Programming the Click Modular Router

Eddie Kohler

Table of Contents

1 Overview

1.1 Packet Transfer

2 Helper Classes

2.1 String

The `String` class represents a string of characters. `String`s may be constructed from C strings, characters, numbers, and so forth. They may also be added together. The underlying character arrays are dynamically allocated; operations on `String`s allocate and free memory as needed. A `String` and its substrings will generally share memory. `String`s may be assigned, stored, and passed to functions.

2.1.1 Class Initialization

The `String` class maintains some global state that must be explicitly initialized with the `static_initialize` static method. You can explicitly clean up this state with `static_cleanup`, if you'd like. String also provides a helper class, `String::Initializer`, that initializes `String`'s global state in its constructor.

`void` **static_initialize** () — Static Method on `String`

Call this function exactly once, at the beginning of the program, before any Strings are created or other String functions called.

`void` **static_cleanup** () — Static Method on `String`

Call this function exactly once, just before the program exits, to clean up String-related memory. It is an error to call any `String` method, except for `String` destructors, after calling `static_cleanup`.

String::Initializer — Class

Declare a `String::Initializer` object in any source file that contains a global string object. The constructor for the `String::Initializer` will call `String::static_initialize` if necessary. For example, this source file is in error, since it declares a global string without a corresponding `Initializer`:

```
#include <click/string.hh>
String foo = "bar";
int main(int, char **) { /* ... */ }
```

To fix it, declare a `String::Initializer` before the global string.

```
#include <click/string.hh>
String::Initializer string_initializer;
String foo = "bar";
int main(int, char **) { /* ... */ }
```

2.1.2 Constructors

String () — Constructor on `String`

Creates a string with no characters.

String (`const char *`*s*) Constructor on `String`
: Creates a string containing a copy of the C string *s*.

String (`const char *`*s*`, int` *len*) Constructor on `String`
: Creates a string containing the first *len* characters of *s*. If *len* is negative, then this function treats *s* as a C string, effectively setting *len* to `strlen(`*s*`)`.

String (`char` *c*) Constructor on `String`
String (`unsigned char` *c*) Constructor on `String`
: Creates a string containing the single character *c*.

String (`int` *n*) Constructor on `String`
String (`unsigned` *n*) Constructor on `String`
String (`long` *n*) Constructor on `String`
String (`unsigned long` *n*) Constructor on `String`
String (`long long` *n*) Constructor on `String`
String (`unsigned long long` *n*) Constructor on `String`
String (`double` *n*) Constructor on `String`
: Creates a string containing an ASCII decimal representation of the number *n*. For example, if *n* is 20, then `String(`*n*`)` equals `"20"`. The `double` constructor is not available in the kernel.

`const String &` **null_string** () Static Method on `String`
: Returns a const reference to a string with no characters. Useful in situations where you wish to avoid unnecessary memory operations by returning string references instead of `String`s.

`String` **stable_string** (`const char *`*s*`, int` *len*) Static Method on `String`
: Creates and returns a string containing the *len* bytes of data starting at *s*. If *len* is negative, then this function treats *s* as a C string, effectively setting *len* to `strlen(`*s*`)`. The caller guarantees that *s* is located in stable, read-only memory and will not be changed while any `String` references to it still exist. For example, *s* might be a C string constant. The `String` implementation will not alter or free *s*. Functions such as `mutable_data` (see below) will return copies of *s*, not *s* itself.

`String` **garbage_string** (`int` *len*) Static Method on `String`
: Creates and returns a string containing *len* bytes of garbage data.

2.1.3 Contents

Caution: Any pointer to a string's data should be treated as temporary, since once the string is destroyed, that memory will be freed. Remember, however, that a temporary `String` object will not be destroyed until the end of the statement in which it was created. Therefore, this use of `cc()` is safe:

```
String a, b; // ...
fprintf(stderr, "%s\n", (a + b).cc());
```

This use is not safe:

```
String a, b; // ...
const char *s = (a + b).cc();
fprintf(stderr, "%s\n", s); // probably an error
```

`const char *` **data** () `const` — Method on `String`

Returns a pointer to the string's data. This data is not guaranteed to be null-terminated. Only the first `length()` of its characters are valid.

`int` **length** () `const` — Method on `String`

Returns the string's length in characters.

operator bool () — Method on `String`
operator bool () `const` — Method on `String`

Returns true iff the string has at least one character.

`char *` **mutable_data** () — Method on `String`

Returns a mutable pointer to the string's data. If the data is shared with any other `String` object, or was allocated by `stable_string` (see above), then this method will transparently modify the `String` to use a unique copy of the data, and return that.

`const char *` **cc** () — Method on `String`
`const char *` **c_str** () — Method on `String`
operator const char * () — Method on `String`

Returns a pointer to the string's data as a C string. This may transparently modify the `String` by adding a null character after the string's data, which may involve making a copy of the data. This null character will not be counted as part of the string's length.

`char *` **mutable_c_str** () — Method on `String`

Returns a mutable pointer to the string's data as a C string.

2.1.4 Characters and Indices

`char` **operator[]** (`int` *i*) `const` — Method on `String`

Returns the *i*th character of the string. *i* should be between 0 and `length`() − 1.

`char` **back** () `const` — Method on `String`

Returns the last character of the string. The string must not be empty.

`int` **find_left** (`int` *c*, `int` *start* `= 0`) `const` — Method on `String`

Returns the position of the first occurrence of the character *c* in the string on or after position *start*. If *c* does not occur on or after position *start*, returns −1.

`int` **find_right** (`int` *c*) `const` — Method on `String`
`int` **find_right** (`int` *c*, `int` *start*) `const` — Method on `String`

Returns the position of the last occurrence of the character *c* in the string before position *start*. If *start* is not supplied, returns the absolute last occurrence of *c* in the string. If *c* does not occur before position *start*, returns −1.

`int` **find_left** (`const String &`*s*`, int` *start* `= 0) const` — Method on `String`

Returns the position of the first occurrence of the substring *s* in the string on or after position *start*. If *s* does not occur on or after position *start*, returns −1.

2.1.5 Derived Strings

`String` **substring** (`int` *pos*`, int` *len*`) const` — Method on `String`

Returns a new string containing characters *pos* through *pos*+*len* − 1 of this string.

If *pos* is negative, then start −*pos* characters from the end of the string. If *len* is negative, then drop −*len* characters from the end of the string. *len* may be too large; only characters actually in the string will be returned. If *pos* is too large or too small, the result is a null string.

These examples demonstrate the use of *substring*:

```
String("abcde").substring(2, 2)   == "cd"
String("abcde").substring(-3, 2)  == "cd"
String("abcde").substring(-3, -1) == "cd"
String("abcde").substring(2, 10)  == "cde"
String("abcde").substring(10, 4)  == ""
String("abcde").substring(-10, 4) == ""
```

`String` **substring** (`int` *pos*`) const` — Method on `String`

Same as `substring(`*pos*`, length() -` *pos*`)`: return a new string containing all of this string's characters starting at *pos*.

`String` **lower** () `const` — Method on `String`

Return a string equal to this string, but with all alphabetic characters translated to lower case.

`String` **upper** () `const` — Method on `String`

Return a string equal to this string, but with all alphabetic characters translated to upper case.

`String` **printable** () `const` — Method on `String`

Return a string equal to this string, but with all non-printable characters replaced by quote sequences. For example, null characters '⟨NUL⟩' become '`^@`' sequences.

2.1.6 Appending to Strings

If you are gradually building up a string by successive appends, you should probably use `StringAccum` instead of these `String` operations (see Section 2.2 [StringAccum], page 8).

`void` **append** (`const char *`*s*`, int` *len*`)` — Method on `String`

Appends the first *len* characters of *s* to the end of this string. If *len* is negative, then this function treats *s* as a C string, effectively setting *len* to `strlen(`*s*`)`.

void **append_fill** (int *c*, int *len*) — Method on String

Adds *len* copies of the character *c* to the end of this string.

void **append_garbage** (int *len*) — Method on String

Adds *len* arbitrary characters to the end of this string.

String & **operator+=** (const String &*s*) — Method on String
String & **operator+=** (const char **s*) — Method on String
String & **operator+=** (char *c*) — Method on String

Appends the string *s* or character *c* to this string.

String **operator+** (String *s1*, const String &*s2*) — Function
String **operator+** (String *s1*, const char **s2*) — Function
String **operator+** (const char **s1*, const String &*s2*) — Function
String **operator+** (String *s1*, char *c*) — Function

Appends the string *s2* or character *c* to the string *s1*, and returns the resulting string.

2.1.7 Comparison

bool **equals** (const char **s*, int *len*) const — Method on String

Compares this string to the first *len* characters of *s*. If *len* is negative, then this function treats *s* as a C string, effectively setting *len* to `strlen(s)`. Returns true iff the two strings have the same length and contain the same characters in the same order.

bool **operator==** (const String &*s1*, const String &*s2*) — Function
bool **operator==** (const char **s1*, const String &*s2*) — Function
bool **operator==** (const String &*s1*, const char **s2*) — Function

Returns true iff the two strings are equal—that is, returns `s1.equals(s2.data(), s2.length())`.

bool **operator!=** (const String &*s1*, const String &*s2*) — Function
bool **operator!=** (const char **s1*, const String &*s2*) — Function
bool **operator!=** (const String &*s1*, const char **s2*) — Function

Returns true iff the two strings are not equal—that is, returns `!(s1 == s2)`.

int **hashcode** (const String &*s*) — Function

Returns a number with the property that, for any two equal strings *s1* and *s2*, `hashcode(s1) == hashcode(s2)`. With this function, `String`s may be used as keys for `HashMap`s and `BigHashMap`s (see Section 2.5 [HashMap], page 13).

2.1.8 Out-of-Memory Conditions

`String` operations are robust against out-of-memory conditions. If there is not enough memory to create a particular string, the `String` implementation returns a special "out-of-memory" string instead. This is a contagious empty string. Any concatenation operation (`operator+` or `append`) involving an out-of-memory string has an out-of-memory result. Out-of-memory strings compare unequal to every other string, including themselves.

All out-of-memory strings share the same `data`, which is different from the `data` of any other string.

`bool` **out_of_memory** () `const` — Method on `String`
: Returns true iff this string is an out-of-memory string.

`const String &` **out_of_memory_string** () — Static Method on `String`
: Returns an out-of-memory string.

2.2 StringAccum

The `StringAccum` class, like `String` (see Section 2.1 [String], page 3), represents a string of characters. `StringAccum` is optimized for building a string through *accumulation*, or successively appending substrings until the whole string is ready. A `StringAccum` has both a *length*—the number of characters it currently contains—and a *capacity*—the maximum number of characters it could hold without reallocating memory.

2.2.1 Constructors

StringAccum () — Constructor on `StringAccum`
: Creates a `StringAccum` with no characters.

StringAccum (`int` *capacity*) — Constructor on `StringAccum`
: Creates a `StringAccum` with no characters, but a capacity of *capacity*. *capacity* must be greater than zero.

`StringAccum`'s copy constructor (`StringAccum(const StringAccum &)`) and assignment operator (`operator=(const StringAccum &)`) are private. Thus, `StringAccum`s cannot be assigned or passed as arguments. Of course, references to `StringAccum`s may be passed as arguments, and this usage is quite common.

2.2.2 Appending with `operator<<`

Generally, you append to a `StringAccum` using iostreams-like `<<` operators, which this section describes. The next section describes the low-level interface, the `append` and `pop_back` methods.

Here is a conventional use of `StringAccum`'s `<<` operators:

```
struct timeval tv; StringAccum sa; int n; // ...
sa << tv << ": There are " << n << " things.\n";
```

`StringAccum &` **operator<<** (`StringAccum &`*sa*`, char` *c*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, unsigned char` *c*) Function

Appends the character *c* to the `StringAccum` *sa* and returns *sa*.

`StringAccum &` **operator<<** (`StringAccum &`*sa*`, const char *`*s*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, const String &`*s*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, const StringAccum &`*sa2*) Function

Appends the string *s*, or the value of the `StringAccum` *sa2*, to *sa* and returns *sa*.

`StringAccum &` **operator<<** (`StringAccum &`*sa*`, short` *n*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, unsigned short` *n*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, int` *n*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, unsigned` *n*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, long` *n*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, unsigned long` *n*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, long long` *n*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, unsigned long long` *n*) Function
`StringAccum &` **operator<<** (`StringAccum &`*sa*`, double` *n*) Function

Appends an ASCII decimal representation of the number *n* to *sa* and returns *sa*. For example, if *n* is 20, then *sa* `<<` *n* has the same effect as *sa* `<< "20"`. The `double` operator is not available in the kernel.

2.2.3 Appending Data Types

`StringAccum` comes with `operator<<` definitions for the `bool`, `struct timeval`, `IPAddress`, and `EtherAddress` types. Of course, you can write your own `operator<<` functions for other data types, either using existing `operator<<`s or the manipulation functions described in the next section.

`StringAccum &` **operator<<** (`StringAccum &`*sa*`, bool &`*val*) Function

Appends the string `true` or `false` to *sa*, according to the value of *val*.

`StringAccum &` **operator<<** (`StringAccum &`*sa*`, const struct timeval &`*tv*) Function

Appends an ASCII decimal representation of the time value *tv* to *sa* and returns *sa*. The time value is printed as if by `printf("%ld.%06ld",` *tv*`.tv_sec,` *tv*`.tv_usec)`.

`StringAccum &` **operator<<** (`StringAccum &`*sa*`, IPAddress &`*addr*) Function

Appends the conventional dotted-quad representation of the IP address *addr* to *sa* and returns *sa*. For example, '`sa << addr`' might have the same effect as '`sa << "18.26.4.44"`'.

StringAccum & **operator<<** (StringAccum &*sa*, const EtherAddress &*addr*) — Function

Appends the conventional colon-separated hexadecimal representation of the Ethernet address *addr* to *sa* and returns *sa*. For example, ‘`sa << addr`’ might have the same effect as ‘`sa << "00:02:B3:06:06:36:EE"`’.

2.2.4 Manipulation

This section describes lower-level methods for manipulating `StringAccum` objects. The `append` methods append data to the `StringAccum`; the `extend`, `reserve`, and `forward` methods add space to the end of it; and the `clear` and `pop_back` methods remove its characters.

void **append** (char *c*) — Method on StringAccum
void **append** (unsigned char *c*) — Method on StringAccum

Appends the character *c* to the end of this `StringAccum`. Equivalent to `*this <<` *c*.

void **append** (const char **s*, int *len*) — Method on StringAccum

Appends the first *len* characters of *s* to the end of this `StringAccum`. If *len* is negative, then this function treats *s* as a C string, effectively setting *len* to `strlen(`*s*`)`.

char * **extend** (int *len*) — Method on StringAccum

Puts *len* arbitrary characters at the end of this `StringAccum` and returns a pointer to those characters. The return value may be a null pointer if there is not enough memory to grow the character array. This method increases the `StringAccum`'s length by *len*, which must be greater than or equal to zero.

char * **extend** (int *len*, int *extra*) — Method on StringAccum

Puts *len* arbitrary characters at the end of this `StringAccum` and returns a pointer to those characters. Also ensures space for *extra* additional characters following the *len* new characters; however, these characters do not contribute to the `StringAccum`'s length. The return value may be a null pointer if there is not enough memory to grow the character array. Increases the `StringAccum`'s length by *len*, which must be greater than or equal to zero.

This form of `extend` is generally used to compensate for the null character appended by C string functions like `sprintf`. For example:

```
if (char *buf = string_accum.extend(4, 1))
    // 4 real characters plus one terminating null
    sprintf(buf, "\\%03o", i);
```

Caution: The pointer returned by `extend`, or the `reserve` method described below, should be treated as transient. It may become invalid after the next call to a method that grows the `StringAccum`, such as `append`, `extend`, or one of the `operator<<` functions, and will definitely become invalid when the `StringAccum` is destroyed.

The `reserve` and `forward` methods together provide a convenient, fast interface for appending strings of unknown, but bounded, length.

char * **reserve** (int *len*) — Method on StringAccum

Reserves space for *len* characters at the end of this StringAccum and returns a pointer to those characters. The return value may be a null pointer if there is not enough memory to grow the character array. This method does not change the StringAccum's length, although it may change its capacity. Use forward to change the StringAccum's length.

void **forward** (int *amt*) — Method on StringAccum

Adds *amt* to the StringAccum's length without changing its data. This method is used in conjunction with reserve, above. Use reserve to get space suitable for appending a string of unknown, but bounded, length. After finding the actual length, use forward to inform the StringAccum. *amt* must be greater than or equal to zero, and less than or equal to the remaining capacity.

Finally, these methods remove characters from a StringAccum rather than add characters to it.

void **clear** () — Method on StringAccum

Erases the StringAccum, making its length zero (an empty string).

void **pop_back** () — Method on StringAccum
void **pop_back** (int *amt*) — Method on StringAccum

Remove the last character, or the last *amt* characters, of the string. *amt* must be greater than or equal to zero, and less than or equal to the StringAccum's length.

2.2.5 Contents

> **Caution**: The pointer returned by data and c_str should be treated as transient. It may become invalid after the next call to a method that grows the StringAccum, such as append, extend, or one of the operator<< functions, and will definitely become invalid when the StringAccum is destroyed.

const char * **data** () const — Method on StringAccum
char * **data** () — Method on StringAccum

Returns a pointer to the character data contained in this StringAccum.

int **length** () const — Method on StringAccum

Returns the number of characters in this StringAccum.

operator bool () — Method on String
operator bool () const — Method on String

Returns true iff this StringAccum has at least one character.

const char * **c_str** () — Method on StringAccum
const char * **cc** () — Method on StringAccum

Returns a pointer to the character data contained in this StringAccum. Guarantees that the returned string is null-terminated: the length()th character will be '\0'. This does not affect the StringAccum's contents or length.

char **operator**[] (int *i*) const — Method on StringAccum
char & **operator**[] (int *i*) — Method on StringAccum

Returns the *i*th character of this StringAccum. *i* must be greater than or equal to zero, and less than the StringAccum's length. Note that the non-const version of this method returns a mutable character reference, which facilitates code like

```
StringAccum sa; // ...
sa[5] = 'a';
```

2.2.6 Results

StringAccum's take methods return the string accumulated by a series of calls to operator<< or similar methods. Each take method makes StringAccum relinquish responsibility for its character array memory, passing that responsibility on to its caller. The caller should free the memory when it is done—either with delete[], for the take and take_bytes methods, or by relying on String to handle it, for the take_string method.

Each take method additionally restores the StringAccum to its original, empty state. Further appends or similar operations will begin building a new string from scratch.

void **take** (unsigned char *&*s*, int &*len*) — Method on StringAccum

Sets the *s* variable to this StringAccum's character data and *len* to its length. Then clears the StringAccum's internal state.

char * **take** () — Method on StringAccum
unsigned char * **take_bytes** () — Method on StringAccum

Returns this StringAccum's character data, then clears the StringAccum's internal state. The methods differ only in their return types. Note that StringAccum::length will always return zero immediately after a take or take_bytes. If you need to know the string's length, call length first.

String **take_string** () — Method on StringAccum

Returns this StringAccum's character data as a String object (see Section 2.1 [String], page 3), then clears the StringAccum's internal state. This method hands the character data over to the String implementation; no data copies are performed.

2.2.7 Out-of-Memory Conditions

StringAccum operations are robust against out-of-memory conditions. If there is not enough memory to complete a particular operation, the StringAccum is erased and turned into a special out-of-memory indicator. This is a contagious empty string. Every operation on such a buffer (except for clear) leaves it in the out-of-memory state.

bool **out_of_memory** () const — Method on StringAccum

Returns true iff this StringAccum is an out-of-memory indicator.

The extend and reserve methods can return null pointers on out-of-memory; their callers should always check that their return values are non-null.

2.3 Vector

2.4 Bitvector

2.5 HashMap

2.6 BigHashMap

2.7 ErrorHandler

All Click error messages are passed to an instance of the `ErrorHandler` class. `ErrorHandler` separates the generation of error messages from the particular way those messages should be printed. It also makes it easy to automatically decorate errors with context information.

Most Click users must know how to report errors to an `ErrorHandler`, and how `ErrorHandlers` count the messages they receive. This section also describes how to decorate error messages with error veneers, and how to write new `ErrorHandlers`.

`ErrorHandler` and its important subclasses are defined in `<click/error.hh>`.

2.7.1 Class Initialization

The `ErrorHandler` class maintains some global state that must be initialized by calling `static_initialize` at the beginning of the program, and may be freed by calling `static_cleanup` when execution is complete.

