

1. Introduction. This document describes *patchwerk*, a library for creating audio signal chains and audio graphs. Patchwerk provides an interface where DSP unit generators can be created and interconnected in a highly modular fashion. The goal of Patchwerk is to beat out the performance of Sporth, a stack based audio language well suited for modular synthesis and a previous project done by the author. There is also a desire to keep the focus on solely the signal chain interface, deferring DSP algorithms to other software such as Soundpipe or FAUST.

If reading this for the first time, it may be best to first read the [general overview] found in the next section. There is also a [table containing an overview of sections] that may also be useful. If using this document as a reference manual, there is a [generated table of contents] that can be found at the end of the document, and above that a hyperlinked index for looking up functions. Unfortunately, the document generated by CWEB does not handle text search all too well. Sorry about that.

2. An Overview of the System. It is perhaps best to conceptualize Patchwerk into three major components: [nodes], [cables], and [patches].

In graph theory, a *node* is the building block for which graphs are formed. In the context of digital signal processing, a node is *unit generator*, capable of reading and writing audio-rate signals. In digitally based modular environments, unit generators are strung together to build a sound. The interface for a node consists of a few user-defined callback functions, as well as data persistence.

Nodes communicate to one another through channels known as *cables*. A cable contains a pointer to some numerical value in memory. Cables, when they start out, point to an internal float value, providing a constant value. This initial pointer address can be overridden to point to other memory addresses. This allows nodes implicitly connected to one another.

A collection of interconnected nodes is known as a *patch*.

A node can have inputs that are fed by the outputs of other nodes. Nodes that feed into inputs are dependencies, and therefore their audio samples must be computed beforehand. Furthermore, nodes in a modular setting must be called exactly once.

Constructing an audio graph can be a very challenging problem. For instance, it takes a non-trivial bit of software engineering to produce a node list given a set of connections. Such complexities need not exist here! The solution the author has chosen is to restrict the way graphs can be constructed and connected. When a new node is created, it is appended to the end of a linked list of nodes and given an incremental unique ID based the position in the list (0, 1, 2, 3, etc.). A node can only connect to inputs of other nodes with higher IDs. This ensures that the node is guaranteed to be rendered beforehand.

3. Overview of Sections. The following section provides an overview of the subsections of Patchwerk.

Section Name	Description
[Header]	The single <i>patchwerk.h</i> header file generated.
[Node]	The <i>pw_node</i> interface for building nodes.
[Cable]	The <i>pw_cable</i> interface for building cables.
[Error]	Error handling.
[Pointer]	The <i>pw_pointer</i> and <i>pw_pointerlist</i> interfaces for handling pointers and pointer lists.
[Pool]	The <i>pw_buffer</i> and <i>pw_bufferpool</i> interfaces for managing buffers and buffer pools.
[Stack]	Describes the <i>pw_stack</i> interface, otherwise known as a buffer stack.
[Patch]	Describes the <i>pw_patch</i> interface, the top-level data structure for describing a single audio graph.
[Subpatch]	Describes the <i>pw_subpatch</i> interface, which introduces the concept of subpatches.
[Event Graph]	Event graphs in Patchwerk via the <i>pw_egraph</i> interface.

4. The first few lines of code are the header declarations, followed by the actual program, which boils down to a single top-level directive.

```
#include <stdlib.h>
#include <stdio.h>
#include "patchwerk.h"
⟨Top 8⟩
```

5. Header. This is the single header file for patchwerk.

```
<patchwerk.h 5> ≡  
#ifndef PATCHWERK_H  
#define PATCHWERK_H  
#include <stdio.h>  
#include <stdarg.h>     /* needed for print functionality */  
    <Header 6>  
#endif
```

This code is cited in section [56](#).

