed, until *count* bytes are read, the end-of-file marker is reached, or an error is found.

w_io_result_t **w_task_yield_io_write**(*w_io_t* *stream, const void *buffer, size_t count)
Writes *count* bytes from the memory block at *buffer* to an output *stream*, suspending the current task as needed.

If the *stream* has been set as non-blocking and writing to it results in an `EAGAIN` or `EWOULDBLOCK` error, the current task will give up the CPU and wait until the stream accepts writing data as many times as needed, until *count* bytes are written, or an error is found.

bool **w_io_task_init**(*w_io_task_t* *wrapper, *w_io_t* *stream)
Initializes a stream *wrapper* object (possibly allocated in the stack) which wraps a *stream*. The *wrapper* behaves like the wrapped *stream*, suspending the current task when needed to ensure that I/O is performed asynchronously.

The return value indicates whether the *stream* can be wrapped. Most of the streams for which *w_io_get_fd()* returns a valid file descriptor can be wrapped.

*w_io_t** **w_io_task_open**(*w_io_t* *stream)
Wraps a *stream* and returns an object that behaves like the wrapped *stream*, suspending the current task when needed to ensure that I/O is performed asynchronously.

Returns `NULL` when the *stream* cannot be wrapped. Most of the streams for which *w_io_get_fd()* returns a valid file descriptor can be wrapped.

void **w_task_system**()
Mark the current task as a system task.

See also `w_task_set_system()`.

const char* **w_task_name**()
Obtain the name of the current task.

See also *w_task_get_name()*.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Index

W