`void` **static_initialize** (`ErrorHandler` **default_errh*) — Static Method on `ErrorHandler`

Call this function exactly once, at the beginning of the program, before any error messages are reported to any `ErrorHandler`. It is OK to create arbitrary `ErrorHandler` objects before calling this method, however. The *default_errh* argument becomes the default `ErrorHandler`; see Section 2.7.5 [Basic ErrorHandlers], page 17.

`void` **static_cleanup** () — Static Method on `ErrorHandler`

Call this function exactly once, just before the program exits. Destroys the default and silent `ErrorHandlers` and cleans up other `ErrorHandler`-related memory. It is an error to call any `ErrorHandler` method after calling `static_cleanup`.

2.7.2 Reporting Errors

`ErrorHandler`'s basic error reporting methods take a format string, which may use `printf`-like '`%`' escape sequences, and additional arguments as required by the format string. See Section 2.7.3 [Error Format Strings], page 15, for more details on the format string. The five methods differ in the seriousness of the error they report.

void **debug** (const char **format*, ...) Method on ErrorHandler
void **message** (const char **format*, ...) Method on ErrorHandler
int **warning** (const char **format*, ...) Method on ErrorHandler
int **error** (const char **format*, ...) Method on ErrorHandler
int **fatal** (const char **format*, ...) Method on ErrorHandler

Report the error described by *format* and any additional arguments. The methods are listed by increasing seriousness. Use debug for debugging messages that should not be printed in a production environment; message for explanatory messages that do not indicate errors; warning for warnings (this function prepends the string 'warning: ' to every line of the error message); error for errors; and fatal for errors so serious that they should halt the execution of the program. The three functions that indicate errors, warning, error, and fatal, always return -EINVAL. In some environments, fatal will actually exit the program with exit code 1.

Each of these methods has an analogue that additionally takes a *landmark*: a string representing where the error took place. A typical landmark contains a file name and line number, separated by a colon—'foo.click:31', for example.

void **ldebug** (const String &*landmark*, const char **format*, ...) Method on ErrorHandler
void **lmessage** (const String &*landmark*, const char **format*, ...) Method on ErrorHandler
int **lwarning** (const String &*landmark*, const char **format*, ...) Method on ErrorHandler
int **lerror** (const String &*landmark*, const char **format*, ...) Method on ErrorHandler
int **lfatal** (const String &*landmark*, const char **format*, ...) Method on ErrorHandler

Report the error described by *format* and any additional arguments. The error took place at *landmark*. Most ErrorHandlers will simply prepend '*landmark*: ' to each line of the error message.

These methods are all implemented as wrappers around the verror function. This function takes a landmark, a format string, a va_list packaging up any additional arguments, and a *seriousness value*, which encodes how serious the error was. The Seriousness enumerated type, which is defined in the ErrorHandler class, represents seriousness values. There are five constants, corresponding to the five error-reporting methods:

ERR_DEBUG
: Corresponds to debug and ldebug.

ERR_MESSAGE
: Corresponds to message and lmessage.

ERR_WARNING
: Corresponds to warning and lwarning.

ERR_ERROR
: Corresponds to error and lerror.

`ERR_FATAL`
Corresponds to `fatal` and `lfatal`.

int verror (`Seriousness` *seriousness*, `const String` &*landmark*, `const char *`*format*, `va_list` *val*) — Method on `ErrorHandler`

Report the error described by *format* and *val*. The error took place at *landmark*, if *landmark* is nonempty. The *seriousness* value is one of the five constants described above. Always returns `-EINVAL`.

2.7.3 Format Strings

`ErrorHandler`'s format strings closely follow C's standard `printf` format strings. Most characters in the format string are printed verbatim. The '`%`' character introduces a *conversion*, which prints data read from the remaining arguments. The format string may contain newlines '`\n`', but it need not end with a newline; `ErrorHandler` will add a final newline if one does not exist.

Each conversion, or formatting escape, follows this pattern:

- First, the '`%`' character introduces each conversion.
- Next comes zero or more *flag characters*;
- then an optional *field width*;
- then an optional *precision*;
- then an optional *length modifier*;
- and finally, the mandatory *conversion specifier*, which is usually a single character, but may be a name enclosed in braces.

We discuss each of these is turn.

Any conversion may be modified by zero or more of these flag characters.

'`#`'
The value should be converted to an "alternate form". For '`o`' conversions, the first character of the output string is made '`0`', by prepending a '`0`' if there was not one already. For '`x`' and '`X`' conversions, nonzero values have '`0x`' or '`0X`' prepended, respectively.

'`0`'
The value should be zero padded. For '`d`', '`i`', '`u`', '`o`', '`x`', and '`X`' conversions, the converted value is padded on the left with '`0`' characters rather than spaces.

'`-`'
The value should be left-justified within the field width.

'` `' (a space)
Leave a blank before a nonnegative number produced by a signed conversion.

'`+`'
Print a '`+`' character before a nonnegative number produced by a signed conversion.

The optional *field width*, a decimal digit string, forces the conversion to use a minimum number of characters. The result of a conversion is padded on the left with space characters to reach the minimum field width, unless one of the '`0`' or '`-`' flags was supplied.

The optional *precision* is a decimal digit string preceded by a period '`.`'. For '`d`', '`i`', '`u`', '`o`', '`x`', and '`X`' conversions, the precision specifies the minimum number of digits that must appear; results with fewer digits are padded on the left with '0' characters. For the 's' conversion, the precision specifies the maximum number of characters that can be printed. For 'e', 'f', 'E', and 'F' conversions, it specifies the number of digits to appear after the radix character; for 'g' and 'G' conversions, the number of significant digits.

If either the field width or precision is specified as a star '*', `ErrorHandler` reads the next argument as an integer and uses that instead.

Length modifiers affect the argument type read by the conversion. There are three modifiers:

'h' The next argument is a `short` or `unsigned short`. Affects the 'd', 'i', 'u', 'o', 'x', and 'X' conversions.

'l' The next argument is a `long` or `unsigned long`. Affects the 'd', 'i', 'u', 'o', 'x', and 'X' conversions.

'll' The next argument is a `long long` or `unsigned long long`. Affects the 'd', 'i', 'u', 'o', 'x', and 'X' conversions.

Finally, these are the conversions themselves.

's' Print the `const char *` argument, treated as a C string.

'c' The `int` argument is treated as a character constant. Printable ASCII characters (values between 32 and 126) are printed verbatim. Characters '\n', '\t', '\r', and '\0' use those C escape representations. Other characters use the representation '\%03o'.

'd', 'i' The argument is an `int`; print its decimal representation.

'u' The argument is an `unsigned int`; print its decimal representation.

'o' The argument is an `unsigned int`; print its octal representation.

'x', 'X' The argument is an `unsigned int`; print its hexadecimal representation. The '%x' conversion uses lowercase letters; '%X' uses uppercase letters.

'e', 'f', 'g', 'E', 'F', 'G'
The argument is a `double`; print its representation as if by `printf` (user-level drivers only).

'p' The `void *` argument is cast to `unsigned long` and printed as by '%#lx'.

'%' Print a literal '%' character.

'{element}'
The argument is an `Element *`. Print that element's declaration.

Note that `ErrorHandler` does not support the 'n' conversion.

2.7.4 Counting Errors

`ErrorHandler` objects count the number of errors and warnings they have received and make those values available to the user.

`virtual int` **nwarnings** () `const` — Method on `ErrorHandler`
`virtual int` **nerrors** () `const` — Method on `ErrorHandler`

Returns the number of warnings or errors received by this `ErrorHandler` so far.

`virtual void` **reset_counts** () — Method on `ErrorHandler`

Resets the `nwarnings` and `nerrors` counters to zero.

These counters are typically used to determine whether an error has taken place in some complex piece of code. For example:

```
int before_nerrors = errh->nerrors();
// ... complex code that may report errors to errh ...
if (errh->nerrors() != before_nerrors) {
    // an error has taken place
}
```

2.7.5 Basic `ErrorHandlers`

Every Click error message eventually reaches some *basic* `ErrorHandler`, which generally prints the messages it receives. The user-level driver's basic `ErrorHandler` prints error messages to standard error, while in the Linux kernel module, the basic `ErrorHandler` logs messages to the syslog and stores them for access via '`/proc/click/errors`'.

Two basic `ErrorHandlers` are always accessible via static methods: the *default* `ErrorHandler`, returned by `default_handler` and set by `set_default_handler`; and the *silent* `ErrorHandler`, returned by `silent_handler`, which ignores any error messages it receives.

`ErrorHandler *` **default_handler** () — Static Method on `ErrorHandler`

Returns the default `ErrorHandler`.

`void` **set_default_handler** (`ErrorHandler *errh`) — Static Method on `ErrorHandler`

Sets the default `ErrorHandler` to *errh*. The `static_initialize` method also sets the default `ErrorHandler`; see Section 2.7.1 [ErrorHandler Initialization], page 13.

`ErrorHandler *` **silent_handler** () — Static Method on `ErrorHandler`

Returns the silent `ErrorHandler`. This handler ignores any error messages it receives. It maintains correct `nwarnings` and `nerrors` counts, however.

`FileErrorHandler`, a kind of basic `ErrorHandler`, is available in any user-level program. It prints every message it receives to some file, usually standard error. It can also prepend an optional context string to every line of every error message.

FileErrorHandler (`FILE` **f*, `const String &`*prefix* `= ""`) — Constructor on `FileErrorHandler`

Constructs a `FileErrorHandler` that prints error messages to file *f*. If *prefix* is nonempty, then every line of every error message is prepended by *prefix*.

2.7.6 Error Veneers

Error veneers wrap around basic `ErrorHandler` objects and change how error text is generated. An error veneer generally changes each error message's text in some way, perhaps by adding a context message or some indentation. It then passes the altered text to the basic `ErrorHandler` for printing. Error veneers can be easily nested.

The first argument to each error veneer constructor is a pointer to another `ErrorHandler` object. The veneer will pass altered error text to this handler, the *base handler*, for further processing and printing. It also delegates `nwarnings()` and `nerrors()` calls to the base handler.

Click comes with three error veneers: one for adding context, one for prepending text to every line, and one for supplying missing landmarks. It is easy to write others; see Section 2.7.7 [Writing ErrorHandlers], page 19, for details.

ContextErrorHandler Constructor on `ContextErrorHandler`
(`ErrorHandler *`*base_errh*`, const String &`*context*`,`
`const String &`*indent* `= "  ")`

Constructs a `ContextErrorHandler` with *base_errh* as base.

The first time this handler receives an error message, it will precede the message with the *context* string—generally more detailed information about where the error has occurred. Every line in every received error message is prepended with *indent*, two spaces by default, to set off the message from its context.

PrefixErrorHandler Constructor on `PrefixErrorHandler`
(`ErrorHandler *`*base_errh*`, const String &`*prefix*)

Constructs a `PrefixErrorHandler` with *base_errh* as base.

This handler precedes every line of every error message with *prefix*.

LandmarkErrorHandler Constructor on `LandmarkErrorHandler`
(`ErrorHandler *`*base_errh*`, const String &`*landmark*)

Constructs a `LandmarkErrorHandler` with *base_errh* as base.

This handler supplies *landmark* in place of any blank landmark passed to it. This will cause the base handler to include *landmark* in its error message.

To demonstrate these veneers in practice, we'll use the following function, which prints two error messages:

```
void f(ErrorHandler *errh) {
    errh->error("First line\nSecond line");
    errh->lwarning("here", "Third line");
}
```

A simple `FileErrorHandler` shows the base case.

```
FileErrorHandler errh1(stderr);
f(&errh1);
    ⊣ First line
    ⊣ Second line
    ⊣ here: warning: Third line
```

The simplest error veneer, `PrefixErrorHandler`, just prepends text to every line.

```
PrefixErrorHandler errh2(&errh1, "prefix - ");
f(&errh2);
     ⊣ prefix - First line
     ⊣ prefix - Second line
     ⊣ prefix - here: warning: Third line
```

`ContextErrorHandler` supplies a line of context before the first error message, and indents all messages except the context.

```
ContextErrorHandler errh3(&errh1, "This was called from ...", "** ");
f(&errh3);
     ⊣ This was called from ...
     ⊣ ** First line
     ⊣ ** Second line
     ⊣ here: ** warning: Third line
```

Note that the indentation '`** `' is printed after the landmark. This often looks better than the alternative.

Of course, an error veneer can take another error veneer as its "base handler", leading to cumulative effects.

```
ContextErrorHandler errh4(&errh2, "This was called from ...", "** ");
f(&errh4);
     ⊣ prefix - This was called from ...
     ⊣ prefix - ** First line
     ⊣ prefix - ** Second line
     ⊣ prefix - here: ** warning: Third line
```

2.7.7 Writing `ErrorHandlers`

`ErrorHandler` constructs an error message using three virtual functions. The first, `make_text`, parses a format string and argument list into a single `String`. This is passed to the second function, `decorate_text`, which may transform the string. The final function, `handle_text`, prints the resulting error message. This structure makes `ErrorHandler` easy to extend. To write a new basic `ErrorHandler`, you will need to override just `handle_text` and the counting functions (`nwarnings`, `nerrors`, and `reset_counts`). The `ErrorVeneer` helper class, described below, lets you override just `decorate_text` when writing an error veneer.

`virtual String` **make_text** (`Seriousness` *s*, `const char *`*format*, `va_list` *val*) — Method on `ErrorHandler`

Parses the format string *format* with arguments from *val*, returning the results as a *String* object.

The default implementation processes the formatting escapes described above (see Section 2.7.3 [Error Format Strings], page 15). It also prepends every line of the error message with '`warning: `' if *s* equals `ERR_WARNING`.

`virtual String` **decorate_text** (`Seriousness` *s*, `const String &`*prefix*, `const String &`*landmark*, `const String &`*text*) — Method on `ErrorHandler`

Decorates the error message *text* as appropriate and returns the result. At minimum, every line of the result should be prepended by *prefix* and, if it is nonempty, the landmark string *landmark*.

The default implementation creates lines like this:

prefixlandmark: *text*	(if *landmark* is nonempty)
prefixtext	(if *landmark* is empty)

Any spaces and/or a final colon are stripped from the end of *landmark*. Special landmarks, which begin and end with a backslash '\', are ignored.

`virtual void` **handle_text** (`Seriousness` *s*, `const String &`*text*) — Method on `ErrorHandler`

This method is responsible for printing or otherwise informing the user about the error message *text*. If *s* equals `ERR_FATAL`, the method should exit the program or perform some other drastic action. It should also maintain the `nwarnings()` and `nerrors()` counters. In most cases, it should ensure that the last character in *text* is a newline.

This method has no default implementation.

The `ErrorVeneer` class, a subclass of `ErrorHandler`, supplies default implementations for these functions that ease the construction of new error veneers. `ErrorVeneer`'s single instance variable, `ErrorHandler *_errh`, is the base handler. `ErrorVeneer` overrides all the relevant virtual functions—`nwarnings`, `nerrors`, `reset_counts`, `make_text`, `decorate_text`, and `handle_text`. Its versions simply delegate to the corresponding methods on `_errh`. An error veneer designer will generally subclass `ErrorVeneer` rather than `ErrorHandler`; then she will override only the methods she cares about (usually `decorate_text`), relying on `ErrorVeneer`'s default implementations for the rest.

ErrorVeneer (`ErrorHandler *`*base_errh*) — Constructor on `ErrorVeneer`

Constructs an `ErrorVeneer` helper class with *base_errh* as its base error handler. This constructor simply sets `_errh = base_errh`.

2.8 IPAddress

The `IPAddress` type represents an IPv4 address. It supports bitwise operations like '`&`' and '`|`' and provides methods for unparsing IP addresses into ASCII dotted-quad form.

2.8.1 Constructors

`IPAddress` objects can be constructed from network-order integers, from pointers to arrays of bytes, from ASCII strings, and from the conventional `struct in_addr` type.

IPAddress () — Method on `IPAddress`

Creates an IP address equal to 0.0.0.0.

`explicit` **IPAddress** (`const unsigned char` `*`*value*) Method on `IPAddress`

Creates an IP address equal to '*value*[0].*value*[1].*value*[2].*value*[3]'.

IPAddress (`unsigned int` *value*) Method on `IPAddress`
`explicit` **IPAddress** (`int` *value*) Method on `IPAddress`
`explicit` **IPAddress** (`long` *value*) Method on `IPAddress`
`explicit` **IPAddress** (`unsigned long` *value*) Method on `IPAddress`

Creates an IP address equal to *value*, which is an IP address in network byte order.

IPAddress (`struct in_addr` *value*) Method on `IPAddress`

Creates an IP address equal to *value*.

`explicit` **IPAddress** (`const String` `&`*text*) Method on `IPAddress`

Creates an IP address equal to *text*, which should be a dotted-quad string in ASCII. For example, *text* might equal `"18.26.4.44"`. If *text* does not parse into a dotted-quad string, the resulting `IPAddress` equals 0.0.0.0.

`IPAddress` **make_prefix** (`int` *k*) Static Method on `IPAddress`

Creates and returns an IP address with the upper *k* bits on and all other bits off. *k* must be between 0 and 32, inclusive. For example, `make_prefix(0)` equals 0.0.0.0, `make_prefix(8)` equals 255.0.0.0, and `make_prefix(32)` equals 255.255.255.255. The netmask corresponding to a CIDR address '*addr*/*k*' equals `IPAddress::make_prefix(`*k*`)`.

2.8.2 Data

These methods return an `IPAddress`'s data in a variety of ways.

operator bool () `const` Method on `IPAddress`

Returns true if and only if this IP address does not equal 0.0.0.0.

`struct in_addr` **in_addr** () `const` Method on `IPAddress`
operator struct in_addr () `const` Method on `IPAddress`

Returns this IP address as a `struct in_addr` object.

`uint32_t` **addr** () `const` Method on `IPAddress`
operator uint32_t () `const` Method on `IPAddress`

Returns this IP address as an unsigned integer in network byte order.

`const unsigned char *` **data** () `const` Method on `IPAddress`
`unsigned char *` **data** () Method on `IPAddress`

Returns a pointer to this IP address's data.

`int` **mask_to_prefix_len** () `const` Method on `IPAddress`

Returns the prefix length *k* so that this IP address equals `make_prefix(`*k*`)`, or −1 if there is no such prefix length.

unsigned **hashcode** (IPAddress *addr*) Function

Returns a number with the property that, for any two equal IP addresses *addr1* and *addr2*, hashcode(*addr1*) == hashcode(*addr2*). With this function, IPAddresses may be used as keys for HashMaps and BigHashMaps (see Section 2.5 [HashMap], page 13).

2.8.3 Operations

bool **operator==** (IPAddress *addr1*, IPAddress *addr2*) Function

Returns true if and only if *addr1* equals *addr2*. Equivalent to *addr1*.addr() == *addr2*.addr().

bool **operator!=** (IPAddress *addr1*, IPAddress *addr2*) Function

Returns true if and only if *addr1* does not equal *addr2*.

bool **matches_prefix** (IPAddress *addr1*, IPAddress *mask*) const Method on IPAddress

Returns true if and only if this IPAddress matches the IP prefix specified by *addr1* and the netmask *mask*. Equivalent to (addr() & *mask*.addr()) == *addr1*.addr().

bool **submask** (IPAddress *mask*) const Method on IPAddress

Returns true if and only if this IPAddress, interpreted as a netmask, is at least as specific as *mask*. Equivalent to (addr() & *mask*.addr()) == *mask*.addr().

IPAddress **operator&** (IPAddress *addr1*, IPAddress *addr2*) Function

Returns a new IP address equal to *addr1* masked by *addr2*. Equivalent to IPAddress(*addr1*.addr() & *addr2*.addr()).

IPAddress **operator|** (IPAddress *addr1*, IPAddress *addr2*) Function

Returns a new IP address equal to the bitwise or of *addr1* and *addr2*. Equivalent to IPAddress(*addr1*.addr() | *addr2*.addr()).

IPAddress **operator~** (IPAddress *addr*) Function

Returns a new IP address equal to the bitwise complement of *addr*. Equivalent to IPAddress(~*addr1*.addr()).

IPAddress & **operator&=** (IPAddress *addr1*) Method on IPAddress

Masks this IP address by *addr1* and returns the result. Equivalent to *this = (*this & *addr1*).

IPAddress & **operator|=** (IPAddress *addr1*) Method on IPAddress

Bitwise-ors this IP address with *addr1* and returns the result. Equivalent to *this = (*this | *addr1*).