6. Type Definitions.

⟨Header 6⟩ ≡

```
#ifndef PWFLT
#define PWFLTfloat
#endif
    typedef struct pw_node pw_node;
    typedef struct pw_pointer pw_pointer;
    typedef void(*pw_function)(pw_node *);
    typedef void(*pw_nodefun)(pw_node *, void *); ⟨Type Declarations 56⟩
```

See also sections 7, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 65, 67, 70, 71, 73, 75, 77, 79, 81, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 105, 108, 111, 113, 128, 130, 132, 134, 136, 138, 141, 143, 145, 147, 162, 164, 167, 169, 171, 173, 175, 177, 178, 180, 183, 185, 187, 189, 191, 193, 195, 197, 198, 208, 210, 212, 214, 216, 223, 225, 227, 229, 231, 233, 235, 237, 239, 241, 243, 246, 247, 263, 265, 267, 269, 271, 273, 275, 277, 279, 281, 284, 286, 288, 291, 293, 295, 297, 299, 301, 303, 305, 312, 315, 318, 320, 322, 324, 326, 330, 334, 336, 339, 341, 343, 345, 347, 349, 351, 353, 355, 357, 363, 365, 367, 370, 372, 374, 376, 378, 380, 382, 384, 387, 391, 393, 396, 399, and 402.

This code is cited in section 309.

This code is used in section 5.

7. C structs.

$\langle \text{Header } 6 \rangle + \equiv$
 $\langle \text{Cable Data } 55 \rangle$

8. Node. A node is the atomic element in the audio signal graph, capable of reading and writing audio-rate signals.

$\langle \text{Top } 8 \rangle \equiv$
 $\langle \text{Node Top } 9 \rangle$

See also sections [54](#), [110](#), [112](#), [114](#), [149](#), [217](#), [249](#), [307](#), [328](#), [369](#), and [386](#).

This code is used in section [4](#).

9. Data. The C-Struct for the node is called **pw_node**. It is implemented to be an opaque data structure.

```

⟨Node Top 9⟩ ≡
    struct pw_node {
        ⟨Node Data 10⟩
    };

```

See also sections 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, and 53.

This code is used in section 8.

10. It is assumed that the node is created using the *pw_patch* interface, so the address of that top-level struct is saved.

```

⟨Node Data 10⟩ ≡
    pw_patch * patch;

```

See also sections 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20.

This code is used in section 9.

11. Each node is assigned a unique ID based on when they were created. nodes with lower id numbers can connect to and control nodes with higher id numbers. This design constraint greatly simplifies the creation of the graph and avoids having to do back-tracing.

```

⟨Node Data 10⟩ +=
    int id;

```

12. The *setup* callback function is called to allocate and initialize any memory.

```

⟨Node Data 10⟩ +=
    pw_function setup;

```

13. The *destroy* function is called at the end of the program. It is meant to free any memory allocated *setup*.

```

⟨Node Data 10⟩ +=
    pw_function destroy;

```

14. The *compute* callback function is called to compute a block of samples.

```

⟨Node Data 10⟩ +=
    pw_function compute;

```

15. DSP algorithms often will need some sort of internal memory. This memory is stored in generic pointer called *ud*.

```

⟨Node Data 10⟩ +=
    void *ud;

```

16. A node can have an arbitrary number of connector cables for input/output.

```

⟨Node Data 10⟩ +=
    pw_cable * cables;
    int ncables;

```

17. The block size is also stored. This is used to initialize audio-rate cables.

```

⟨Node Data 10⟩ +=
    int blksize;

```


18. A node can also have a type id specifier that can be optionally set by a unit generator.

$\langle \text{Node Data } 10 \rangle + \equiv$

int *type*;

19. In addition to a type id, a node can have a group id. This is useful when wanting to distinguish between different sets of nodes.

$\langle \text{Node Data } 10 \rangle + \equiv$

int *group*;

20. Nodes hold a pointer to the next sequential node so that they can be used in a linked list.

$\langle \text{Node Data } 10 \rangle + \equiv$

pw_node **next*;

21. Functions.

22. The function *pw_node_size* returns the size of the opaque data struct **pw_node**.

⟨Header 6⟩ +≡
size_t *pw_node_size*();

23. ⟨Node Top 9⟩ +≡
size_t *pw_node_size*()
{
 return **sizeof**(**pw_node**);
}

24. The function *pw_node_init* zeros out and initializes the **pw_node** structure. By default, the compute + setup callbacks are set to be an empty callback. The destroy callback by default will call upon *pw_node_cables_free*, as this is nearly always needed to be called here anyways.

⟨Header 6⟩ +≡
void *pw_node_init*(**pw_node** **node*, **int** *blksize*);

25. ⟨Node Top 9⟩ +≡
static void *empty*(**pw_node** **node*)
{ }
static void *free_cables*(**pw_node** **node*)
{
 pw_node_cables_free(*node*);
}
void *pw_node_init*(**pw_node** **node*, **int** *blksize*)
{
 node→*id* = -1;
 node→*setup* = *empty*;
 node→*compute* = *empty*;
 node→*destroy* = *free_cables*;
 node→*ncables* = 0;
 node→*blksize* = *blksize*;
 node→*type* = -1;
 node→*group* = -1;
 node→*next* = Λ ;
}

26. The function *pw_node_get_id* returns the node id.

⟨Header 6⟩ +≡
int *pw_node_get_id*(**pw_node** **node*);

27. ⟨Node Top 9⟩ +≡
int *pw_node_get_id*(**pw_node** **node*)
{
 return *node*→*id*;
}

28. The function *pw_node_set_id* sets the node id.

⟨Header 6⟩ +≡
void *pw_node_set_id*(**pw_node** **node*, **int** *id*);

29. $\langle \text{Node Top 9} \rangle + \equiv$
void *pw_node_set_id*(**pw_node** **node*, **int** *id*)
{
 node→*id* = *id*;
}

30. The following functions set the callbacks for the setup, compute, and destroy functions.

$\langle \text{Header 6} \rangle + \equiv$
void *pw_node_set_setup*(**pw_node** **node*, *pw_function* *fun*);
void *pw_node_set_compute*(**pw_node** **node*, *pw_function* *fun*);
void *pw_node_set_destroy*(**pw_node** **node*, *pw_function* *fun*);

31. $\langle \text{Node Top 9} \rangle + \equiv$
void *pw_node_set_setup*(**pw_node** **node*, *pw_function* *fun*)
{
 node→*setup* = *fun*;
}
void *pw_node_set_compute*(**pw_node** **node*, *pw_function* *fun*)
{
 node→*compute* = *fun*;
}
void *pw_node_set_destroy*(**pw_node** **node*, *pw_function* *fun*)
{
 node→*destroy* = *fun*;
}

32. The following functions are wrappers around the internal callback functions for setup, compute, and destroy.

$\langle \text{Header 6} \rangle + \equiv$
void *pw_node_setup*(**pw_node** **node*);
void *pw_node_compute*(**pw_node** **node*);
void *pw_node_destroy*(**pw_node** **node*);

33. $\langle \text{Node Top 9} \rangle + \equiv$
void *pw_node_setup*(**pw_node** **node*)
{
 node→*setup*(*node*);
}
void *pw_node_compute*(**pw_node** **node*)
{
 node→*compute*(*node*);
}
void *pw_node_destroy*(**pw_node** **node*)
{
 node→*destroy*(*node*);
}

34. User data for a given node can be set and retrieved with the wrapper functions *pw_node_set_data* and *pw_node_get_data*.

⟨Header 6⟩ +≡

```
void pw_node_set_data(pw_node *node, void *data);
void *pw_node_get_data(pw_node *node);
```

35. ⟨Node Top 9⟩ +≡

```
void pw_node_set_data(pw_node *node, void *data)
{
    node->ud = data;
}

void *pw_node_get_data(pw_node *node)
{
    return node->ud;
}
```

36. Nodes can contain an arbitrary number of patch cables, which must be preallocated with *pw_node_cables_alloc*■

⟨Header 6⟩ +≡

```
int pw_node_cables_alloc(pw_node *node, int ncables);
```

37. ⟨Node Top 9⟩ +≡

```
int pw_node_cables_alloc(pw_node *node, int ncables)
{
    int n;
    int rc;
    if (node->patch ==  $\Lambda$ ) return PW_NOT_OK;
    rc = pw_memory_alloc(node->patch, sizeof (pw_cable) * ncables, (void **) &node->cables);
    if (rc  $\neq$  PW_OK) return rc;
    node->ncables = ncables;
    for (n = 0; n < ncables; n++) {
        pw_cable_init(node, &node->cables[n]);
    }
    return PW_OK;
}
```