2.8.4 Unparsing

These functions unparse IP addresses, IP netmasks, and address/netmask pairs into conventional ASCII text form.

`String` **unparse** () `const` — Method on `IPAddress`
`String` **s** () `const` — Method on `IPAddress`
operator String () `const` — Method on `IPAddress`

Unparses this IP address into dotted-quad ASCII form and returns the result as a `String` object. A sample result might be `"18.26.4.9"`.

`String` **unparse_mask** () `const` — Method on `IPAddress`

Unparses this IP address as a netmask. If the IP address equals `make_prefix(`*k*`)` for some *k*, then the result is the ASCII decimal representation of *k*. Otherwise, it is just the dotted-quad ASCII form of the IP address. For example, `IPAddress("18.26.4.9").unparse_mask()` equals `"18.26.4.9"`, but `IPAddress("255.0.0.0").unparse_mask()` equals `"8"`.

`String` **unparse_with_mask** (`IPAddress` *mask*) `const` — Method on `IPAddress`

Unparses this IP address with *mask* as its netmask. The result has the form `"`*addrtext/masktext*`"`, where *addrtext* equals `this->unparse()` and *masktext* equals *mask*`.unparse_mask()`.

2.9 IP6Address

3 Packets

The Packet class represents Click packets. The single Packet interface has multiple implementations, one per driver. Inside the Linux kernel driver, a Packet object is equivalent to a Linux sk_buff structure; most Packet methods are inline functions that expand to sk_buff calls. The user-level driver, however, uses a purpose-built Packet implementation.

Click packets separate header information from data. The Packet * pointer points to a header structure, which holds pointers to the actual packet data and a set of *annotations*. Packet data may be shared by two or more packet headers. Packet headers, however, should never be shared.

Packets come in two flavors, Packet and WritablePacket. A Packet object represents a packet header whose data might be shared with other packets. Because of this potential sharing, Packet data is read-only, and its methods return const pointers to data. A WritablePacket object, in contrast, represents a packet header whose data is known not to be shared. Its methods return non-const pointers. The uniqueify method turns a Packet into a WritablePacket, possibly by making a copy of the packet's data. WritablePacket is a subclass of Packet, so you can turn a WritablePacket into a Packet implicitly.

The Packet and WritablePacket classes are defined in '<click/packet.hh>'.

3.1 Structure and Contents

Packet data is stored in a single flat array of bytes. There is no support for linked chains à la BSD mbuf. The actual packet data is embedded inside a buffer that may be much larger, leaving unused spaces called *headroom* and *tailroom* before and after the data proper. Therefore, tasks like prepending a header need not always reallocate memory. If the headroom is big enough, prepending space for a new header just requires bumping a pointer.

This diagram shows how a typical packet is laid out, with the relevant Packet methods' names.

```
                 data
     |<- headroom ->|<----- length ----->|<- tailroom ->|
     |              |                    |              |
     +==============+====================+==============+
     |XXXXXXXXXXXXXX| PACKET CONTENTS... |XXXXXXXXXXXXXX|
     +==============+====================+==============+
     |                                                  |
     |<---------------- buffer_length ----------------->|
buffer_data
```

And here are those methods' descriptions.

const unsigned char * **data** () const — Method on Packet

Returns a pointer to the packet data proper.

unsigned **length** () const — Method on Packet

Returns the length of the packet data proper.

const unsigned char * **buffer_data** () const — Method on Packet

Returns a pointer to the buffer that contains the packet data.

unsigned **headroom** () const — Method on Packet
unsigned **tailroom** () const — Method on Packet
unsigned **buffer_length** () const — Method on Packet

Returns the length of the headroom area, the tailroom area, and the whole buffer, respectively.

unsigned char * **data** () const — Method on WritablePacket
unsigned char * **buffer_data** () const — Method on WritablePacket

These WritablePacket methods are identical to Packet's data and buffer_data methods except for their non-const return type.

Two invariants relate these methods' values:

```
buffer_length() = headroom() + length() + tailroom()
data() = buffer_data() + headroom()
```

3.2 Creation and Destruction

Packets are created with the Packet::make static methods, and destroyed with the Packet::kill method. (The Packet and WritablePacket classes have private constructors and destructors; you cannot create or destroy packets with new or delete.)

3.2.1 Packet::make

The make methods always take the length of the packet data; some of them take the packet contents and the headroom and tailroom lengths as well. (The contents of any headroom and tailroom areas are always undefined.) Most of them return a WritablePacket *, since new packets are not shared.

The Packet class defines two constants related to packet creation, DEFAULT_HEADROOM and MIN_BUFFER_LENGTH. Those make methods that do not take an explicit headroom parameter use DEFAULT_HEADROOM instead. Furthermore, no make method will create a packet with buffer length less than MIN_BUFFER_LENGTH. If the sum of a packet's headroom and length is less than this, the packet buffer is given extra tailroom to bump the buffer length up to MIN_BUFFER_LENGTH. These constants have the values DEFAULT_HEADROOM = 28 and MIN_BUFFER_LENGTH = 64.

WritablePacket * **make** (unsigned *len*) — Static Method on Packet

Returns a new packet containing *len* bytes of undefined data.

WritablePacket * **make** (const char **data*, unsigned *len*) — Static Method on Packet
WritablePacket * **make** (const unsigned char **data*, unsigned *len*) — Static Method on Packet

Returns a new packet whose contents equal the first *len* bytes of *data*. *data* may be a null pointer, in which case the packet contains *len* bytes of undefined data.

WritablePacket * **make** (unsigned *headroom*, const unsigned char **data*, unsigned *len*, unsigned *tailroom*) — Static Method on Packet

Returns a new packet containing *headroom* bytes of headroom, *len* bytes of contents, and at least *tailroom* bytes of tailroom. The packet contents will equal the first *len* bytes of *data* unless *data* is a null pointer, in which case the contents are undefined.

The following make method is only available in the user-level driver.

WritablePacket * **make** (unsigned char **data*, unsigned *len*, void (**destructor*)(unsigned char *, size_t)) — Static Method on Packet

Returns a new packet that uses *data* as a buffer. That is, the new packet will have the following characteristics:

buffer_data	*data*
buffer_length	*len*
headroom	0
length	*len*
tailroom	0

When the resulting packet is destroyed, the function *destructor* will be called with *data* and *len* as arguments. *destructor* may be a null pointer, in which case Packet calls delete[] *data* instead.

This method lets a user-level element manage packet memory itself, rather than relying on Packet.

See Section 3.3 [Packets and sk_buffs], page 26, for a make method only available in the Linux kernel driver.

3.2.2 Packet::kill

To destroy a Packet, simply call its kill method.

void **kill** () — Method on Packet

Frees this packet. If this packet contained the last reference to its data buffer, then frees the data buffer as well.

3.3 Packets and sk_buffs

In the Linux kernel driver, Packet objects are equivalent to struct sk_buffs. This avoids indirection overhead and makes it cheap to pass packets back and forth between Linux and Click. The Packet operations described in this section are mostly inline functions that expand to conventional sk_buff calls like skb_clone().

Click Packet sk_buffs should always have skb->users equal to 1. That is, the sk_buff headers should not be shared, although the data buffers they point to may be shared.

The make, skb, and steal_skb methods described in this section convert Packets to sk_buffs and vice versa.

Packet * **make** (struct sk_buff **skb*) Static Method on Packet

Returns a new packet equivalent to the sk_buff *skb*. All of *skb*'s data pointers and annotations are left unchanged. This method generally does nothing, since Packets and sk_buffs are equivalent in the Linux kernel. However, if *skb*->users field is bigger than 1, the method will return a clone of *skb*. This method returns a Packet *, not a WritablePacket *, because the *skb* argument might share data with some other sk_buff.

Do not use or manipulate *skb* after passing it to this method, since Click and the Packet implementation now own *skb*.

struct sk_buff * **skb** () Method on Packet
const struct sk_buff * **skb** () const Method on Packet

Returns the sk_buff corresponding to this packet. Use this method to examine the sk_buff version of a Packet.

Do not pass the result to a function that might free it or increment its users field; use steal_skb for that.

struct sk_buff * **steal_skb** () Method on Packet

Returns the sk_buff corresponding to this packet. Use this method to permanently change a Packet into an sk_buff—for example, to create an sk_buff you'd like to send to Linux.

Do not use or manipulate a Packet after calling its steal_skb method, since Linux now owns the resulting sk_buff.

3.4 Sharing—clone and uniqueify

The clone method creates a new packet header that shares data with an existing packet. The uniqueify method, in contrast, ensures that a packet's data is not shared by anyone, perhaps by making a copy of the data.

Packet * **clone** () Method on Packet

Creates and returns a new packet header that shares this packet's data. The new packet's annotations are copied from this packet's annotations.

The result may be a null pointer if there was not enough memory to make a new packet header.

WritablePacket * **uniqueify** () Method on Packet

Ensures that this packet does not share data with any other packet. This may involve copying the packet data, and perhaps creating a new packet header, but if this packet is already unshared, no real work is required. Returns a WritablePacket * because the new packet is unshared.

Do not use, manipulate, or free a Packet after calling its uniqueify method. Manipulate the returned WritablePacket * instead.

The result may be a null pointer if there was not enough memory to make a required data copy. In this case, the old packet is freed.

`bool` **shared** () `const` Method on `Packet`

Returns true if and only if this packet shares data with some other packet.

3.5 Buffer Manipulation—`push`, `pull`, `put`, and `take`

The `push`, `pull`, `put`, and `take` methods manipulate a packet's contents by adding or removing space from its headroom or tailroom. Given a packet, use `push` to add space to its beginning, `pull` to remove space from its beginning, `put` to add space to its end, and `take` to remove space from its end. The methods that add space, `push` and `put`, uniqueify the relevant packet as a side effect. This ensures that the packet's data is unshared so you can immediately manipulate the added space.

`WritablePacket *` **push** (`unsigned` *amt*) Method on `Packet`

Adds *amt* bytes of space to the beginning of the packet's data and returns the resulting packet. The new space is uninitialized. The result will not share data with any other packet; thus, it is a `WritablePacket *`. If this packet is unshared and its headroom is bigger than *amt*, then this operation is cheap, amounting to a bit of pointer arithmetic. Otherwise, it requires copying the packet data and possibly creating a new packet header.

Do not use, manipulate, or free a `Packet` after calling its `push` method. Manipulate the returned `WritablePacket *` instead.

The result may be a null pointer if there was not enough memory to make a required new packet. In this case, the old packet is freed.

`void` **pull** (`unsigned` *amt*) Method on `Packet`

Removes *amt* bytes of space from the beginning of the packet's data. *amt* must be less than or equal to the packet's `length()`. This operation is always cheap, amounting to a bit of pointer arithmetic.

`WritablePacket *` **put** (`unsigned` *amt*) Method on `Packet`

Adds *amt* bytes of space to the end of the packet's data and returns the resulting packet. The new space is uninitialized. The result will not share data with any other packet; thus, it is a `WritablePacket *`. If this packet is unshared and its tailroom is bigger than *amt*, then this operation is cheap, amounting to a bit of pointer arithmetic. Otherwise, it requires copying the packet data and possibly creating a new packet header.

Do not use, manipulate, or free a `Packet` after calling its `put` method. Manipulate the returned `WritablePacket *` instead.

The result may be a null pointer if there was not enough memory to make a required new packet. In this case, the old packet is freed.

`void` **take** (`unsigned` *amt*) Method on `Packet`

Removes *amt* bytes of space from the end of the packet's data. *amt* must be less than or equal to the packet's `length()`. This operation is always cheap, amounting to a bit of pointer arithmetic.

The `push` and `put` methods have "nonunique" variants, `nonunique_push` and `nonunique_put`, which do not have the side effect of uniqueifying their resulting packet. These methods are rarely used.

`Packet *` **nonunique_push** (`unsigned` *amt*) Method on `Packet`

Adds *amt* bytes of space to the beginning of the packet's data and returns the resulting packet. The new space is uninitialized. The result may share data with other packets. If this packet's headroom is bigger than *amt*, then this operation is cheap, amounting to a bit of pointer arithmetic. Otherwise, it requires copying the packet data and possibly creating a new packet header.

Do not use, manipulate, or free a `Packet` after calling its `nonunique_push` method. Manipulate the returned `Packet *` instead.

The result may be a null pointer if there was not enough memory to make a required new packet. In this case, the old packet is freed.

`Packet *` **nonunique_put** (`unsigned` *amt*) Method on `Packet`

Adds *amt* bytes of space to the end of the packet's data, returning the resulting packet. The new space is uninitialized. The result may share data with other packets. If this packet's tailroom is bigger than *amt*, then this operation is cheap, amounting to a bit of pointer arithmetic. Otherwise, it requires copying the packet data and possibly creating a new packet header.

Do not use, manipulate, or free a `Packet` after calling its `nonunique_put` method. Manipulate the returned `Packet *` instead.

The result may be a null pointer if there was not enough memory to make a required new packet. In this case, the old packet is freed.

3.6 Annotations

Each packet header has space for a number of *annotations*, extra information about the packet that is not contained in its data. Click supports *header annotations*, which indicate where in the packet a network header, such as an IP header, is located; *user annotations*, whose semantics are left undefined by Click—different elements can treat them in different ways; and other specialized annotations, such as the *timestamp annotation*, the *destination IP address annotation*, and so forth.

New packets begin with all annotations cleared: numeric annotations are zero, pointer annotations are null pointers. `clone`, `uniqueify`, and their equivalents always copy each of the original packet's annotations in the appropriate way. (For example, the new header annotations will point into the new data, if a data copy was made.)

3.6.1 Header Annotations

Many packets contain a network header of some kind, such as an IP header. This header may be located anywhere in the packet depending on how the packet was encapsulated. Furthermore, the data encapsulated by that network header may be located anywhere after the network header, given the presence of options. With the *network header annotation* and the *transport header annotation*, one element can determine where a network header is and how long it is, then store this information for other elements to use. For example, the *CheckIPHeader* element sets the header annotations on packets it receives. Elements like *SetIPDSCP* then require a non-null IP header annotation on their input packets.

The header annotations on new packets are each set to a null pointer.

`const unsigned char *` **network_header** () `const` — Method on `Packet`
`unsigned char *` **network_header** () `const` — Method on `WritablePacket`

Returns the network header annotation. The resulting pointer is read-only on `Packets` and read/write on `WritablePackets`.

`const unsigned char *` **transport_header** () `const` — Method on `Packet`
`unsigned char *` **transport_header** () `const` — Method on `WritablePacket`

Returns the transport header annotation. The resulting pointer is read-only on `Packets` and read/write on `WritablePackets`.

`int` **network_header_offset** () `const` — Method on `Packet`

Returns the offset from `data()` to `network_header()`. The result might be negative, since the `data` pointer may have been advanced past the network header annotation with the `pull` method.

`int` **network_header_length** () `const` — Method on `Packet`

Returns the network header's length. This equals `transport_header()` − `network_header()`.

`unsigned` **transport_header_offset** () `const` — Method on `Packet`

Returns the offset from `data()` to `transport_header()`. The result might be negative, since the `data` pointer may have been advanced past the transport header annotation with the `pull` method.

Several invariants relate these methods' values whenever the header annotations are non-null:

```
buffer_data() ≤ network_header() ≤ transport_header()
        ≤ buffer_data() + buffer_length()
network_header_offset() = network_header() − data()
transport_header_offset() = transport_header() − data()
network_header_length() = transport_header() − network_header()
```

Set the network and transport header annotations simultaneously with the `set_network_header` method.

`void` **set_network_header** (`const unsigned char *`*header*, `unsigned` *len*) — Method on `Packet`

Sets the network header annotation to *header*, which must lie between `buffer_data()` and `buffer_data()` + `buffer_length()`. The network header is *len* bytes long, so `network_header_length()` will equal *len* and `transport_header()` will equal *header* + *len*.

3.6.1.1 Typed Header Annotations

For convenience, `Packet` provides methods for accessing and setting the network header annotation as an IP or IPv6 header. These methods use the same annotations as the generic `network_header` methods; they are just typed differently.

`const click_ip *` **ip_header** () `const` — Method on `Packet`
`click_ip *` **ip_header** () `const` — Method on `WritablePacket`
`const click_ip6 *` **ip6_header** () `const` — Method on `Packet`
`click_ip6 *` **ip6_header** () `const` — Method on `WritablePacket`

Returns `network_header()` as a pointer to an IP or IPv6 header structure.

`int` **ip_header_offset** () `const` — Method on `Packet`
`unsigned` **ip_header_length** () `const` — Method on `Packet`
`int` **ip6_header_offset** () `const` — Method on `Packet`
`unsigned` **ip6_header_length** () `const` — Method on `Packet`

Equivalent to `network_header_offset()` and `network_header_length()`.

`void` **set_ip_header** (`const click_ip *`*header*, `unsigned` *len*) — Method on `Packet`
`void` **set_ip6_header** (`const click_ip6 *`*header*, `unsigned` *len*) — Method on `Packet`

Equivalent to `set_network_header(`*header*`,` *len*`)`.

`void` **set_ip6_header** (`const click_ip6 *`*header*) — Method on `Packet`

Equivalent to `set_ip6_header(`*header*`, 40)`.

`const click_tcp *` **tcp_header** () `const` — Method on `Packet`
`click_tcp *` **tcp_header** () `const` — Method on `WritablePacket`
`const click_udp *` **udp_header** () `const` — Method on `Packet`
`click_udp *` **udp_header** () `const` — Method on `WritablePacket`

Returns `transport_header()` as a pointer to a TCP or UDP header structure.

3.6.2 User Annotations

Each packet header has a *user annotation area*, space reserved for arbitrary annotations. Different methods access this space as an array of bytes, integers, or unsigned integers. The `Packet` class does not assign semantics to any particular byte in the user annotation area. Instead, macros in '`<click/packet_anno.hh>`' provide names for particular bytes. Some of these names have overlapping byte ranges; the user must take care not to define a configuration whose elements use an annotation byte on a packet for different purposes. The next section describes the macros in Click's default '`<click/packet_anno.hh>`'.

These constants define the size of the annotation area.

`Packet::USER_ANNO_SIZE`
: The size of the annotation area in bytes.

`Packet::USER_ANNO_US_SIZE`
: The size of the annotation area in `unsigned short`s.

`Packet::USER_ANNO_S_SIZE`
: The size of the annotation area in `short`s.

`Packet::USER_ANNO_U_SIZE`
: The size of the annotation area in `unsigned int`s.

`Packet::USER_ANNO_I_SIZE`
: The size of the annotation area in `int`s.

Currently, `USER_ANNO_SIZE` is 24, `USER_ANNO_U_SIZE` and `USER_ANNO_I_SIZE` are both 6, and `USER_ANNO_US_SIZE` and `USER_ANNO_S_SIZE` are both 12.

The user annotation area may be accessed as an array of bytes, an array of `unsigned int`s, or an array of `int`s. The elements of these arrays are numbered from 0 to $k-1$, where k is the appropriate `SIZE` constant.

`unsigned char` **user_anno_c** (`int` *i*) `const` — Method on `Packet`

Returns the *i*th byte in the user annotation area. *i* must be between 0 and `USER_ANNO_SIZE` − 1.

`unsigned` **user_anno_u** (`int` *i*) — Method on `Packet`
`int` **user_anno_i** (`int` *i*) — Method on `Packet`

Returns the *i*th `unsigned int` or `int` in the user annotation area. *i* must be between 0 and `USER_ANNO_U_SIZE` − 1. The *i*th `unsigned int` or `int` annotation occupies bytes $4i$ through $4i+3$ of the user annotation area.

`void` **set_user_anno_c** (`int` *i*, `unsigned char` *value*) — Method on `Packet`
`void` **set_user_anno_u** (`int` *i*, `unsigned` *value*) — Method on `Packet`
`void` **set_user_anno_i** (`int` *i*, `int` *value*) — Method on `Packet`

Sets the *i*th byte, `unsigned int`, or `int` user annotation to *value*.

`unsigned *` **all_user_anno_u** () — Method on `Packet`

Returns a pointer to the user annotation area, treated as an array of `unsigned int`s.

3.6.3 Specific User Annotations

The '`<click/packet_anno.hh>`' header file defines macros for accessing a packet's user annotation area by name. These macros follow some simple guidelines. Each user annotation is given a name like '`PAINT`' or '`FIX_IP_SRC`'. Then, two macros are written for each annotation, *name*`_ANNO` and `SET_`*name*`_ANNO`.