38. Nodes allocated with *pw_node_cables_alloc* must be freed with *pw_node_cables_free*.

⟨Header 6⟩ +≡

```
int pw_node_cables_free(pw_node *node);
```

39. $\langle \text{Node Top 9} \rangle + \equiv$

```

int pw_node_cables_free(pw_node *node)
{
    int n;
    if (node->patch  $\equiv \Lambda$ ) return PW_NOT_OK;
    if (node->ncables > 0) {
        for (n = 0; n < node->ncables; n++) {
            pw_cable_free(&node->cables[n]);
        }
        return pw_memory_free(node->patch, (void **) &node->cables);
    }
    return PW_NOT_OK;
}

```

40. A *pw_cable* can be retrieved from the opaque **pw_node** struct using *pw_node_get_cable*. On success, the function returns PW_OK, and PW_INVALID_CABLE for failure.

$\langle \text{Header 6} \rangle + \equiv$

```

int pw_node_get_cable(pw_node *node, int id, pw_cable **cable);

```

41. $\langle \text{Node Top 9} \rangle + \equiv$

```

int pw_node_get_cable(pw_node *node, int id, pw_cable **cable)
{
    if (id  $\geq$  node->ncables) {
        return PW_INVALID_CABLE;    /* fail on out-of-range cable ID */
    }
    *cable = &node->cables[id];
    return PW_OK;
}

```

42. The function *pw_node_blksize* gets the block size of the cables.

$\langle \text{Header 6} \rangle + \equiv$

```

int pw_node_blksize(pw_node *node);

```

43. $\langle \text{Node Top 9} \rangle + \equiv$

```

int pw_node_blksize(pw_node *node)
{
    return node->blksize;
}

```

44. The function *pw_node_set_block* sets a cable internal to the node to be an audio signal using the node's internal block size. It returns PW_OK on success, and PW_INVALID_CABLE on failure.

$\langle \text{Header 6} \rangle + \equiv$

```

int pw_node_set_block(pw_node *node, int id);

```

45. \langle Node Top 9 $\rangle + \equiv$

```
int pw_node_set_block(pw_node *node, int id)
{
    if (id > node->ncables) {
        return PW_INVALID_CABLE; /* fail on out-of-range cable ID */
    }
    return pw_cable_make_block(&node->cables[id], pw_patch_stack(node->patch), node->blksize);
}
```

46. The function *pw_node_get_ncables* returns the number of cables in a node.

\langle Header 6 $\rangle + \equiv$

```
int pw_node_get_ncables(pw_node *node);
```

47. \langle Node Top 9 $\rangle + \equiv$

```
int pw_node_get_ncables(pw_node *node)
{
    return node->ncables;
}
```

48. The functions *pw_node_set_type* and *pw_node_get_type* set and get the type ID of a particular node.

\langle Header 6 $\rangle + \equiv$

```
int pw_node_get_type(pw_node *node);
void pw_node_set_type(pw_node *node, int type);
```

49. \langle Node Top 9 $\rangle + \equiv$

```
int pw_node_get_type(pw_node *node)
{
    return node->type;
}

void pw_node_set_type(pw_node *node, int type)
{
    node->type = type;
}
```

50. A node is typically created using the *pw_patch* interface. This function explicitly stores the pointer location of *pw_patch* inside of a **pw_node**. When using the *pw_patch* abstraction, this function is called automatically.

\langle Header 6 $\rangle + \equiv$

```
void pw_node_set_patch(pw_node *node, pw_patch *patch);
int pw_node_get_patch(pw_node *node, pw_patch **patch);
```

51. \langle Node Top 9 $\rangle + \equiv$

```
void pw_node_set_patch(pw_node *node, pw_patch *patch)
{
    node->patch = patch;
}

int pw_node_get_patch(pw_node *node, pw_patch **patch)
{
    *patch = node->patch;
    return PW_OK;
}
```

52. The function *pw_node_get_next* and *pw_node_set_next* both get and set the next **pw_node** entry.

⟨Header 6⟩ +≡

```
pw_node *pw_node_get_next(pw_node *node);  
void pw_node_set_next(pw_node *node, pw_node *next);
```

53. ⟨Node Top 9⟩ +≡

```
pw_node *pw_node_get_next(pw_node *node)  
{  
    return node→next;  
}  
void pw_node_set_next(pw_node *node, pw_node *next)  
{  
    node→next = next;  
}
```

54. Cable. The cable is the primary means of sharing signals between modules. The cabling system takes advantage of C pointers to create implicit connections, which is very computationally efficient.

⟨ Top [8](#) ⟩ +≡
 ⟨ Cable Enums [61](#) ⟩
 ⟨ Cable Top [66](#) ⟩

55. Data. The c-structure for the cable is contained in a typedef called *pw_cable*.

```
< Cable Data 55 > ≡
    struct pw_cable {
        < Variables in Cable Data 57 >
    };

```

This code is used in section 7.

56. The **pw_cable** forward declaration is defined in the header file `<patchwerk.h 5>`.

```
< Type Declarations 56 > ≡
    typedef struct pw_cable pw_cable;

```

See also sections 117, 123, 151, 220, 251, 309, 329, 360, and 390.

This code is used in section 6.

57. A **pw_cable** is assumed to be created using a node. A copy of this node is stored inside of the data structure.

```
< Variables in Cable Data 57 > ≡
    pw_node *node;

```

See also sections 58, 59, 60, 62, and 63.

This code is used in section 55.

58. The output of a cable is stored in the pointer *val*. By default, it points to the internal variable *ival*. This is a constant value. This pointer can be reset to point to another memory address.

```
< Variables in Cable Data 57 > +=
    PWFLT *val;
    PWFLT ival;

```

59. It is a common paradigm in audio signal process to write DSP routines that process blocks rather than single samples. Every cable has an internal block of memory that must explicitly be allocated, with the size of the block stored as an integer. This block size should be a system-wide size set by the *pw_patch* struct. Any non-zero block size will be an indicator that memory for the block has been allocated, and must therefore be freed.

```
< Variables in Cable Data 57 > +=
    PWFLT *blk;
    int blksize;

```

60. The internal type of a cable keeps track of if a memory pointer is overridden or not. It is set to be one of the macros defined in `< Cable Enums 61 >`.

```
< Variables in Cable Data 57 > +=
    unsigned char type;

```

61. The following enum contains the various states a cable can be in. **CABLE_IVAL** is the default value, meaning the pointer is set to be the internal constant value. **CABLE_BLOCK** indicates that the cable is set to point to the internally allocated audio block. **CABLE_OVERRIDE** indicates that the cable has been overridden by another block. The assumption is that the other block is set to be a **CABLE_BLOCK**.

```
< Cable Enums 61 > ≡
    enum {
        CABLE_IVAL, CABLE_BLOCK
    };

```

This code is cited in section 60.

This code is used in section 54.

62. A **pw_cable** pointer is saved internally. By default, this is set to itself. Whenever a cable connection happens, the connecting cable gets stored here. This can be used for back tracing the graph.

⟨ Variables in Cable Data 57 ⟩ +≡
 pw_cable **pcable*;

63. Patchwerk uses an internal buffer interface known as a *pw_buffer* to store audio-rate blocks. This buffer is a low-level component to patchwerk's memory-efficient interfaces for handling audio buffers.

More information can be found at ⟨ A Single Buffer 152 ⟩ and ⟨ The Buffer Pool 155 ⟩.

⟨ Variables in Cable Data 57 ⟩ +≡
 pw_buffer **buf*;

64. Functions.

65. The function *pw_cable_init* initializes and zeros out variables. No memory allocation is done here. If a cable is not using a node, the *node* parameter can be set to be NULL.