*name*_**ANNO** (`const Packet *`*p*) — Macro

Returns the value of *p*'s *name* annotation.

SET__name_**_ANNO** (`Packet *`*p*, *value*) — Macro

Sets *p*'s *name* annotation to *value*.

For example, here are the definitions of `PAINT_ANNO` and `SET_PAINT_ANNO` from Click's default '`<click/packet_anno.hh>`'.

```
#define PAINT_ANNO(p)           ((p)->user_anno_c(0))
#define SET_PAINT_ANNO(p, v)    ((p)->set_user_anno_c(0, (v)))
```

This table lists the annotations declared in Click's default '`<click/packet_anno.hh>`'.

Annotation name	Type	Bytes	Some relevant elements
`PAINT`	`unsigned char`	0	*Paint*, *CheckPaint*, *PaintTee*
`ICMP_PARAM_PROB`	`unsigned char`	1	*IPGWOptions*, *ICMPError*
`FIX_IP_SRC`	`unsigned char`	3	*ICMPError*, *FixIPSrc*
`FWD_RATE`	`int`	4–7	*IPRateMonitor*
`REV_RATE`	`int`	8–11	*IPRateMonitor*

3.6.4 Other Annotations

Packet headers have space for four other particular annotations, and special methods for accessing them. These annotations do not overlap the user annotation area. There are annotations that hold a destination IP address, a timestamp, the device on which the packet arrived, a packet type constant, and, in the Linux kernel module, a performance counter value.

3.6.4.1 Destination Address

The destination address annotation stores the IP or IPv6 address of the next hop towards the packet's destination. Elements check and manipulate this address, rather than the IP header's destination address, since the next-hop address often differs from the final destination. The destination IP address and IPv6 address are different annotations, but they overlap; you may set only one at a time.

`IPAddress` **dst_ip_anno** () `const` — Method on `Packet`

Returns this packet's destination IP address annotation.

`const IPAddress &` **dst_ip6_anno** () `const` — Method on `Packet`

Returns a reference to this packet's destination IPv6 address annotation.

`void` **set_dst_ip_anno** (`IPAddress` *value*) — Method on `Packet`
`void` **set_dst_ip6_anno** (`const IP6Address &`*value*) — Method on `Packet`

Sets this packet's destination IP or IPv6 address annotation to *value*.

The destination IP address annotation is set by the *GetIPAddress* and *SetIPAddress* elements, manipulated by *LookupIPRoute* and its cousins, and used by *ARPQuerier*. It defaults to zero.

3.6.4.2 Timestamp

The timestamp annotation generally indicates when a packet was received.

`const struct timeval &` **timestamp_anno** `()` `const` — Method on `Packet`
`struct timeval &` **timestamp_anno** `()` — Method on `Packet`

Returns a reference to this packet's timestamp annotation.

`void` **set_timestamp_anno** `(const struct timeval &`*value*`)` — Method on `Packet`

Sets this packet's timestamp annotation to *value*.

`void` **set_timestamp_anno** `(int` *sec*`, int` *usec*`)` — Method on `Packet`

Sets this packet's timestamp annotation to *sec* and *usec*. Equivalent to `struct timeval tv; tv.tv_sec =` *sec*`; tv.tv_usec =` *usec*`; set_timestamp_anno(tv)`.

Linux device drivers set this annotation, so packets emitted by *FromDevice* and *PollDevice* in the Linux kernel driver have the annotation set. Packet sources like *InfiniteSource* and *RatedSource* also set the annotation, as does *FromDump* in the user-level driver. Debugging elements like *Print* generally take a keyword argument that makes them print packet timestamps.

The timestamp annotation defaults to zero.

3.6.4.3 Device

In the Linux kernel, packets received from some device are annotated with a pointer to the relevant `struct net_device` object. (In versions of the kernel prior to 2.3, this type was called `struct device`.) The `Packet` class provides access to this annotation. The annotation has type `net_device *`; Click defines `net_device` as a synonym for `struct device` in kernel versions 2.2 and prior.

`net_device *` **device_anno** `()` `const` — Method on `Packet`

Returns this packet's device annotation.

`void` **set_device_anno** `(net_device *`*value*`)` — Method on `Packet`

Sets this packet's device annotation to *value*.

In the user-level driver, `device_anno` always returns 0, and `set_device_anno` does nothing.

The *ARPResponder* element sets this annotation on every generated response to the value of the annotation on the relevant query. Because of this, those responses can be safely forwarded to Linux: Linux's ARP-response code requires a correct device annotation.

The device annotation defaults to a null pointer.

3.6.4.4 Packet Type

The packet type annotation specifies how a packet was received. Its value is one of the following constants, which are defined in the `Packet::PacketType` enumeration.

‘`HOST`’ The packet was sent to this host.

‘`BROADCAST`’
: The packet was sent to a link-level broadcast address.

‘`MULTICAST`’
: The packet was sent to a link-level multicast address.

‘`OTHERHOST`’
: The packet was sent to a different host, but received anyway. The relevant device is probably in promiscuous mode.

‘`OUTGOING`’
: The packet was generated at this host and is being sent to another host.

‘`LOOPBACK`’, ‘`FASTROUTE`’
: See the Linux kernel documentation. These values correspond to ‘`PACKET_LOOPBACK`’ and ‘`PACKET_FASTROUTE`’, which are defined in ‘`<linux/if_packet.h>`’.

`Packet::PacketType` **packet_type_anno** () `const` — Method on `Packet`
: Returns this packet’s packet type annotation.

`void` **set_packet_type_anno** (`Packet::PacketType` *value*) — Method on `Packet`
: Sets this packet’s packet type annotation to *value*.

In the Linux kernel, device drivers set the packet type annotation for the packets they receive. Thus, the *FromDevice* and *PollDevice* elements generate packets with correct packet type annotations. The user-level driver’s *FromDevice* also sets the packet type annotation. The *ICMPError* and *DropBroadcasts* elements use the annotation’s value.

The packet type annotation defaults to `Packet::HOST`.

3.6.4.5 Performance Counter

This annotation is available only in the Linux kernel driver. Its value is an `unsigned long long` that generally corresponds to some performance counter value.

`unsigned long long` **perfctr_anno** () `const` — Method on `Packet`
: Returns this packet’s performance counter annotation.

`void` **set_perfctr_anno** (`unsigned long long` *value*) — Method on `Packet`
: Sets this packet’s performance counter annotation to *value*.

The *SetCycleCount*, *SetPerfCount*, *CycleCountAccum*, and *PerfCountAccum* elements manipulate this annotation. Its default value is zero.

3.6.5 Annotations In General

`Packet` provides methods for clearing a packet's annotations, and for copying all of a packet's annotations from another packet.

`void` **clear_annotations** () — Method on `Packet`

Clears all of this packet's annotations to their default state, which is generally zero.

`void` **copy_annotations** (`const Packet *`*p*) — Method on `Packet`

Copies all of *p*'s annotations into this packet except for its header annotations. (This packet's current header annotations are left unchanged.)

3.7 Out-of-Memory Conditions

Any method that potentially allocates memory for a `Packet` may fail due to an out-of-memory condition. The complete list of these methods follows:

- `make` variants
- `clone`
- `uniqueify`
- `push`
- `put`
- `nonunique_push`
- `nonunique_put`

These methods always return a null pointer on out-of-memory. Methods that manipulate existing packets—`uniqueify`, `push`, `put`, `nonunique_push`, and `nonunique_put`—additionally free any existing packet before returning a null pointer. You should always check the results of these methods to see if you have run out of memory.

4 Element Characteristics

4.1 Element Class

Every element belongs to a single element class, and every element class has a name. The `class_name` virtual function returns that name.

`virtual const char *` **class_name** () `const` Method on `Element`

Returns the element's class name as a null-terminated C string. This method has no default implementation; every element must supply a definition.

The `class_name` method should be declared on a single line in the element's class definition, and should return a C string constant. This makes the element's class name easy to automatically extract from the source code.

Here is a typical `class_name` method.

```
class ARPQuerier : public Element { public: // ...
    const char *class_name() const { return "ARPQuerier"; }
}
```

Click creates new element objects by calling their default, zero-argument constructors. The resulting element should not be configured or initialized. It will be configured independently through element initialization methods; see Chapter 5 [Element Initialization], page 42, for more information.

4.2 Casting

Each element conforms to one or more named *interfaces*. Each element class is an interface, whose name is just the element class name, but the user can create additional interfaces at will. Generally, these interfaces export functionality that elements may be interested in, but that is not specific to any one element class. For example, the *Storage* interface provides information about how many packets are stored in an element; elements that implement this interface include *Queue*, *FrontDropQueue*, and *FromDevice*. Elements interested in packet storage, such as *RED*, then look for *Storage* elements, making them independent of any particular storage strategy.

A caller can discover whether an element implements a particular interface by calling its `cast` method. This method takes an interface name and returns a non-null pointer if and only if the element implements that interface.

`virtual void *` **cast** (`const char *`*name*) Method on `Element`

The *name* argument is an interface name, represented as a null-terminated C string. If this element implements the *name* interface, `cast` should return a pointer to the corresponding data. If it does not, `cast` should return a null pointer.

The default representation returns `this` if *name* equals the element's `class_name()`, or a null pointer if it does not.

Some care is required when one element class is a subclass of another. Say that element class `Derived` is a subclass of `Base`. Then `Derived`'s `cast` method should return a non-null pointer when passed either `"Derived"`, `"Base"`, or any additional interfaces that `Derived` or `Base` might implement. Here is a first try at `Derived`'s `cast` implementation:

```
void *
Derived::cast(const char *name)
{
    if (strcmp(name, "Derived") == 0)
        return (Derived *)this;
    else // rely on Base::cast to check for "Base"
        return Base::cast(name);
}
```

This code is correct and preferred as long as `Base` has its own `cast` implementation. Unfortunately, if `Base` took advantage of `cast`'s default implementation, which uses `class_name`, the code is broken. Since a `Derived` element's `class_name` method returns `"Derived"`, the default `cast` method will check only for `"Derived"`, not for `"Base"` as we wished. The solution is either to write an explicit `cast` method for `Base`, or to write `Derived::cast` differently—like so, for example:

```
void *
Derived::cast(const char *name)
{
    if (strcmp(name, "Derived") == 0)
        return (Derived *)this;
    else if (strcmp(name, "Base") == 0)
        return (Base *)this;
    else
        return 0;
}
```

Always explicitly cast `this` to the correct type before returning it. This is important because of multiple inheritance, where the value of a pointer to a supertype may be different from the value of `this`. (The type system generally determines when pointer arithmetic is necessary, but the `void *` return type hides this type information from `cast`'s caller.)

We encourage you to write simple `cast` methods that compare the *name* argument against a set of fixed strings. Arbitrary computation inside *cast* is discouraged; we may eventually want to analyze `cast` definitions.

Click uses a `cast` method rather than C++'s standard `dynamic_cast` mechanism because it's difficult to use `dynamic_cast` in the Linux kernel.

4.3 Names

Each element in a router configuration has a *name* under which it was declared and a *landmark*, a string indicating where it was declared in the configuration file.

`String` **id** () `const` — Method on `Element`

Returns the element's name.

`String` **declaration** () `const` — Method on `Element`

Returns a textual representation of the element's declaration. The result has the form '*id* `::` *cname*', where *id* is the element's `id()` and *cname* is its `class_name()`.

`String` **landmark** () `const` — Method on `Element`

Returns a string indicating where the element was declared in the configuration file. The result generally has the form '*filename*`:`*linenumber*'.

4.4 Router Relationship

Elements may be part of some router configuration, which is represented by a `Router` object. Elements in a `Router` are numbered between 0 and that router's `nelements()`; `eindex` returns that number.

`Router *` **router** () `const` — Method on `Element`

Returns the element's corresponding `Router` object.

`int` **eindex** () `const` — Method on `Element`

Returns the element's index in its router.

`int` **eindex** (`Router *`*r*) `const` — Method on `Element`

Returns the element's index in its router, if that router is *r*, or −1, if that router is not *r*. Equivalent to:

```
return (router() == r ? eindex() : -1);
```

4.5 Creating Ports

These methods return or change how many input and output ports an element has.

`int` **ninputs** () `const` — Method on `Element`
`int` **noutputs** () `const` — Method on `Element`

Returns the element's number of input or output ports.

The `set_` and `add_` methods, which add or remove ports, must be called only by the element itself. For example, the Click infrastructure never calls `set_ninputs` or `set_noutputs`. Click will inform the element how many of its ports were used in a particular router configuration; see Section 5.1 [notify_ninputs notify_noutputs], page 42.

You may change an element's number of ports only during router initialization. You may not, for example, call `set_ninputs` at run time, or even during the element's `initialize` method (see Section 5.7 [initialize], page 48). See Section 4.7 [When to Call Element Methods], page 41, for more information.

`void` **set_ninputs** (`int` *n*) `const` — Method on `Element`
`void` **set_noutputs** (`int` *n*) `const` — Method on `Element`

Sets the element's number of input or output ports to *n*, which must be greater than or equal to zero.

`void` **add_input** () `const` — Method on `Element`
`void` **add_output** () `const` — Method on `Element`

Add an input or output port to the element. Same as `set_ninputs(ninputs() + 1)` or `set_noutputs(noutputs() + 1)`.

4.6 Using Ports

Each of an element's input and output ports is represented by an `Element::Port` object. The `input` and `output` methods return the `Port` object corresponding to a given port number.

`const Port &` **input** (`int` *p*) `const` — Method on `Element`
`const Port &` **output** (`int` *p*) `const` — Method on `Element`

Returns the `Element::Port` object corresponding to the element's *p*th input or output port. *p* must be a valid port number: greater than or equal to zero and less than `ninputs()` or `noutputs()`, respectively.

The following methods return information about a port. `input_is_pull` and `output_is_push` are `Element` methods; the rest are methods on `Element::Port`. All of these methods return meaningful results only after the router has been partially initialized; see Section 4.7 [When to Call Element Methods], page 41.

`bool` **input_is_pull** (`int` *p*) `const` — Method on `Element`
`bool` **output_is_push** (`int` *p*) `const` — Method on `Element`

Returns true if input port *p* is pull or output port *p* is push, respectively. *p* must be a valid port number.

`Element *` **element** () `const` — Method on `Element::Port`

Returns the element this port is connected to, if one exists. Pull input ports and push output ports are always connected to another element; push input ports and pull output ports never are. `element()` returns a null pointer when called on a push input port or pull output port.

`int` **port** () `const` — Method on `Element::Port`

Returns the port number this port is connected to, if one exists. Pull input ports and push output ports are always connected to another port; push input ports and pull output ports never are. `port()` returns −1 when called on a push input port or pull output port.

For example, consider this router configuration.

```
x :: X; y :: Y;
x [0] -> [1] y; // push connection
```

Because `x [0]` is a push output port, `x->output(0).element()` will return `y` and `x->output(0).port()` will return 1. On the other hand, `y->input(1).element()` will return a null pointer and `y->input(1).port()` will return −1.

The `element` and `port` methods only supply local information about how elements are connected. Furthermore, they provide no information about how push input ports and pull output ports are connected. For these reasons, most elements interested in router configuration topology call `Router`'s `upstream_elements` and `downstream_elements` methods instead.

4.7 When Element Methods May Be Called

This chart shows when it is OK to call particular `Element` methods. Methods not mentioned here are generally not called by the user.

```
  Method Name                     constr notify config  init   run
+--------------------------------+------+------+------+------+------+
| class_name, cast               |  OK  |  OK  |  OK  |  OK  |  OK  |
| id, declaration, landmark      |      |  OK  |  OK  |  OK  |  OK  |
| router, eindex                 |      |  OK  |  OK  |  OK  |  OK  |
+--------------------------------+------+------+------+------+------+
| ninputs, noutputs              |  OK  |  OK  |  OK  |  OK  |  OK  |
| set_ninputs, set_noutputs      |  OK  |  OK  |  OK  |      |      |
| add_input, add_output          |  OK  |  OK  |  OK  |      |      |
+--------------------------------+------+------+------+------+------+
| input, output                  |      |      |      |  OK  |  OK  |
| input_is_pull, output_is_push  |      |      |      |  OK  |  OK  |
| Port::element, Port::port      |      |      |      |  OK  |  OK  |
+--------------------------------+------+------+------+------+------+
  Method Name                     constr notify config  init   run
```

The headings denote:

'`constr`' Construction time. This includes the element's constructor and its destructor.

'`notify`' Inside the `notify_ninputs` and `notify_noutputs` methods.

'`config`' Inside the `configure` method.

'`init`' Inside the `add_handlers`, `initialize`, and `uninitialize` methods.

'`run`' At run time. That is, inside some `push` or `pull` method, or some task or timer callback, or some handler, or some function called from one of these places.

5 Element Initialization

The process of making an element ready for inclusion in an active router is called *element initialization*. This includes processing the element's configuration string, setting up internal state and any input and output ports, and querying the router about neighboring elements.

Every element in an active router must have successfully initialized. If there is an error initializing even one element, the router is aborted. Router initialization happens in sequential phases: every element must successfully complete one phase before the next phase begins.

5.1 `notify_ninputs` and `notify_noutputs`

The router calls each element's `notify_ninputs` and `notify_noutputs` methods to tell it how many of its input and output ports were used in the configuration. A port is used if it is used in a connection.

`virtual void` **notify_ninputs** (`int` *ninputs*) Method on `Element`
`virtual void` **notify_noutputs** (`int` *noutputs*) Method on `Element`

The *ninputs* and *noutputs* arguments specify how many input and output ports were used in the configuration. For example, if *ninputs* is 5, then input ports 0 through 4 were used.[1]

These methods' default implementations do nothing.

`notify_ninputs` and `notify_noutputs` are called early in the initialization process—before `configure`, for example, and before ports are assigned to push or pull. They may create and destroy input and output ports or set other private element state.

A `notify_ninputs` or `notify_noutputs` method should generally be very short and stylized. It should call no Element methods except for possibly `set_ninputs` or `set_noutputs`. This typical `notify_noutputs` method sets the element's number of outputs to one or two, depending on how many outputs were actually used:

```
void
ARPQuerier::notify_noutputs(int n)
{
    set_noutputs(n < 2 ? 1 : 2);
}
```

There is no need to supply a `notify_ninputs` or `notify_noutputs` method if your element has a fixed number of inputs or outputs.

[1] Strictly speaking, it is possible that one or more of the lower-numbered ports were not used—for example, that input port 0 was not used by the configuration. This is always a configuration error, however. A later stage will report unused ports as errors and abort router initialization.

5.2 `configure_phase`—Initialization Order

Some elements depend on being configured and initialized before or after other elements. For example, the *AddressInfo* element must be configured before all other elements, since its address abbreviations must be available in their configuration strings. The `configure_phase` method makes this possible.

`virtual int` **configure_phase** () `const` — Method on `Element`

Returns the element's *configure phase*, an integer that specifies when it should be configured and initialized relative to other elements.

An element with a low configure phase will be configured before an element with a high configure phase. Elements with the same configure phase might be configured in any order relative to one another.

The following basic configure phase constants are defined in `<click/element.hh>`:

`CONFIGURE_PHASE_FIRST`
: Configure before most other elements. Only used by *AddressInfo* in the Click distribution.

`CONFIGURE_PHASE_INFO`
: Configure early. Appropriate for most information elements.

`CONFIGURE_PHASE_DEFAULT`
: Default configuration phase. Appropriate for most elements.

`CONFIGURE_PHASE_LAST`
: Configure after most other elements. No elements in the Click distribution use this configure phase.

`configure_phase` may also return a number based on these constants. For example, all *FromLinux* elements should be initialized before any *ToDevice* elements. The *FromLinux* element therefore contains the following definitions:

```
enum { CONFIGURE_PHASE_FROMLINUX = CONFIGURE_PHASE_DEFAULT,
       CONFIGURE_PHASE_TODEVICE = CONFIGURE_PHASE_FROMLINUX + 1 };
```

`FromLinux::configure_phase` returns `CONFIGURE_PHASE_FROMLINUX`, and `ToDevice::configure_phase` returns `FromLinux::CONFIGURE_PHASE_TODEVICE`.

The default implementation returns `CONFIGURE_PHASE_DEFAULT`.

Click uses all elements' configure phases to construct a single element configuration order. It then configures elements in this order and, if there were no errors, initializes them in the same order. The `configure_phase` method is called once, relatively early—before `configure` and `initialize`.

An element's configure phase should depend only on its class. In particular, the body of a `configure_phase` method should consist of a single `return` statement returning some constant.

5.3 `configure`—Parsing Configure Strings

The `configure` method is passed the element's configuration string. This method is expected to parse the configuration string, report any errors, and initialize the element's internal state.

`virtual int` **configure** (`Vector<String> &`*conf*, `ErrorHandler *`*errh*) Method on `Element`

The *conf* argument is the element's configuration string, divided into configuration arguments by splitting at commas, and with comments and leading and trailing whitespace removed. If *conf* is empty, the element was not supplied with a configuration string (or its configuration string contained only comments and whitespace).

Any errors, warnings, or messages should be reported to *errh*. Messages should not specify the element name or type; this information will be supplied externally.

This method should return zero if configuration succeeds, or a negative number if it fails. Returning a negative number prevents the router from initializing.

The default `configure` method succeeds if and only if there are no configuration arguments.

The method may modify *conf* however it would like.

`configure` is called relatively early in the initialization process. For instance, `configure` may create or destroy input and output ports—the port validity check happens after `configure` completes. `configure` cannot determine whether a port is push or pull; neither can it query the router for information about its neighbors.

A `configure` method should not perform potentially harmful actions, such as truncating files or attaching to devices. These actions should be left for the `initialize` method, which is called later. This avoids harm if another element cannot be configured, or if the router is incorrectly connected, since in these cases `initialize` will never be called.

The *conf* argument is created by calling `cp_argvec` on the element's configuration string; see Section 7.3 [Config String Splitting], page 61.

5.4 `processing`—Push and Pull Processing

Elements use the `processing` method to specify whether their ports are push, pull, or agnostic. This method returns a *processing code*—an ASCII string that, properly interpreted, specifies the processing type for each port.

`virtual const char *` **processing** () `const` Method on `Element`

Returns the element's processing code as a null-terminated C string.

Processing codes look like this:

'*inputspec*`/`*outputspec*'

Each of *inputspec* and *outputspec* is a sequence of '`h`', '`l`', and '`a`' characters, containing at least one character. '`h`' indicates a push port, '`l`' a pull port, and '`a`' an agnostic port. The first character in each sequence represents the first port (port 0), and so forth. For example, "a/ah" says that the element's first input and first output ports are both agnostic, but the second output port is push.

Inputspec and *outputspec* need not have the correct numbers of characters. The last character in each specification is duplicated as many times as necessary, and any extra characters are ignored. Thus, the processing codes "aaaaaaaa/haaaaaaa" and "a/ha" behave identically.

The Element class provides mnemonic names for five common processing codes:

AGNOSTIC "a/a" (agnostic ports).

PUSH "h/h" (push ports).

PULL "l/l" (pull ports).

PUSH_TO_PULL
"h/l" (push input ports, pull output ports).

PULL_TO_PUSH
"l/h" (pull input ports, push output ports).

The default implementation for Element::processing returns AGNOSTIC.

The processing method should be declared on a single line in the element's class definition. It should return a C string constant or one of the five mnemonic names above. These guidelines make the element's processing code easy to automatically extract from the source code.

Here is a typical processing method.

```
class ARPQuerier : public Element { public: // ...
    const char *processing() const { return PUSH; }
}
```

5.5 flow_code—Packet Flow Within an Element

Connections determine how packets flow between elements in a router configuration. Packets flow *within* elements as well: packets arriving on an element's input port will then be emitted on zero or more output ports, possibly after some modification. The user supplies connection information explicitly, but information about packet flow within an element is provided by the element itself, via its flow_code method. This method returns a *flow code*: an ASCII string that, properly interpreted, defines how packets may travel within the element.

virtual const char * **flow_code** () const — Method on Element

Returns the element's flow code as a null-terminated C string.

Flow codes look like '*inputspec/outputspec*', where each of *inputspec* and *outputspec* is a sequence of *port codes*. The simplest port code is a single letter. Packets can travel from an input port to an output port if and only if the port codes match. (Case is significant.) For example, the flow code "x/x" says that packets can travel from the element's input port to its output port, while "x/y" says that packets never travel between ports.

A port code may also be a sequence of letters in brackets, such as '[abz]'. Two port codes match iff they have at least one letter in common, so '[abz]' matches 'a', but '[abz]' and '`[cde]`' do not match. The opening bracket may be followed by a caret '`^`'; this makes the port code match letters *not* mentioned between the brackets. Thus, the port code '`[^abc]`' is equivalent to '`[ABC...XYZdef...xyz]`'.

Finally, the '`#`' character is also a valid port code, and may be used within brackets. One '`#`' matches another '`#`' only when they represent the same port number—for example, when one '`#`' corresponds to input port 2 and the other to output port 2. '`#`' never matches any letter. Thus, for an element with exactly 2 inputs and 2 outputs, the flow code `"##/##"` behaves like `"xy/xy"`.

Inputspec and *outputspec* need not have the correct numbers of port codes. The last code in each specification is duplicated as many times as necessary, and any extra codes are ignored. Thus, the flow codes `"[x#][x#][x#][x#]/x######"` and `"[x#]/x#"` behave identically.

This table describes some simple flow codes.

`"x/x"` Packets may travel from any input port to any output port. Most elements use this flow code.

`"xy/x"` Packets arriving on input port 0 may travel to any output port, but those arriving on other input ports will not be emitted on any output. *ARPQuerier* uses this flow code.

`"x/y"` Packets never travel between input and output ports. *Idle* and *Error* use this flow code. So does *KernelTap*, since its input port and output port are decoupled (packets received on its input are sent to the kernel; packets received from the kernel are sent to its output).

`"#/#"` Packets arriving on input port *K* may travel only to output port *K*. *Suppressor* uses this flow code.

`"#/[^#]"` Packets arriving on input port *K* may travel to any output port except *K*. *EtherSwitch* uses this flow code.

The `Element` class provides a mnemonic name for a common flow code:

`COMPLETE_FLOW`
`"x/x"` (packets travel from any input to all outputs).

The default implementation for `Element::processing` returns `COMPLETE_FLOW`.

The `flow_code` method should be declared on a single line in the element's class definition. It should return a C string constant or `COMPLETE_FLOW`. These guidelines make the element's flow code easy to extract from the source code.

Here is a typical `flow_code` method.

```
class ARPQuerier : public Element { public: // ...
    const char *flow_code() const { return "xy/x"; }
}
```

Most elements do not declare a `flow_code` method, relying on the default implementation instead.

Click uses flow code information in its agnostic port assignment algorithm and its algorithms for finding upstream and downstream elements.

5.5.1 What Is a Flow Code?

Flow codes conveniently encode a more primitive concept, *flow matrices*. An element's flow matrix, M, is a Boolean matrix with `ninputs` rows and `noutputs` columns. The matrix element $m[i, j]$ is true if and only if packets can "travel" from input port i to output port j. Note that this is independent of the element's processing code; it holds for push, pull, and agnostic ports.

But what does it mean for a packet to "travel" from one port to another? This principle will help you pick the right flow code for an element: Consider how an element's flow matrix would affect a simple router.

Take an input port, i, and output port, j, on some element M. To decide whether $m[i, j]$ should be true, imagine this simple configuration (or a similar configuration):

```
... -> RED -> [i] M [j] -> Queue -> ...;
```

Now, should the *RED* element include the *Queue* element in its queue length calculation? The $m[i, j]$ element should be true if and only if the answer is yes.

For example, consider *ARPQuerier*'s second input port, which receives ARP responses. *ARPQuerier* may, on receiving an ARP response, emit a held-over IP packet on its first output. However, a *RED* element upstream of that second input port would probably not include the downstream *Queue* in its queue length configuration. After all, the ARP responses are effectively dropped; packets emitted onto the *Queue* originally came from *ARPQuerier*'s first input port. Therefore, $m[1, 0]$ is false, and *ARPQuerier*'s flow code specifies that packets arriving on the second input port are not emitted on any output port.

The *ARPResponder* element provides a contrasting example. It has one input port, which receives ARP queries, and one output port, which emits the corresponding ARP responses. A *RED* element upstream of *ARPResponder* would probably want to include a downstream *Queue*, since queries received by *ARPResponder* are effectively transmuted into emitted responses. Thus, $m[0, 0]$ is true, even though the packets *ARPResponder* emits are completely different from the packets it receives.

If you find this confusing, don't fret. It is perfectly fine to be conservative when assigning flow codes. About 96% of the Click distribution's elements use `COMPLETE_FLOW`.

5.6 `add_handlers`—Creating Handlers

After successfully configuring every element and assigning ports to push or pull, the driver calls every element's `add_handlers` method. This method should create any handlers provided by the element. See Section 6.4 [Handlers], page 52, for more information on handlers.

`virtual void` **add_handlers** `()` — Method on `Element`

> This method takes no arguments and returns no results. Its only side effect should be to create the element's class-specific handlers. Most `add_handlers` methods simply call `add_read_handler` and `add_write_handler` one or more times (see Section 6.4.2 [Adding Handlers], page 53), and possibly `add_task_handlers` (see Section 8.6 [Task Handlers], page 78).
>
> The default implementation does nothing.

The driver also calls every element's `add_default_handlers` method. This nonvirtual method adds the default handlers that every element shares. See Section 6.4.3 [Default Handlers], page 54, for more information.

`void` **add_default_handlers** (`bool` *config_writable*) Method on `Element`

Adds the default collection of handlers for the element. Most of these handlers are read-only. The 'config' handler may be read/write, but only if *config_writable* is true and the `can_live_reconfigure` method also returns true (see Section 6.5.1 [can_live_reconfigure], page 58).

5.7 `initialize`—Element Initialization

The `initialize` method is called just before the router is placed on line. It performs any final initialization, and provides the last chance to abort router installation with an error.

`virtual int` **initialize** (`ErrorHandler *`*errh*) Method on `Element`

Any errors, warnings, or messages should be reported to *errh*. Messages should not specify the element name; this information will be supplied externally.

This method should return zero if initialization succeeds, or a negative number if it fails. Returning a negative number prevents the router from initializing.

The default `initialize` method simply returns zero.

An element's `initialize` method may check whether its input or output ports are push or pull, or query the router for information about its neighbors. It may not create or destroy input or output ports.

If every element's `initialize` method succeeds, then the router is installed, and will remain installed until another router replaces it. Any errors that occur later than `initialize`—during a `push` or `pull` method, perhaps—will not take the router off line.

Common tasks performed in `initialize` methods include:

- Initializing `Tasks` (see Section 8.1 [Task Initialization], page 74).
- Allocating memory.
- Opening files.

5.8 `cleanup`—Cleaning Up State

The `cleanup` method should clean up any state allocated by the initialization process. For example, it should close any open files, free up memory, and unhook from network devices. Click calls `cleanup` when it determines that an element's state is no longer needed, either because a router configuration is about to be removed or because the router configuration failed to initialize properly. Click will call the `cleanup` method exactly once on every element it creates.

`virtual void` **cleanup** (`CleanupStage` *stage*) Method on `Element`

Clean up state related to this element. The method should never report errors to any source. The *stage* parameter is an enumeration indicating how far the element made it through the initialization process. Its values are, in increasing order:

`CLEANUP_NO_ROUTER`
: The element was never attached to a router.

`CLEANUP_CONFIGURE_FAILED`
: The element's `configure` method was called, but it failed.

`CLEANUP_CONFIGURED`
: The element's `configure` method was called and succeeded, but its `initialize` method was not called (because some other element's `configure` method failed).

`CLEANUP_INITIALIZE_FAILED`
: The element's `configure` and `initialize` methods were called. `configure` succeeded, but `initialize` failed.

`CLEANUP_INITIALIZED`
: The element's `configure` and `initialize` methods were called and succeeded, but its router was never installed (because some other element's `initialize` method failed).

`CLEANUP_ROUTER_INITIALIZED`
: The element's `configure` and `initialize` methods were called and succeeded, and the router of which it is a part was successfully installed.

`CLEANUP_MANUAL`
: Never used by Click. Intended for use when element code calls `cleanup` explicitly.

The default `cleanup` method does nothing.

`cleanup` serves some of the same functions as an element's destructor, and it's usually called immediately before the element is destroyed. However, `cleanup` may be called long before destruction. Elements that are part of an erroneous router are cleaned up, but kept around for debugging purposes until another router is installed.

5.9 `static_initialize` and `static_cleanup`

Use the `static_initialize` and `static_cleanup` methods to set up and remove any global state required by an element. Click calls each element's `static_initialize` method as the element code is loaded (before any elements are created), and calls its `static_cleanup` method as the element code is unloaded (generally when the driver exits). Each method is called exactly once. `static_initialize` is suitable for installing configuration string parsing routines, for example.

`void` **static_initialize** () — Static Method on `Element`

The default implementation does nothing.

`void` **static_cleanup** () — Static Method on `Element`

The default implementation does nothing.

Care is required when inheriting from elements that have `static_initialize` and/or `static_cleanup` methods. In particular, if an element `Derived` inherits from an element `Base` that has a `static_initialize` method, then `Derived` *must* provide its own `static_initialize` method, to ensure that `Base::static_initialize` doesn't get called twice. Generally `Derived::static_initialize` will do nothing. A similar statement holds for `static_cleanup`. See the `CheckIPHeader2` element source code for an example.

5.10 Initialization Phases

1. Determines how many ports are used on each element and calls their `notify_ninputs` and `notify_noutputs` methods.
2. Calls each element's `configure_phase` method, and uses the result to construct a configuration order.
3. Calls each element's `configure` method, passing in the relevant configuration string. The elements are configured according to the configuration order.
4. Checks that each connection connects a valid input port to a valid output port. This catches errors where a connection uses a port that does not exist.
5. Calls each element's `processing` method to determine whether its ports are push, pull, or agnostic.
6. For each element with agnostic ports, calls the corresponding `flow_code` method to determine constraints linking agnostic input ports to agnostic output ports.
7. Runs the constraint-satisfaction algorithm that determines whether each agnostic port is push or pull. This catches errors where a single agnostic port is used as both push and pull.
8. Checks that every connection is between two push ports or two pull ports.
9. Checks that push output ports and pull input ports are connected exactly once.
10. Checks that no input or output port goes unused.
11. Calls every element's `add_handlers` method.
12. Calls every element's `add_default_handlers` method. The 'config' handler may be read-write.
13. If there have been no errors up to this point, then calls each element's `initialize` method. The elements are initialized according to the configuration order. No `initialize` methods are called if there were any errors in any previous phase.
14. If there were no errors, then router initialization has succeeded, and the router is placed on line.
15. If there were errors, then router initialization has failed.
    a. Removes all handlers, then calls every element's `add_default_handlers` again to make information about the erroneous configuration available for debugging. The 'config' handler is always read-only.
    b. Calls the `uninitialize` method on each element whose `initialize` method returned successfully.

6 Element Runtime

6.1 Moving Packets

Two virtual functions on `Element`, `push` and `pull`, provide Click's means are the main methods for packet transfer.

6.1.1 `push`

`virtual void` **push** (`int` *port*, `Packet *`*p*) Method on `Element`

Called when an upstream element pushes the packet *p* onto this element's input port *port*. This element is expected to process the packet however it likes.

6.1.2 `pull`

`virtual Packet *` **pull** (`int` *port*) Method on `Element`

Called when a downstream element makes a pull request of this element's output port *port*. This element is expected to process the request however it likes and to return a packet.

6.1.3 Transferring Packets

`void` **push** (`Packet *`*p*) `const` Method on `Element::Port`

`Packet *` **pull** () `const` Method on `Element::Port`

6.1.4 `simple_action`

`Packet *` **simple_action** (`Packet *`*p*) Method on `Element`

6.2 Handling Packets

Every `Packet` object should be single-threaded through Click: the same `Packet` pointer should never be in use in two different places. In particular, an element should not use a `Packet` after passing it downstream to the rest of the configuration (by calling `output().push`, for example).

This, for example, is the wrong way to write a *Tee* with two outputs.

```
void
BadTee::push(int, Packet *p)
{
    output(0).push(p);
    output(1).push(p);
}
```

The same packet pointer, `p`, has been pushed to two different outputs. This is always illegal; the rest of the configuration may have modified or even freed the packet before returning control to `BadTee`. The correct definition uses the `clone` method:

```
void
GoodTee::push(int, Packet *p)
{
    output(0).push(p->clone());
    output(1).push(p);
}
```

Every push or pull method must account for every packet it receives by freeing it, emitting it on some output, or perhaps explicitly storing it for later. This push method, for example, contains a memory leak:

```
void
Leaky::push(int, Packet *p)
{
    const click_ip *iph = p->ip_header();
    // ... more processing ...
    _counter++;
    return;  // XXX Oops!
    // Must push the packet on, store it, or kill it before returning.
}
```

6.3 Running Tasks

6.4 Handlers

Handlers are access points through which users can interact with elements in a running Click router, or with the router as a whole. *Read* and *write handlers* behave like files in a file system, while *LLRPCs* provide a remote procedure call interface.

6.4.1 Read and Write Handler Overview

Read and write handlers appear to the user like files in a file system, or alternatively, like a limited RPC mechanism that uses ASCII strings for data transfer. To the element programmer, a read handler is simply a function that takes an element and returns a String; a write handler is a function that takes an element and a String and returns an error code.

`String` **(*ReadHandler)** (`Element *`*element*`, void *`*thunk*) Function Type

Read handler functions have this type. When the user accesses a read handler on an element, Click calls some `ReadHandler` function and passes the element as an argument. The *thunk* argument contains callback data specified when the handler was added (see Section 6.4.2 [Adding Handlers], page 53). The function's String return value is passed back to the user.

`int` **(*WriteHandler)** (`const String &`*data*`, Element *`*element*`, void *`*thunk*`, ErrorHandler *`*errh*) Function Type

Write handler functions have this type. When the user accesses some element write handler by passing in a string, Click calls some `WriteHandler` function and passes the data and the relevant element as arguments. The *thunk* argument contains callback data specified when the handler was added (see Section 6.4.2 [Adding Handlers], page 53). The return value is an error code: zero when there are no errors, and the negative of some `errno` value when there is an error. More detailed information about any errors should be reported to the *errh* argument.

Each handler has an ASCII *name*. Handler names must be unique within each element; for example, there can be at most one '`x`' read handler in a given element. A given name can be shared by a read handler and a write handler, however. Such a handler pair is colloquially called a "read/write handler", although its two components need not have anything to do with one another.

There is currently no way to pass data to a read handler or return data from a write handler. Use LLRPCs if you need a more RPC-like read-write interface.

Note that read and write handler functions are regular functions, not virtual functions. Often, therefore, handler functions are defined as private static member functions in the relevant element class.

Read and write handlers need not use ASCII-formatted data. Most existing handlers do format their data in ASCII, however, and use `cp_uncomment` to ignore leading and trailing whitespace and comments (see Section 7.2 [Quoting and Unquoting], page 60). You may want to do the same for consistency's sake.

Be careful when writing handlers that modify element state, or read state that packet processing can modify. On an SMP machine, a handler may be called on one processor while packets are passing through the router on another processor. Furthermore, multiple read handlers and safe LLRPCs (see Section 6.4.5 [LLRPC Overview], page 57) may be active simultaneously on different processors. Write handlers are serialized with respect to other handlers and LLRPCs (but not packet processing). That is, no other handler or LLRPC will proceed while a write handler is active.

6.4.2 Adding Handlers

Use `Element`'s `add_read_handler` and `add_write_handler` methods to add handlers for an element. You will generally call these methods only from within your element's `add_handlers` method (see Section 5.6 [add_handlers], page 47), although nothing prevents you from adding handlers at any time.

`void` **add_read_handler** (`const String &`*name*`,` `ReadHandler` *func*`, void *`*thunk*) — Method on `Element`

Adds a read handler named *name* for this element. When the handler is accessed, *func* will be called with `this` and *thunk* as parameters.

`void` **add_write_handler** (`const String &`*name*`,` `WriteHandler` *func*`, void *`*thunk*) — Method on `Element`

Adds a write handler named *name* for this element. When the handler is accessed, *func* will be called with the relevant data, `this`, *thunk*, and an `ErrorHandler` as parameters.

To create a read/write handler, call `add_read_handler` and `add_write_handler` and supply the same handler name.

These methods simply forward their requests to static `add_read_handler` and `add_write_handler` methods on the `Router` class. Call those methods directly to add handlers to other elements, or to add global handlers.