⟨Header 6⟩ +≡

```
void pw_cable_init(pw_node *node, pw_cable *cable);
```

66. ⟨Cable Top 66⟩ ≡

```
void pw_cable_init(pw_node *node, pw_cable *cable)
{
    cable->ival = 0;
    cable->blksize = 0;
    cable->blk = Λ;
    cable->type = CABLE_IVAL;
    cable->val = &cable->ival;
    cable->node = node;
    cable->pcable = cable;
    cable->buf = Λ;
}
```

See also sections 68, 69, 72, 74, 76, 78, 80, 82, 83, 85, 87, 89, 91, 93, 95, 97, 99, 103, and 106.

This code is used in section 54.

67. The function *pw_cable_free* frees any allocated memory blocks. If the block size is nonzero, it assumes that memory has been previously allocated and frees it.

⟨Header 6⟩ +≡

```
void pw_cable_free(pw_cable *cable);
```

68. ⟨Cable Top 66⟩ +≡

```
void pw_cable_free(pw_cable *cable)
{
    if (cable->blksize ≠ 0) { /* free(cable->blk); */
    }
}
```

69. The function *pw_cable_set_block* sets an internal block of memory for block-based processing. The block must be allocated beforehand.

⟨Cable Top 66⟩ +≡

```
void pw_cable_set_block(pw_cable *cable, PWFLT *blk, int blksize)
{
    cable->blk = blk;
    cable->blksize = blksize;
    cable->val = cable->blk;
    cable->type = CABLE_BLOCK;
}
```

70. ⟨Header 6⟩ +≡

```
void pw_cable_set_block(pw_cable *cable, PWFLT *blk, int blksize);
```

71. The function *pw_cable_set_constant* sets the internal constant value.

⟨Header 6⟩ +≡

```
void pw_cable_set_constant(pw_cable *cable, PWFLT val);
```

72. $\langle \text{Cable Top 66} \rangle + \equiv$
void *pw_cable_set_constant*(**pw_cable** **cable*, **PWFLT** *val*)
{
 cable->val = &*cable->ival*;
 cable->ival = *val*;
 cable->type = **CABLE_IVAL**;
}

73. The function *pw_cable_set_value* sets the value of the cable to a variable. Unlike *pw_cable_set_constant*, this function does not override the pointer value. If a cable value has been overridden with another pointer, it will set that value.

$\langle \text{Header 6} \rangle + \equiv$
void *pw_cable_set_value*(**pw_cable** **c*, **PWFLT** *val*);

74. $\langle \text{Cable Top 66} \rangle + \equiv$
void *pw_cable_set_value*(**pw_cable** **c*, **PWFLT** *val*)
{
 **c->val* = *val*;
}

75. The function *pw_cable_get* retrieves a value from a cable at a given index position *pos*. If the cable type is a constant value **CABLE_IVAL**, the index position is discarded.

$\langle \text{Header 6} \rangle + \equiv$
PWFLT *pw_cable_get*(**pw_cable** **cable*, **int** *pos*);

76. $\langle \text{Cable Top 66} \rangle + \equiv$
PWFLT *pw_cable_get*(**pw_cable** **cable*, **int** *pos*)
{
 if (*cable->type* \equiv **CABLE_IVAL**) {
 return **cable->val*;
 }
 else {
 return *cable->val*[*pos*];
 }
}

77. $\langle \text{Header 6} \rangle + \equiv$
void *pw_cable_set*(**pw_cable** **cable*, **int** *pos*, **PWFLT** *val*);

78. $\langle \text{Cable Top 66} \rangle + \equiv$
void *pw_cable_set*(**pw_cable** **cable*, **int** *pos*, **PWFLT** *val*)
{
 if (*cable->type* \equiv **CABLE_IVAL**) {
 **cable->val* = *val*;
 }
 else {
 cable->val[*pos*] = *val*;
 }
}

79. The function *pw_cable_connect* connects the output of cable *c1* to cable *c2*. In order for a cable to connect to another cable, the node id must be less. If it isn't, it returns the error code `PW_CONNECTION_MISMATCH`. The pointer value and type entries of *c2* are set to be whatever the *c1* values are.