`void` **add_read_handler** (`const Element *`*element*, `const String &`*name*, `ReadHandler` *func*, `void *`*thunk*) — Static Method on `Router`

`void` **add_write_handler** (`const Element *`*element*, `const String &`*name*, `WriteHandler` *func*, `void *`*thunk*) — Static Method on `Router`

Adds a read or write handler for *element*, or a global read or write handler if *element* is null. The handler is named *name*.

The `change_handler_flags` method lets you change a handler's flags word (see Section 6.4.4.1 [Handler Objects], page 55).

`void` **change_handler_flags** (`Element *`*element*, `const String &`*name*, `uint32_t` *clear_flags*, `uint32_t` *set_flags*) — Static Method on `Router`

Changes the flags for *element*'s *name* handler, or the global *name* handler if *element* is null. The flags are changed by first clearing the bits set in *clear_flags*, then setting the bits set in *set_flags*. This method fails and returns −1 when the specified handler does not exist; otherwise, it returns 0.

6.4.3 Default Read and Write Handlers

Every element automatically provides five handlers, 'class', 'name', 'config', 'ports', and 'handlers'. There is no need to add these handlers yourself. The default handlers behave as follows:

'class' — Returns the element's class name, as returned by `class_name()`, followed by a newline. Example result: `"ARPQuerier\n"`.

'name' — Returns the element's name, as returned by `id()`, followed by a newline. Example result: `"arpq_0\n"`.

'config' — Returns the element's configuration string. If the configuration string does not end in newline, the hander appends a newline itself. Example result: `"18.26.7.1, 00:00:C0:4F:71:EF\n"`.

If `can_live_reconfigure` returns true, 'config' is also a write handler, and writing to it reconfigures the element. See Section 6.5 [Live Reconfiguration], page 58.

'ports' — Returns a multi-line string describing the element's ports and what they are connected to. The string has the form

M `input[s]`
. . . *M* input port descriptions, one per line . . .
N `output[s]`
. . . *N* output port descriptions, one per line . . .

Each port description lists the port's processing type, a dash, and then a comma-separated list of all the ports to which this port is connected. The processing type is either '`push`' or '`pull`'; formerly agnostic ports are indicated by a trailing tilde ('`push~`' or '`pull~`'). Example result:

```
1 input
push~   -       Strip@2 [0]
2 outputs
push~   -       [0] GetIPAddress@4
push    -       [0] Print@7
```

If Click was compiled with statistics collection enabled, the dash on each line is replaced by a packet count.

'handlers' Returns a string listing the element's visible handlers, one per line. Each line contains the handler name, a tab, and then either '`r`', '`w`', or '`rw`', depending on whether the handler is read-only, write-only, or read/write. Example result for an *InfiniteSource* element, which has many handlers:

```
scheduled       r
tickets r
reset   w
count   r
active  rw
burstsize       rw
limit   rw
data    rw
handlers        r
ports   r
config  rw
name    r
class   r
```

6.4.4 Accessing Handlers Internally

Element handlers are stored in the relevant `Router` as objects of type `Router::Handler`. (This design allows handler objects to be shared between elements when possible.) Handlers are often referred to by index; indexes between 0 and `Router::FIRST_GLOBAL_HANDLER` − `1` refer to element handlers, while indexes above `Router::FIRST_GLOBAL_HANDLER` refer to global handlers. Indexes less than 0 are used for error returns, such as nonexistent handlers. `Router` methods translate between handler indexes and `Router::Handler` objects, and find handlers or handler indexes given handler names.

6.4.4.1 The Router::Handler Type

The `Router::Handler` type allows you to check a handler's properties and call the handler. All of its methods are `const`; you must go through `Router` to change a handler's properties. `Router::Handler` objects do not contain element references, since they are shared among elements. That means you can't easily find the element (if any) to which a particular `Router::Handler` is attached.

`const String &` **name** () `const` — Method on `Router::Handler`

Returns the handler's name.

`uint32_t` **flags** () `const` — Method on `Router::Handler`

Returns the handler's flags as an integer. The lower bits of the flags word are reserved for the system, and four bits are reserved for drivers, but the upper bits (at least 16) are left uninterpreted, and may be used by elements. The first user flag bit is called `Router::Handler::USER_FLAG_0`; its position in the word equals `Router::Handler::USER_FLAG_SHIFT`. To change a handler's flags, use the `Router::change_handler_flags` method (see [Changing Handler Flags], page 54).

`bool` **readable** () `const` — Method on `Router::Handler`

Returns true iff this handler is readable.

`bool` **read_visible** () `const` — Method on `Router::Handler`

Returns true iff this handler is readable, and that read handler should be externally visible. Drivers and the ControlSocket element use `read_visible` rather than `readable` when deciding whether to tell the user that a read handler exists. Inter-element communication within the router, however, may use `readable` rather than `read_visible`.

`bool` **writable** () `const` — Method on `Router::Handler`
`bool` **write_visible** () `const` — Method on `Router::Handler`

The analogous methods for write handlers.

`bool` **visible** () `const` — Method on `Router::Handler`

Equivalent to `read_visible() || write_visible()`.

`String` **unparse_name** (`Element *`*element*) `const` — Method on `Router::Handler`

Returns the handler's name, including its attached element's name if *element* is non-null. For example, calling `unparse_name` on element '`e`''s '`foo`' handler would return '`e.foo`', while calling it on a global '`bar`' handler would return '`bar`'.

`String` **unparse_name** (`Element *`*element*, `const String &`*name*) — Static Method on `Router::Handler`

Returns a string representing *element*'s hypothetical *name* handler, or the global *name* handler if *element* is null.

`String` **call_read** (`Element *`*element*) `const` — Method on `Router::Handler`

Calls this read handler on *element* and returns the result. Do not use this method unless you know the handler is `readable()`.

`int` **call_write** (`const String &`*data*, `Element *`*element*, `ErrorHandler *`*errh*) `const` — Method on `Router::Handler`

Calls this write handler on *element*, passing it *data* and *errh*, and returns the result. Do not use this method unless you know the handler is `writable()`.

6.4.4.2 Handlers By Name or Index

These `Router` methods locate handlers by name, returning either a pointer to a handler object or a handler index. The methods are static to allow access to global handlers outside the context of a running router.

`const Router::Handler *` **handler** (`const Element *`*element*`, const String &`*name*) — Static Method on `Router`

Returns a pointer to the handler object for *element*'s handler named *name*, or null if no such handler exists. *Element* may be null, in which case the method looks for a global handler named *name*.

> **Caution**: Handler pointers returned by `Router::handler` and similar methods should be treated as transient, since they may become invalid when new handlers are added.

`int` **hindex** (`const Element *`*element*`, const String &`*name*) — Static Method on `Router`

Like `Router::handler`, above, but returns an integer handler index for the named handler, or a negative number if no such handler exists. All valid handler indexes are nonnegative.

`const Router::Handler *` **handler** (`const Router *`*router*`, int` *hindex*) — Static Method on `Router`

Returns *router*'s handler object corresponding to *hindex*, or a null pointer if *hindex* is invalid with respect to *router*. There are three possibilities: (1) *hindex* coresponds to a valid global handler, which is returned. In this case, *router* need not be valid. (2) *hindex* corresponds to a valid local handler in class *router*, which is returned. (3) Otherwise, a null pointer is returned.

`const Router::Handler *` **handler** (`const Element *`*element*`, int` *hindex*) — Static Method on `Router`

Convenience function equivalent to `handler(`*element*`->router(), `*hindex*`)`. Note that *hindex* need not refer to one of *element*'s handlers.

`const Router::Handler *` **handler** (`int` *hindex*) `const` — Method on `Router`

Convenience function equivalent to `handler(this, `*hindex*`)`.

Finally, the `element_hindexes` static method returns all the handler indices that apply to a given element.

`void` **element_hindexes** (`const Element *`*element*`, Vector<int> &`*results*) — Static Method on `Router`

Appends to *results* all the handler indexes for *element*'s handlers, or all global handlers if *element* is null.

6.4.5 LLRPC Overview

6.5 Live Reconfiguration

6.5.1 can_live_reconfigure

7 Configuration Strings

7.1 Structure

Configuration strings consist of a list of comma-separated *arguments*. For example, this configuration string has three arguments, ‘a’, ‘b’, and ‘c’:

```
a,     b            , c
```

Leading and trailing whitespace is trimmed from each argument.

Configuration strings can contain two kinds of *comments* and three kinds of *quoted strings*. Comments let you document a configuration string; they behave like spaces. With quoted strings, you can protect special characters like whitespace, commas, and comment-starting sequences from interpretation.

‘//’ comments
: Begins with two adjacent slashes, ‘//’, and continues up to and including the next end-of-line (‘\n’, ‘\r’, or ‘\r\n’). Comment starters (‘//’ and ‘/*’) and the quote sequences (‘'’, ‘"’, and ‘\<’) have no special meaning inside ‘//’ comments.

‘/* ... */’ comments
: Begins with slash-star, ‘/*’, and continues up to and including the next star-slash, ‘*/’. Comment starters (‘/*’ and ‘//’) and the quote sequences (‘'’, ‘"’, and ‘\<’) have no special meaning inside ‘/*’ comments.

Single-quoted strings ‘' ... '’
: Begins with a single-quote character ‘'’ and continues up to the next single quote. Comments, double quotes, and backslashes have no special meaning inside single quotes. There is no way to include a single quote in a single-quoted string.

Double-quoted strings ‘" ... "’
: Begins with a double-quote character ‘"’ and continues up to the next unescaped double quote. Backslash ‘\’ acts as an escape character inside double quotes, as in C. Click’s escape sequences are described below. Comments and single quotes have no special meaning inside double quotes. ‘\<’ retains its usual meaning, however.

Hex strings ‘\< ... >’
: The ‘\<’ sequence begins a string of hexadecimal digits terminated by ‘>’. Each pair of digits expands to the corresponding character value. For example, ‘\<48454c4c4F>’ expands to ‘HELLO’. Whitespace and comments (either ‘//’ or ‘/*’ style) may be arbitrarily interleaved with the hex digits; any ‘>’ characters inside comments are ignored. Characters other than whitespace, hex digits, comments, and ‘>’ should not appear inside a hex string.

  Hex strings may be placed within double-quoted strings.

Escape Sequences

Most of Click's escape sequences are borrowed from C, and behave the same way. The '`\< . . . >`' escape sequence is new, however.

'\⟨END-OF-LINE⟩'
: A backslash followed by an end-of-line sequence—'`\n`', '`\r`', or '`\r\n`'—is removed from the string. This string

```
"a\
b"
```

is equivalent to `"ab"`.

'`\a`', '`\b`', '`\t`', '`\n`', '`\v`', '`\f`', '`\r`'
: These escape sequences produce the characters with decimal ASCII values 7, 8, 9, 10, 11, 12, and 13, respectively.

'`\\`', '`\"`', '`\'`', '`\$`'
: These escape sequences expand to a literal backslash, double quote, single quote, and dollar sign, respectively.

'\⟨1 TO 3 OCTAL DIGITS⟩'
: A backslash followed by 1 to 3 octal digits ('`0`' . . . '`7`') expands to the character with that octal value. For example, '`\046`' expands to '`&`'.

'`\x`⟨HEX DIGITS⟩'
: '`\x`' followed by an arbitrary number of hexadecimal digits expands to the single character whose value equals the lower 8 bits of that number. Thus, '`\x45`' and '`\x94839E89DB00ACF45`' both expand to '`E`'.

'`\<` ⟨HEX DIGITS⟩ `>`'
: '`\<`' introduces a hex string, as described above.

Any other escape sequence '\⟨CHAR⟩' is an error. Currently, such sequences expand to '⟨CHAR⟩', but their semantics may eventually change.

7.2 Quoting and Unquoting

These functions interpret quote sequences and comments in configuration strings. `cp_uncomment` removes comments and leading and trailing whitespace, but does not expand quote sequences. `cp_unquote` both removes comments and expands quote sequences. Finally, `cp_quote` protects special characters, such as whitespace and commas, within double quotes.

`String` **cp_uncomment** (`const String &`*str*) Function
: Replaces any comments in *str* by single spaces, then removes any leading and trailing whitespace and returns the result.

`String` **cp_unquote** (`const String &`*str*) Function
: Replaces any comments in *str* by single spaces, then removes any leading and trailing whitespace. Finally, replaces every quoted string by its expansion and returns the result.

String **cp_quote** (const String &*str*, bool *allow_newlines* = false) Function

Returns a quoted version of *str*. Any whitespace, commas, comments, quote sequences, and non-ASCII characters in *str* are protected within double quotes. If *allow_newlines* is true, then the result may contain newline characters (within double quotes); otherwise, any newline characters in *str* are replaced by '\n' sequences. The returned result is never empty (unless Click has run out of memory). If *str* is the empty string, cp_quote will return '""' (a string containing two double quotes).

For example:

```
cp_uncomment("  /* blah */ "quote"/*xx*/\<2 c>") ⇒ ""quote" \<2 c>"
cp_unquote("  /* blah */ "quote"/*xx*/\<2 c>") ⇒ "quote ,"
cp_quote("quote ,") ⇒ ""quote ,""
```

7.3 Splitting and Combining

void **cp_argvec** (const String &*str*, Vector<String> &*conf*) Function

Splits *str* into arguments by breaking it at every comma not part of a quote or comment. Comments and leading and trailing whitespace are removed from each argument, as by cp_uncomment, and the results are pushed, in order, onto the vector *conf*. If *str* contains only whitespace and comments, nothing is pushed onto *conf*.

void **cp_spacevec** (const String &*str*, Vector<String> &*conf*) Function

Splits *str* into arguments by breaking it at every sequence of whitespace characters and/or comments. Leading and trailing whitespace is removed from each argument, as by cp_uncomment, and the results are pushed, in order, onto the vector *conf*. If *str* contains only whitespace and comments, nothing is pushed onto *conf*.

For example:

```
cp_argvec("  x/* ,,, */ab" c",   \<de>   , ',' ", vec)
     ⇒ 3 arguments:  "x ab" c""  "\<de>"  "','"
cp_argvec("  /* blah, blah, blah, blah */  ", vec)
     ⇒ 0 arguments
cp_argvec("  /* blah, blah, blah, blah */,  ", vec)
     ⇒ 2 empty arguments:  ""  ""
cp_spacevec("  x/* ,,, */yz" w"  \<d e>  """, vec)
     ⇒ 4 arguments:  "x"  "yz" w""  "\<d e>"  """"
cp_spacevec("  /* blah, blah, blah, blah */  ", vec)
     ⇒ 0 arguments
cp_spacevec("  /* blah, blah, blah, blah */,  ", vec)
     ⇒ 1 argument:  ","
```

Since the const Vector<String> &*conf* arguments passed to elements' configure methods (see Section 5.3 [configure], page 44) have been processed by cp_argvec, there is no need to process them with cp_uncomment.

The cp_unargvec and cp_unspacevec functions take a vector of arguments and combine them into a single string. These functions do not protect their arguments by quoting; use cp_quote explicitly when necessary (see Section 7.2 [Quoting and Unquoting], page 60). If the arguments are properly quoted, then calling `cp_argvec(cp_unargvec(`*conf*`),` *conf2*`)` or `cp_spacevec(cp_unspacevec(`*conf*`),` *conf2*`)` will produce a new vector of arguments equal to the original.

`String` **cp_unargvec** (`const Vector<String> &`*conf*) Function

Returns a string consisting of the elements of *conf* separated by '`, `'.

`String` **cp_unspacevec** (`const Vector<String> &`*conf*) Function

Returns a string consisting of the elements of *conf* separated by '` `'.

For example:

`cp_unargvec(`[“`x ab" c"`”, “`\<de>`”, “`','`”]`)`
⇒ “`x ab" c", \<de>, ','`”
`cp_unargvec(`[“`whatever`”]`)`
⇒ “`whatever`”
`cp_unargvec(`[ ]`)`
⇒ “”
`cp_unargvec(`[“`,`”, “`,`”]`)`
(Probably a mistake: caller should have quoted the arguments!)
⇒ “`,, ,`”
`cp_unspacevec(`[“`xy" z"`”, “`\<de>`”, “`','`”]`)`
⇒ “`xy" z" \<de> ','`”

7.4 Parsing Functions

Click's *parsing functions* parse strings into various kinds of data, such as integers, fixed-point real numbers, and IP addresses. Parsing functions follow some consistent conventions:

- Their first argument, `const String &`*str*, contains the string to be parsed.
- At least one additional argument points to a location where any parsed result should be stored. These *result* arguments have pointer type.
- Their return type is `bool`.
- They return true if and only if parsing succeeds.
- The values pointed to by the *result* arguments are modified only if parsing succeeds.
- Most parsing functions expect to parse the entire supplied string. Any extraneous characters, such as trailing whitespace, cause parsing to fail.
- Parsing functions never report errors to any source; they simply return false when parsing fails.

7.4.1 Strings and Words

These functions parse strings from their input strings. The resulting strings may be arbitrary (`cp_string`) or constrained (`cp_word`, `cp_keyword`). As noted above (see Section 7.4 [Parsing Functions], page 62), the functions have `bool` return type; they return true if parsing was successful.

`bool` **cp_string** (`const String &`*str*`, String *`*result*`,` `String *`*rest* `= 0`) — Parsing Function

Parses a string from the beginning of *str* and stores the result in `*`*result*. The parsed string may contain single and double quotes and hex strings ("`\< ... >`"), which are processed as by `cp_unquote` (see Section 7.2 [Quoting and Unquoting], page 60).

If the *rest* argument is null and *str* contains any unquoted whitespace, then parsing will fail. If *rest* is not null, then parsing stops at the first unquoted whitespace character, and any leftover portion of *str* is stored in `*`*rest*.

For example:

> `cp_string(`"`"a b c d"`"`,` *result*`)` ⇒ true
>   `*`*result* = "`a b c d`"
> `cp_string(`""`,` *result*`)` ⇒ false
> `cp_string(`"`""`"`,` *result*`)` ⇒ true
>   `*`*result* = ""
> `cp_string(`"` "a b c d"`"`,` *result*`)` ⇒ false
>   (*str* began with an unquoted space)
> `cp_string(`"`"a b c d" e`"`,` *result*`)` ⇒ false
>   (*str* contained an unquoted space)
> `cp_string(`"`"a b c d" e`"`,` *result*`,` *rest*`)` ⇒ true
>   `*`*result* = "`a b c d`", `*`*rest* = "` e`"

`bool` **cp_word** (`const String &`*str*`, String *`*result*`,` `String *`*rest* `= 0`) — Parsing Function

Parses a word from the beginning of *str* and stores the result in `*`*result*. A *word* is a string that does not contain whitespace, control characters, non-ASCII characters (with values 127 or higher), or special characters ('`'`', '`"`', '`\`', or '`,`'). *str* may contain single and double quotes and hex strings ("`\< ... >`"), which are processed as by `cp_unquote` (see Section 7.2 [Quoting and Unquoting], page 60). The unquoted result must not contain quote marks, whitespace, or other special characters, however. Returns true if and only if *str* contained a valid word.

If the *rest* argument is null and *str* contains any unquoted whitespace, then parsing will fail. If *rest* is not null, then parsing stops at the first unquoted whitespace character, and any leftover portion of *str* is stored in `*`*rest*.

For example:

> `cp_word(`"`word`"`,` *result*`)` ⇒ true
>   `*`*result* = "`word`"
> `cp_word(`"`"wor"\<64>`"`,` *result*`)` ⇒ true
>   `*`*result* = "`word`"
> `cp_word(`"`"wor d"`"`,` *result*`)` ⇒ false
>   (processed string contained a space)

`bool` **cp_keyword** (`const String &`*str*`, String *`*result*`,` `String *`*rest* `= 0`) — Parsing Function

Parses a keyword from the beginning of *str* and stores the result in `*`*result*. A *keyword* is a string consisting of one or more letters, numbers, underscores ('`_`'), periods ('`.`'), and colons ('`:`'). Keywords may not contain quoted substrings—'`'`', '`"`', and '`\<`' are not allowed. Returns true if and only if *str* contained a valid keyword.

If the *rest* argument is null and *str* contains any unquoted whitespace, then parsing will fail. If *rest* is not null, then parsing stops at the first unquoted whitespace character, and any leftover portion of *str* is stored in `*`*rest*.

For example:

> `cp_keyword(`"`word`"`, ` *result*`)` ⇒ true
>
> `*`*result* = "`word`"
>
> `cp_keyword(`"`"wor"\`<64>"`, ` *result*`)` ⇒ false
>
> (quotes not allowed in keywords)

To summarize:

- `cp_string` and `cp_word` allow quoted substrings; `cp_keyword` does not.
- `cp_string` results may contain arbitrary characters; `cp_word` and `cp_keyword` restrict the characters allowed in their results.
- If `cp_keyword(`*str*`, `*result*`)` is true, then `cp_word(`*str*`, `*result*`)` is true.
- If `cp_word(`*str*`, `*result*`)` is true, then `cp_string(`*str*`, `*result*`)` is true.

7.4.2 Booleans

The `cp_bool` function parses a string into a Boolean value.

`bool` **cp_bool** `(const String &`*str*`, bool *`*result*`)` Parsing Function

Parses *str* into a Boolean value and stores the result in `*`*result*. Allowable Boolean strings are as follows:

'`0`', '`false`', '`no`'
: `*`*result* becomes false.

'`1`', '`true`', '`yes`'
: `*`*result* becomes true.

The words must be all lower case.

7.4.3 Integers

`cp_integer` and `cp_unsigned` parse strings into `int` and `unsigned int` values, respectively.

Each function comes in two variants, one with a *base* parameter and one without. If *base* is 0 or unspecified, then the function examines the string to determine the relevant base. Strings beginning with '`0x`' or '`0X`' (after the optional sign) use base 16; other strings beginning with '`0`' use base 8; and all other strings use base 10. Nonzero *base*s must be at least 2 and no more than 36.

The functions accept the same strings as C's `strtol` function, except that `strtol` will accept leading whitespace and trailing characters that are not part of the parsed integer. The string should contain, in order:

- An optional '`+`' or '`-`' sign. (The `cp_unsigned` functions do not accept a minus sign.)
- An optional '`0x`' or '`0X`', if the *base* argument is 0 or 16.
- One or more alphanumeric digits. The maximum allowed digit is specified by *base*.

If a string contains a valid number too large (or small) to represent, the parsing function sets `cp_errno` to `CPE_OVERFLOW`, stores the largest (or smallest) allowable number in *result*, and returns true. If a function succeeds without overflow, `cp_errno` is set to `CPE_OK`.

`bool` **cp_integer** (`const String &`*str*`, int *`*result*) — Parsing Function
`bool` **cp_integer** (`const String &`*str*`, int` *base*`, int *`*result*) — Parsing Function

Parses *str* into a signed integer in base *base* and stores the result in `*`*result*. Detects overflow on numbers greater than 2147483647 or less than −2147483648.

`bool` **cp_unsigned** (`const String &`*str*`, unsigned *`*result*) — Parsing Function
`bool` **cp_unsigned** (`const String &`*str*`, int` *base*`, unsigned *`*result*) — Parsing Function

Parses *str* into an unsigned integer in base *base* and stores the result in `*`*result*. Detects overflow on numbers greater than 4294967295.

For example:

cp_integer('-0x8000', *result*) ⇒ true
    `*`*result* = −32768, `cp_errno` = `CPE_OK`
cp_integer('-0x8000 ', *result*) ⇒ false
    (trailing whitespace not allowed)
cp_unsigned('3333333333333333', 4, *result*) ⇒ true
    `*`*result* = 4294967295, `cp_errno` = `CPE_OK`
cp_unsigned('33333333333333333', 4, *result*) ⇒ true
    `*`*result* = 4294967295, `cp_errno` = `CPE_OVERFLOW`

7.4.4 Real Numbers

Several functions parse real numbers into fixed-point integers. (Some drivers, such as the Linux kernel driver, can't use floating point arithmetic, so `double`s are not allowed.)

Each function takes an integer argument that determines how many digits of fraction the result should have. Since the result is a single fixed-point number, the more digits of fraction in the result, the fewer digits are available for the integer part.

You may request binary or decimal digits of fraction. The `real10` function variants use decimal digits, while the `real2` variants use binary digits: bits. For example, `cp_real10('1', 2,` *result*`)`, which parses the string '`1`' with 2 decimal digits of fraction, yields the number 100 (10^2). The similar call to a binary-digit function, `cp_real2('1', 2,` *result*`)`, yields 4 (2^2). Parsing '`0.5`' with the same functions yields 50 and 2, respectively.

A real number string should contain, in order:

- An optional '`+`' or '`-`' sign. (The `unsigned` variants do not accept minus signs.)
- An optional sequence of decimal digits representing the integer part.
- An optional fraction point '`.`'.
- An optional sequence of decimal digits representing the fraction part.
- An optional *exponent*—an '`E`' or '`e`' character followed by a signed decimal integer.

The string must contain at least one digit in either the integer part or the fraction part.

All the parsing functions round to the nearest relevant number. For example, `cp_real10('0.59', 1, ` *result*`)` stores 6 in *result*, since 0.59 rounded to one digit of fraction is 0.6.

If a string contains a real number too large in magnitude for the specified format, the parsing function will set `cp_errno` to `CPE_OVERFLOW`, store the largest representable number in *result*, and return true. For example, the largest number representable as an unsigned integer with 16 bits of fraction is 65535.99998, which has the bit pattern `0xFFFFFFFF`. Therefore, `cp_unsigned_real2('65536', 16, ` *result*`)` stores `0xFFFFFFFF` in *result* and sets `cp_errno` to `CPE_OVERFLOW`. If there was no overflow or other error, `cp_errno` is set to `CPE_OK`.

`bool` **cp_unsigned_real10** `(const String &`*str*`, int` *frac_digits*`, unsigned *`*result*`)` — Parsing Function

Parses *str* into an unsigned real number, and stores the result in *result* as an unsigned integer with *frac_digits* decimal digits of fraction.

`bool` **cp_real10** `(const String &`*str*`, int` *frac_digits*`, int *`*result*`)` — Parsing Function

Parses *str* into a unsigned real number, and stores the result in *result* as an integer with *frac_digits* decimal digits of fraction.

`bool` **cp_unsigned_real2** `(const String &`*str*`, int` *frac_bits*`, unsigned *`*result*`)` — Parsing Function

Parses *str* into an unsigned real number, and stores the result in *result* as an unsigned integer with *frac_bits* bits of fraction.

`bool` **cp_real2** `(const String &`*str*`, int` *frac_bits*`, int *`*result*`)` — Parsing Function

Parses *str* into a real number, and stores the result in *result* as an integer with *frac_bits* bits of fraction.

The fixed-point real parsing functions are built on a lower-level variant that returns the integer and fraction parts in two different `unsigned int`s.

`bool` **cp_unsigned_real10** `(const String &`*str*`, int` *frac_digits*`, unsigned *`*int_result*`, unsigned *`*frac_result*`)` — Parsing Function

Parses *str* into an unsigned real number and stores the result in *int_result* and *frac_result*. *int_result* holds the integral part of the resulting real, while *frac_result* holds its fractional part as a fixed-point number with *frac_digits* decimal digits of fraction. *frac_result* is always less than 10^*frac_digits*.

For example:

`cp_unsigned_real10('10.952', 3, ` *int_result*`, ` *frac_result*`)` ⇒ true
    **int_result* = 10, **frac_result* = 952
`cp_unsigned_real10('10.9526', 3, ` *int_result*`, ` *frac_result*`)` ⇒ true
    **int_result* = 10, **frac_result* = 953
    (note rounding)
`cp_unsigned_real10('10.9996', 3, ` *int_result*`, ` *frac_result*`)` ⇒ true
    **int_result* = 11, **frac_result* = 0

7.4.5 IP Addresses

The `cp_ip_address` functions parse strings into IP addresses. Related `cp_ip_prefix` and `cp_ip_address_set` functions parse strings into IP address/netmask pairs and sets of IP addresses, respectively.

Parsable IP addresses are simply dotted quads like '`18.26.4.44`'. IP prefixes may be specified using CIDR notation, such as '`18.26.4.44/16`'; as explicit address/netmask pairs, such as '`18.26.4.44/255.255.0.0`'; or, optionally, as bare IP addresses, such as '`18.26.4.44`' (which means '`18.26.4.44/255.255.255.255`').

Besides these conventional forms, the `cp_ip` functions understand user-defined shorthand names for IP addresses and prefixes. Shorthand names are router-specific; users define them with *AddressInfo* elements. Furthermore, a name's meaning is dependent on its context: an *AddressInfo* inside a compound element defines shorthand names local to that compound element. The `cp_ip` functions, then, take optional `Element *`*context* arguments to specify any router and compound-element context. If a `cp_ip` function's *context* argument is null, it will parse only the conventional IP address forms described above.

`bool` **cp_ip_address** `(const String &`*str*`, unsigned char *`*result*`, Element *`*context* `= 0)` — Parsing Function

`bool` **cp_ip_address** `(const String &`*str*`, IPAddress *`*result*`, Element *`*context* `= 0)` — Parsing Function

Parses *str* into an IP address and stores the result in `*`*result*. *context* supplies any element context.

`bool` **cp_ip_prefix** `(const String &`*str*`, unsigned char *`*result_addr*`, unsigned char *`*result_mask*`, Element *`*context* `= 0)` — Parsing Function

`bool` **cp_ip_prefix** `(const String &`*str*`, IPAddress *`*result_addr*`, IPAddress *`*result_mask*`, Element *`*context* `= 0)` — Parsing Function

Parses *str* into an IP address/netmask pair and stores the resulting address in `*`*result_addr*, and the resulting netmask in `*`*result_mask*. The resulting address is not pre-masked by the resulting mask. For example, `cp_ip_prefix('18.26.4.44/16',` *result_addr*`,` *result_mask*`)` stores 18.26.4.44 in *result_addr*, not 18.26.0.0. Bare addresses, such as '`18.26.4.44`', are never allowed.

`bool` **cp_ip_prefix** `(const String &`*str*`, unsigned char *`*result_addr*`, unsigned char *`*result_mask*`, bool` *allow_bare_addr*`, Element *`*context* `= 0)` — Parsing Function

`bool` **cp_ip_prefix** `(const String &`*str*`, IPAddress *`*result_addr*`, IPAddress *`*result_mask*`, bool` *allow_bare_addr*`, Element *`*context* `= 0)` — Parsing Function

Parses *str* into an IP address/netmask pair and stores the resulting address in `*`*result_addr* and netmask in `*`*result_mask*. Bare addresses, such as '`18.26.4.44`', are allowed if and only if *allow_bare_addr* is true. The netmask corresponding to a bare address is 255.255.255.255.

Finally, the `cp_ip_address_list` function parses a whitespace-separated list of IP addresses into to an `IPAddressList` object.

`bool` **cp_ip_address_list** (`const String` &*str*, `IPAddressList` **result*, `Element` **context* = 0) — Parsing Function

Parses *str* into a list of IP addresses and stores the result in **result*. *str* must be a whitespace-separated list of IP addresses, which can take any of the forms accepted by `cp_ip_address`.

7.4.6 IPv6 Addresses

The `cp_ip6_address` functions parse strings into IPv6 addresses. Related `cp_ip6_prefix` functions parse strings into IPv6 address/netmask pairs.

Parsable IPv6 addresses and prefixes take any of the forms described in RFC 2373, *IP Version 6 Addressing Architecture.* A nonabbreviated address consists of eight colon-separated 16-bit hexadecimal numbers, as in '`1080:0:0:0:8:800:200C:417A`'. Strings of zero bits may be abbreviated with two colons, as in '`1080::8:800:200C:417A`', and an address may end in an embedded IPv4 address, as in '`::13.1.68.3`' and '`::FFFF:129.144.52.38`'. IPv6 prefixes are written in '*address*/*prefixlen*' form, like '`12AB:0:0:CD30::/60`'. Click also supports '*address*/*netmask*' syntax, where *netmask* is an IPv6 address. *netmask* must correspond to some contiguous prefix, however: '`12AB:0:0:CD30::/60`' and '`12AB:0:0:CD30::/FFFF:FFFF:FFFF:FFF0::`' are equivalent, but '`12AB:0:0:CD30::/FFFF::1`' is illegal.

Analogously to the `cp_ip` functions (see Section 7.4.5 [Parsing IP Addresses], page 67), the `cp_ip6` functions understand *AddressInfo*'s shorthand names for IPv6 addresses, and take optional `Element` **context* arguments to specify any router and compound-element context.

`bool` **cp_ip6_address** (`const String` &*str*, `unsigned char` **result*, `Element` **context* = 0) — Parsing Function

`bool` **cp_ip6_address** (`const String` &*str*, `IP6Address` **result*, `Element` **context* = 0) — Parsing Function

Parses *str* into an IPv6 address and stores the result in **result*. *context* supplies any element context.

`bool` **cp_ip6_prefix** (`const String` &*str*, `unsigned char` **result_addr*, `int` **result_prefix_len*, `bool` *allow_bare_addr*, `Element` **context* = 0) — Parsing Function

`bool` **cp_ip6_prefix** (`const String` &*str*, `IP6Address` **result_addr*, `int` **result_prefix_len*, `bool` *allow_bare_addr*, `Element` **context* = 0) — Parsing Function

Parse *str* into an IPv6 address/prefix length pair and stores the resulting address in **result_addr*, and the resulting prefix length in **result_prefix_len*. Bare addresses, such as '`1080::8:800:200C:417A`', are allowed if and only if *allow_bare_addr* is true. The prefix length corresponding to a bare address is 128.

bool **cp_ip6_prefix** (const String &*str*, unsigned char **result_addr*, unsigned char **result_mask*, bool *allow_bare_addr*, Element **context* = 0) — Parsing Function

bool **cp_ip6_prefix** (const String &*str*, IP6Address **result_addr*, IP6Address **result_mask*, bool *allow_bare_addr*, Element **context* = 0) — Parsing Function

Parse *str* into an IPv6 address/prefix length pair and stores the resulting address in **result_addr*, and the netmask corresponding to the resulting prefix length in **result_mask*. Bare addresses are allowed if and only if *allow_bare_addr* is true.

7.4.7 Ethernet Addresses

The `cp_ethernet_address` functions parse strings into Ethernet addresses. A parsable Ethernet address consists of six colon-separated 8-bit hexadecimal numbers, as in '`0:2:B3:06:36:EE`'.

Analogously to the `cp_ip` functions (see Section 7.4.5 [Parsing IP Addresses], page 67), the `cp_ethernet_address` functions understand *AddressInfo*'s shorthand names for Ethernet addresses, and take optional `Element` **context* arguments to specify any router and compound-element context.

bool **cp_ethernet_address** (const String &*str*, unsigned char **result*, Element **context* = 0) — Parsing Function

bool **cp_ethernet_address** (const String &*str*, EtherAddress **result*, Element **context* = 0) — Parsing Function

Parses *str* into an Ethernet address and stores the result in **result*. *context* supplies any element context.

7.4.8 Elements

`cp_element` parses an element name into a pointer to an element in some router configuration. It differs from other parsing functions in two important ways. First, it returns its result, or a null pointer on error; parsing functions store their results in some pointer. Second, it reports errors to the supplied `ErrorHandler`.

The `cp_element` function follows lexical scoping rules when called from a compound element: it will check for components of that compound element first. For instance, say you've called `cp_element` on the string '`e`'. Normally, this would check the router for an element named, simply, '`e`'. However, if called within a compound element '`x`', `cp_element` will first check for an element named '`x/e`' before looking for the global '`e`' element. The function uses its *context* argument, an element pointer, to determine both the relevant router object and any compound element context.

More explicitly, the `cp_element` function uses the following procedure to search for an element named *str*:

1. Set *prefix* to *context*`->id()`.
2. Remove the final component of *prefix*.
3. Search for an element named '*prefixstr*' in *context*`->router()`. If one is found, return it.

4. Otherwise, no element was found. If *prefix* is already empty, parsing fails; report an error to *errh* and return a null pointer. Otherwise, return to step 2.

`Element *` **cp_element** (`const String &`*str*`, Element *`*context*`, ErrorHandler *`*errh*) — Function

Returns a element named *str* in *context*'s router configuration. *str* is first processed as by `cp_unquote`. *context* determines both the relevant router configuration and any compound element context. Returns a null pointer if no element is found; if *errh* is nonnull and no element is found, additionally reports an error to *errh*.

A variant function does not perform a lexically scoped search, so its *str* argument must contain a fully-qualified element name.

`Element *` **cp_element** (`const String &`*str*`, Router *`*router*`, ErrorHandler *`*errh*) — Function

Returns a element named *str* in *router*. *str* is first processed as by `cp_unquote`. Returns a null pointer if no element is found; if *errh* is nonnull and no element is found, additionally reports an error to *errh*.

7.4.9 Handlers

The `cp_handler` functions parse a handler specification, such as '`e.config`', into the relevant pair of element and handler ID. Unlike most other parsing functions, it can report errors to an `ErrorHandler`, if one is supplied.

Most handler specifications consists of an element name and a handler name separated by a period: '*element*`.`*handler*'. The simplest `cp_handler` function parses such a specification into an element pointer, corresponding to *element*, and the handler name, *handler*. Like `cp_element` (see Section 7.4.8 [Parsing Elements], page 69), `cp_handler` uses a lexically-scoped search to find the element corresponding to a given name.

Click also supports a few global handlers, such as '`config`'. `cp_handler` will also parse global handler names, returning null for the element pointer.

`bool` **cp_handler** (`const String &`*str*`, Element *`*context*`, Element **`*result_element*`, String *`*result_hname*`, ErrorHandler *`*errh*) — Parsing Function

Parses *str* into a handler specification, storing the resulting element (if any) in `*`*result_element* and handler name in `*`*result_hname*. *str* is first processed as by `cp_unquote`. *context* determines both the relevant router configuration and any compound element context. Returns true if and only if *str* contained a valid handler specification whose element part named an actual element. Note that this function will not check whether `*`*result_element* actually has a handler named `*`*result_hname*—or, for global handlers, whether the global handler `*`*result_hname* actually exists.

The other `cp_handler` variants ensure that the input string names an actual handler. These variants are useless until handlers are added to the router configuration. Therefore, they should be called in elements' `initialize` methods, not their `configure` methods, since handlers are not added until `initialize` time (see Section 5.10 [Initialization Phases], page 50).

bool **cp_handler** (const String &*str*, Element **context*, Element ***result_element*, int **result_hid*, ErrorHandler **errh*) — Parsing Function

Parses *str* into a handler specification, storing the resulting element in **result_element* and handler ID in **result_hid*. This function just calls the simpler `cp_handler`, above, then checks that the resulting element has the named handler.

bool **cp_handler** (const String &*str*, Element **context*, bool *need_read*, bool *need_write*, Element ***result_element*, int **result_hid*, ErrorHandler **errh*) — Parsing Function

Similar, but additionally checks for read and/or write handlers. If *need_read* is true, then *str* must name a valid read handler; if *need_write* is true, then *str* must name a valid write handler. Returns false if these checks aren't met.

7.4.10 Miscellaneous

The `cp_seconds_as` and `cp_timeval` functions parse strings into time.

bool **cp_seconds_as** (int *p*, const String &*str*, int **result*) — Parsing Function

Parses *str* as a possibly fractional length of time in seconds. The returned *result* is measured in (seconds $* 10^{-p}$); for example, if p is 3, then *result* is measured in milliseconds, and `cp_seconds_as(3, "8", `*result*`)` stores 8000 in **result*.

Str may contain an optional time unit suffix. Valid units are '`h`' or '`hr`' for hours, '`m`'/'`min`' for minutes, '`s`'/'`sec`' for seconds, '`ms`'/'`msec`' for milliseconds, '`us`'/'`usec`' for microseconds, and '`ns`'/'`nsec`' for nanoseconds. For example, `cp_seconds_as(0, "1h", `*result*`)` stores 3600 in **result*.

Negative values are not allowed.

bool **cp_seconds_as_milli** (const String &*str*, int **result*) — Parsing Function
bool **cp_seconds_as_micro** (const String &*str*, int **result*) — Parsing Function

Same as `cp_seconds_as(3, `*s*`, `*result*`)` and `cp_seconds_as(6, `*s*`, `*result*`)`, respectively.

bool **cp_timeval** (const String &*str*, struct timeval **result*) — Parsing Function

Parses *str* as a `struct timeval` representing some number of seconds and microseconds. Textually, this looks like a nonnegative real number with 6 decimal digits of fraction. Stores the integer part of the result in *result*`->tv_sec` and the fraction part in *result*`->tv_usec`. Basically equivalent to `cp_unsigned_real10(`*str*`, 6, 0, &`*result*`->tv_sec, &`*result*`->tv_usec)`.

7.5 Parsing Argument Lists

7.5.1 Concepts

7.5.2 Global Initialization

The `cp_va` functions maintain some private global state—for example, a list of the data types they understand. You must explicitly initialize this state with `cp_va_static_initialize` before calling any other `cp_va` function. You can free this state, if you'd like, with `cp_va_static_cleanup`.

`void` **cp_va_static_initialize** () — Function

Call this function exactly once, at the beginning of the program, before calling any other `cp_va` functions.

`void` **cp_va_static_cleanup** () — Function

Call this function exactly once, at the end of the program. It is an error to call any `cp_va` function after calling `cp_va_static_cleanup`.

Constant	**Storage Arguments**
`cpArgument`	`String` **result*
`cpString`	`String` **result*
`cpWord`	`String` **result*
`cpKeyword`	`String` **result*
`cpByte`	`unsigned char` **result*
`cpShort`	`short` **result*
`cpUnsignedShort`	`unsigned short` **result*
`cpInteger`	`int` **result*
`cpUnsigned`	`unsigned` **result*
`cpReal2`	`int` *frac_bits*, `int` **result*
`cpUnsignedReal2`	`int` *frac_bits*, `unsigned` **result*
`cpReal10`	`int` *frac_digits*, `int` **result*
`cpUnsignedReal10`	`int` *frac_digits*, `unsigned` **result*
`cpIPAddress`	`IPAddress` **result*
`cpIPPrefix`	`IPAddress` **result_address*, `IPAddress` **result_mask*
`cpIPAddressOrPrefix`	`IPAddress` **result_address*, `IPAddress` **result_mask*
`cpIPAddressList`	`IPAddressList` **result*
`cpEtherAddress`	`EtherAddress` **result*
`cpIP6Address`	`IP6Address` **result*
`cpIP6Prefix`	`IP6Address` **result_address*, `IP6Address` **result_mask*
`cpIP6AddressOrPrefix`	`IP6Address` **result_address*, `IP6Address` **result_mask*
`cpElement`	`Element` ***result*
`cpHandlerName`	`Element` ***result_element*, `String` **result_hname*
`cpHandler`	`Element` ***result_element*, `int` **result_hid*
`cpReadHandler`	`Element` ***result_element*, `int` **result_hid*
`cpWriteHandler`	`Element` ***result_element*, `int` **result_hid*
`cpBool`	`bool` **result*
`cpSeconds`	`int` **result*
`cpSecondsAsMilli`	`int` **result*
`cpSecondsAsMicro`	`int` **result*

cpTimeval	struct timeval *result

Constant	Corresponding Parsing Function
cpArgument	*result = arg
cpString	cp_string(arg, result)
cpWord	cp_string(arg, result)
cpKeyword	cp_keyword(arg, result)
cpByte	cp_unsigned(arg, &tmp), check range, store in result
cpShort	cp_integer(arg, &tmp), check range, store in result
cpUnsignedShort	cp_unsigned(arg, &tmp), check range, store in result
cpInteger	cp_integer(arg, result)
cpUnsigned	cp_unsigned(arg, result)
cpReal2	cp_real2(arg, frac_bits, result)
cpUnsignedReal2	cp_unsigned_real2(arg, frac_bits, result)
cpReal10	cp_real10(arg, frac_digits, result)
cpUnsignedReal10	cp_unsigned_real10(arg, frac_digits, result)
cpIPAddress	cp_ip_address(arg, result, context)
cpIPPrefix	cp_ip_prefix(arg, result_address, result_mask, false, context)
cpIPAddressOrPrefix	cp_ip_prefix(arg, result_address, result_mask, true, context)
cpIPAddressList	cp_ip_address_list(arg, result, context)
cpEtherAddress	cp_ether_address(arg, result, context)
cpIP6Address	cp_ip6_address(arg, result, context)
cpIP6Prefix	cp_ip6_prefix(arg, result_address, result_mask, false, context)
cpIP6AddressOrPrefix	cp_ip6_prefix(arg, result_address, result_mask, false, context)
cpElement	cp_element(arg, context, result)
cpHandlerName	cp_handler(arg, context, result_element, result_hname)
cpHandler	cp_handler(arg, context, result_element, result_hid)
cpReadHandler	cp_handler(arg, context, true, false, result_element, result_hid)
cpWriteHandler	cp_handler(arg, context, false, true, result_element, result_hid)
cpBool	cp_bool(arg, result)
cpSeconds	cp_seconds_as(0, arg, result)
cpSecondsAsMilli	cp_seconds_as(3, arg, result)
cpSecondsAsMicro	cp_seconds_as(6, arg, result)
cpTimeval	cp_timeval(arg, result)

8 Tasks

Click schedules a router's CPU or CPUs with one or more *task queues*. These queues are simply lists of *tasks*, which represent functions that would like access to the CPU. Tasks are generally associated with elements. When scheduled, most tasks call some element's `run_task` method.

Click tasks are represented by `Task` objects. An element that would like special access to a router's CPU should include and initialize a `Task` instance variable.

Tasks are generally called very frequently, up to tens of thousands of times per second. For infrequent events, it is far more efficient to use timers than to use tasks; see Chapter 9 [Timers], page 80.

Executing a task should not take a long time. The Click driver loop is not currently adaptive, so very long tasks can inappropriately delay timers and other periodic events. We may address this problem in a future release, but for now, keep tasks short.

The `Task` class is defined in the `<click/task.hh>` header file.

8.1 Task Initialization

Task initialization is a two-step process. First, when a `Task` object is constructed, you must supply information about the function that it should call when it is scheduled. Second, when the router is initialized, you must initialize the task by supplying it with the relevant router. (You must initialize the task even if it will not be scheduled right away.)

`Task` has two constructors. One of them asks the task to call an element's `run_task` method when scheduled; the other asks it to call an arbitrary function pointer.

Task (`Element *`*e*) — Constructor on `Task`

When this task is scheduled, call *e*`->run_task()`.

Task (`TaskHook` *hook*`, void *`*thunk*) — Constructor on `Task`

When this task is scheduled, call *hook*`(this, `*thunk*`)`. The *hook* argument is a function pointer with type `void (*)(Task *, void *)`.

The `Task::initialize` method places the task on a router-wide list of `Task`s, associates the task with a particular task queue, and, optionally, schedules it. Typically, an element's `initialize` method calls `Task::initialize` (see Section 5.7 [initialize], page 48).

`void` **initialize** (`Router *`*r*`, bool` *scheduled*) — Method on `Task`
`void` **initialize** (`Element *`*e*`, bool` *scheduled*) — Method on `Task`

Attaches the task to the router object *r* (or *e*`->router()`). Additionally sets the task's tickets to a default value, and schedules the task if *scheduled* is true.

Many elements call `ScheduleInfo::initialize_task` instead of calling `Task::initialize` directly. This method queries any *ScheduleInfo* elements in the configuration to determine the task's scheduling parameters, sets those parameters, and calls `Task::initialize` to schedule the task. The `ScheduleInfo::initialize_task` method is defined in the `<click/standard/scheduleinfo.hh>` header file.

void **initialize_task** (Element **e*, Task **task*, bool *schedule*, ErrorHandler **errh*) — Static Method on ScheduleInfo

Sets *task*'s scheduling parameters as specified by any *ScheduleInfo* elements in the router configuration. The element *e* is used to find the correct router, and provides the relevant name for parameter lookup—the user supplies parameters to *ScheduleInfo* by element name. If *schedule* is true, also schedules *task* on *e*->router()'s task queue. Reports any errors to *errh*.

void **initialize_task** (Element **e*, Task **task*, ErrorHandler **errh*) — Static Method on ScheduleInfo

A synonym for initialize_task(*e*, *task*, true, *errh*).

void **join_scheduler** (Element **e*, Task **task*, ErrorHandler **errh*) — Static Method on ScheduleInfo

A synonym for initialize_task(*e*, *task*, true, *errh*).

The initialize_task method is generally called like this:

```
int
SomeElement::initialize(ErrorHandler *errh)
{
    ScheduleInfo::initialize_task(this, &_task, errh);
}
```

Here, _task, a Task object, is one of SomeElement's instance variables.

8.2 Scheduling Tasks

The user may take a task off its task queue with the unschedule method, and place it back onto its task queue with the reschedule method. As tasks move to the head of the task queue, they are unscheduled and their callbacks are called. Within these callback functions, the user will typically call fast_reschedule, which is like reschedule without the locking overhead.

void **unschedule** () — Method on Task

Unschedules the task by removing it from its task queue. Does nothing if if the task is currently unscheduled, or if it was never initialized. When this function returns, the task will not be scheduled.

void **reschedule** () — Method on Task

Reschedules the task by placing it on its task queue. If the task is already scheduled, then this method does nothing.

All three functions lock the task queue before manipulating it. This avoids corruption when there are multiple processors executing simultaneously. If reschedule cannot immediately lock a task queue—perhaps because it is being used on another processor—then they register a task request, which will be executed in the near future. In contrast, the unschedule function will wait until it can lock the task queue.

Sometimes unscheduling a task is not enough: you don't want the task to run, even if someone else (an upstream queue, for example) were to reschedule it. The `strong_unschedule` method both unschedules the task and shifts the task to the quiescent thread, which never runs. Thus, a `strong_unschedule`d task will not run until someone calls `strong_reschedule`, which reschedules the task on its original preferred thread.

`void` **strong_unschedule** () Method on `Task`

Unschedules the task by removing it from its task queue and shifting it to the quiescent thread. Does nothing if if the task is currently unscheduled, or if it was never initialized. When this function returns, the task will not be scheduled.

`void` **strong_reschedule** () Method on `Task`

Reschedules the task by placing it on the task queue corresponding to its thread preference. The task will not be scheduled immediately upon return, but it will become scheduled soon—`strong_reschedule` uses a task request to avoid locking.

The `fast_reschedule` method avoids locking overhead in the common case that a task must be rescheduled from within its callback.

`void` **fast_reschedule** () Method on `Task`

Reschedules the task by placing it on its preferred task queue. This method avoids locking overhead, so it is faster than `reschedule`.

> **Caution**: You may call a `Task`'s `fast_reschedule` method only from within its callback function. For instance, if an element has a task, `_task`, that calls the element's `run_task` method when scheduled, and if `run_task` is called only by that task's callback, then that element's `run_task` method should call `_task.fast_reschedule()` instead of `_task.reschedule()`.

The `fast_unschedule` method is to `unschedule` as `fast_reschedule` is to `reschedule`. It is rarely used, since tasks are automatically unscheduled before they are run.

`void` **fast_unschedule** () Method on `Task`

Unschedules the task by removing it from its task queue. Does nothing if if the task is currently unscheduled, or if it was never initialized. This method avoids locking overhead, so it is faster than `unschedule`.

> **Caution**: You may call a `Task`'s `fast_unschedule` method only from within its callback function.

8.3 Tickets

Click tasks are scheduled using the flexible, lightweight stride scheduling algorithm.[1] This algorithm assigns each task a parameter called its *tickets*. A task with twice as many tickets as usual is scheduled twice as frequently.

`Task`s have methods for querying, setting, and adjusting their tickets.

---

[1] For more information, see MIT Laboratory for Computer Science Technical Memo MIT/LCS/TM-528, *Stride scheduling: deterministic proportional-share resource management*, by Carl A. Waldspurger and William E. Weihl, June 1995.

`int` **tickets** () `const` — Method on `Task`

Returns this task's tickets. This number will be at least 1 and no more than `Task::MAX_TICKETS`, which equals 32768.

`void` **set_tickets** (`int` *t*) — Method on `Task`

Sets this task's tickets to *t*. The *t* parameter should lie between 1 and `Task::MAX_TICKETS`, inclusive; numbers outside this range are constrained to the nearest valid value.

`void` **adj_tickets** (`int` *delta*) — Method on `Task`

Equivalent to `set_tickets(tickets() +` *delta*`)`.

8.4 Choosing a Thread

Each task belongs to some task queue, which generally corresponds to a thread of control. Single-threaded Click has one active thread, and therefore one task queue, but multithreaded Click can have an arbitrary number of threads. Either Click hasa special thread, the *quiescent thread*, numbered −1; tasks belonging to the quiescent thread never run, whether or not they are scheduled. Every task starts out belonging to the first thread, thread 0. The `change_thread` method moves a task to another thread.

`void` **change_thread** (`int` *thread_id*) — Method on `Task`

Move this task to thread *thread_id*, which should be a number between −1 and the relevant `Router`'s `nthreads()`.

The task is scheduled on the new task queue if and only if it was scheduled on the old task queue.

Like `reschedule`, `change_thread` must lock the task queue before manipulating it. (Unlike those methods, `change_thread` must lock two task queues, the old and the new.) If `change_thread` cannot lock a task queue, then it registers a task request that will be executed in the near future. This implies that a task may remain on the same thread, or become unscheduled, for some time after `change_thread` is called.

8.5 Task Status Methods

These methods let a user check various properties of a task—for instance, whether it is initialized or scheduled.

`bool` **initialized** () `const` — Method on `Task`

Returns true iff the task has been initialized—that is, if it is associated with some router.

`bool` **scheduled** () `const` — Method on `Task`

Returns true iff the task is currently scheduled on some task queue.

RouterThread * **scheduled_list** () const — Method on Task

Returns the task queue with which this task is associated. Even unscheduled tasks are associated with some task queue; this is the task queue on which the task will be placed if `reschedule` is called.

TaskHook **hook** () const — Method on Task

Returns the callback function that is called when the task is scheduled. If the task is associated with some element, this method returns a null pointer.

void * **thunk** () const — Method on Task

Returns the extra data passed to the callback function when the task is scheduled.

Element * **element** () const — Method on Task

If the task is associated with some element, this method returns that element. Otherwise, returns a null pointer.

8.6 Task Handlers

By convention, elements with tasks should provide handlers that access task properties. The `Element::add_task_handlers` method automatically adds these handlers for a given `Task` object.

void **add_task_handlers** (Task **task*, const String &*prefix* = String()) — Method on Element

Adds task handlers for *task* to this element. The string *prefix* is prepended to every handler name.

This method adds at least the following handlers:

'scheduled'
: Returns a Boolean value saying whether the task is currently scheduled on some task queue. Example result: `"true\n"`.

'tickets'
: Returns or sets the task's currently allocated tickets. This handler is only available if Click was compiled to support stride scheduling. Example result: `"1024\n"`.

'thread_preference'
: Returns the task's thread preference. This handler is only available on multi-threaded Click. Example result: `"2\n"`.

8.7 Task Cleanup

You generally don't need to worry about destroying `Task` objects: they are automatically unscheduled and removed when the `Router` is destroyed. This only works if the `Task` objects have the same lifetime as the `Router`, however. This includes the normal case, when `Tasks` are element instance variables. If you create and destroy `Task` objects as the router runs, however, you will need to call the following method before deleting the `Task`.

`void` **cleanup** () Method on `Task`

Cleans up the `Task` object.

9 Timers

Click *timers*, like Click tasks, represent callback functions that the driver calls when appropriate. Unlike tasks, however, you schedule timers to go off at a specified time. Timers are intended for more infrequent and/or slower tasks.

As with `Task`, most `Timer` objects are declared as instance variables of elements and scheduled when needed.

Timers may be scheduled with microsecond precision, but on current hardware, only millisecond precision is likely to be achievable.

The `Timer` class is defined in the `<click/timer.hh>` header file.

9.1 Timer Initialization

Timer initialization resembles task initialization. When the timer is constructed, you must supply it with information about its callback function. Later, after the router is initialized, you must initialize and, optionally, schedule it.

Timer (`Element *`*e*) — Constructor on `Timer`

When this timer goes off, call *e*`->run_timer()`.

Timer (`Task *`*t*) — Constructor on `Timer`

When this timer goes off, call *t*`->reschedule()`.

Timer (`TimerHook` *hook*`, void *`*thunk*) — Constructor on `Timer`

When this timer goes off, call *hook*`(this, `*thunk*`)`. The *hook* argument is a function pointer with type `void (*)(Timer *, void *)`.

`void` **initialize** (`Router *`*r*) — Method on `Timer`
`void` **initialize** (`Element *`*e*) — Method on `Timer`

Attaches the timer to the router object *r* (or *e*`->router()`).

Typically, an element's `initialize` method (see Section 5.7 [initialize], page 48) calls `Timer::initialize`, and possibly one of the `schedule` functions described below.

9.2 Scheduling Timers

A variety of methods schedule timers to go off at specified times. The basic method is `schedule_at`, which schedules the timer for a specified time. Subsidiary methods schedule the timer relative to the current time (the `schedule_after` methods), or relative to the last time the timer was scheduled to run (the `reschedule_after` methods). Finally, `unschedule` unschedules the timer.

All `schedule` and `reschedule` functions first unschedule the timer if it was already scheduled.

The `reschedule` methods are particularly useful for timers that should occur periodically. For example, this callback function will cause its timer to go off at 20-second intervals:

```
void timer_callback(Timer *t, void *) {
    t->reschedule_after_s(20);
}
```

void **schedule_at** (const struct timeval &*when*) Method on Timer
Schedule the timer to go off at *when*. You must have initialized the timer earlier.

void **schedule_now** () Method on Timer
Schedule the timer to go off as soon as possible.

void **schedule_after** (const struct timeval &*delay*) Method on Timer
Schedule the timer to go off *delay* after the current time.

void **schedule_after_s** (uint32_t *delay*) Method on Timer
Schedule the timer to go off *delay* seconds after the current time.

void **schedule_after_ms** (uint32_t *delay*) Method on Timer
Schedule the timer to go off *delay* milliseconds after the current time.

void **reschedule_after** (const struct timeval &*delay*) Method on Timer
Schedule the timer to go off *delay* after it was last scheduled to go off. If the timer was never previously scheduled, this method will schedule the timer for some arbitrary time.

void **reschedule_after_s** (uint32_t *delay*) Method on Timer
Schedule the timer to go off *delay* seconds after it was last scheduled to go off.

void **reschedule_after_ms** (uint32_t *delay*) Method on Timer
Schedule the timer to go off *delay* milliseconds after it was last scheduled to go off.

void **unschedule** () Method on Timer
Unschedules the timer, if it was scheduled.

9.3 Timer Status Methods

These methods return information about a timer, including when it is scheduled to expire.

bool **initialized** () const Method on Timer
Returns true iff the timer has been initialized with a call to `initialize()`. Uninitialized timers must not be scheduled.

bool **scheduled** () const Method on Timer
Returns true iff the timer is scheduled to expire some time in the future.

const struct timeval & **expiry** () const Method on Timer
Returns the time that the timer is set to expire. If the timer has never been scheduled, the value is garbage. If the timer was scheduled but is not scheduled currently, the value is most recently set expiry time.

9.4 Timer Cleanup

You don't need to worry about cleaning up `Timer` objects. They are automatically unscheduled and removed when the `Router` is destroyed, and deleting a `Timer` automatically removes it from any relevant lists. The following function is nevertheless provided for consistency with `Task`s, which do need to be cleaned up in certain circumstances (see Section 8.7 [Task Cleanup], page 78).

`void` **cleanup** () Method on `Timer`

Cleans up the `Timer` object.

10 Notification

11 Coding Standards

11.1 Upper and Lower Case in Names

Keep to the following consistent scheme for choosing between upper and lower case when naming variables, types, and functions.

Classes Use mixed case with an initial capital letter and no underscores: `LookupIPRoute`.

Methods Use all lower case with underscores separating words: `negation_is_simple`.

Constants Use all upper case with underscores separating words: `TYPE_ICMP_TYPE`.

Instance variables
Begin with an underscore, then use all lower case with underscores separating words: `_length`.

Regular variables
Use all lower case with underscores separating words: `i`, `the_handler`.

Class variables
These variables are declared as `static` in the class header. Name them like regular variables: `nelements_allocated`.

Functions Name them like methods: `quicksort_hook`.

Other types
This includes typedefs and enumerated types. Name them like classes: `CpVaParseCmd`, `ConfigurePhase`.

There are exceptions to these guidelines. In particular:

- Instance variables in C structs—that is, classes with few methods whose instance variables are mostly public—may be named like regular variables, without a preceding underscore. The same goes for the components of unions.
- Classes that act like simple types, such as `uatomic32_t`, take names similar to the types they replace (in this case `uint32_t`).

11.2 Common Name Patterns

- Many instance variables have associated *getter methods* that return their values, and/or *setter methods* that change their values. For an instance variable named `_x`, the getter method should be named `x()` and the setter method should be named `set_x()`.
- A variable or method which counts something is often named n*object*s—for instance, `_nwarnings`, `ninputs()`, `_npackets`.
- Use a bare '`0`' for a null pointer, except where some ambiguity might arise (for example, where an incorrect overloading might be selected).

Index

(Index is nonexistent)