⟨Header 6⟩ +≡

```
int pw_cable_connect(pw_cable *c1, pw_cable *c2);
```

80. ⟨Cable Top 66⟩ +≡

```
int pw_cable_connect(pw_cable *c1, pw_cable *c2)
{
    int id1, id2;
    id1 = pw_node_get_id(c1->node);
    id2 = pw_node_get_id(c2->node);
    if (id1 > id2) return PW_CONNECTION_MISMATCH;
    pw_cable_connect_nocheck(c1, c2);
    return PW_OK;
}
```

81. The function *pw_cable_connect_nocheck* connects the output of cable *c1* to the *c2*. This cable does not check for cable ID numbers, so it is useful for floating cables not connected to a node.

⟨Header 6⟩ +≡

```
void pw_cable_connect_nocheck(pw_cable *c1, pw_cable *c2);
```

82. ⟨Cable Top 66⟩ +≡

```
void pw_cable_connect_nocheck(pw_cable *c1, pw_cable *c2)
{
    c2->type = c1->type;
    pw_cable_override(c1, c2);
}
```

83. The function *pw_cable_pop* indirectly pops a buffer from the buffer stack. This function was written specifically to work with the Runt interface. It assumed that buffer being popped is the buffer belonging to this specific cable, but some error checking is done. If the cable is not a block cable, the function will return `PW_NOT_OK`. Otherwise, it will return `PW_OK`.

⟨Cable Top 66⟩ +≡

```
int pw_cable_pop(pw_cable *cab)
{
    pw_stack *stack;
    pw_node *node;
    pw_buffer *tmp;
    if (!pw_cable_is_block(cab)) return PW_NOT_OK;
    node = cab->node;
    stack = pw_patch_stack(node->patch);
    pw_stack_pop(stack, &tmp);
    return PW_OK;
}
```

84. ⟨Header 6⟩ +≡

```
int pw_cable_pop(pw_cable *cab);
```

85.

```

⟨ Cable Top 66 ⟩ +=
  void pw_cable_push(pw_cable *cab)
  {
    pw_stack *stack;
    pw_node *node;
    pw_buffer *tmp;
    tmp = Λ;
    node = cab->node;
    stack = pw_patch_stack(node->patch);
    pw_stack_push(stack, &tmp);
  }

```

86. ⟨ Header 6 ⟩ +=

```

void pw_cable_push(pw_cable *cab);

```

87. The function *pw_cable_is_block* checks if the cable type is a block not. This a direct boolean comparison, so the function will return 1 on true, and 0 on false.

```

⟨ Cable Top 66 ⟩ +=
  int pw_cable_is_block(pw_cable *cab)
  {
    return cab->type == CABLE_BLOCK;
  }

```

88. ⟨ Header 6 ⟩ +=

```

int pw_cable_is_block(pw_cable *cab);

```

89. The function *pw_cable_is_constant* checks if the cable type is a constant. This a direct boolean comparison, so the function will return 1 on true, and 0 on false.

```

⟨ Cable Top 66 ⟩ +=
  int pw_cable_is_constant(pw_cable *cab)
  {
    return cab->type == CABLE_IVAL;
  }

```

90. ⟨ Header 6 ⟩ +=

```

int pw_cable_is_constant(pw_cable *cab);

```

91. The function *pw_cable_get_buffer* and *pw_cable_set_buffer* get and set the buffer value inside of a cable.

Note: This function does not do type checking and assumes the cable is block. If the cable is not a block, this buffer value may return null.

```

⟨ Cable Top 66 ⟩ +=
  pw_buffer *pw_cable_get_buffer(pw_cable *cab)
  {
    return cab->buf;
  }

  void pw_cable_set_buffer(pw_cable *cab, pw_buffer *buf)
  {
    cab->buf = buf;
  }

```

92. \langle Header 6 $\rangle + \equiv$

```
pw_buffer * pw_cable_get_buffer(pw_cable *cab);
void pw_cable_set_buffer(pw_cable *cab, pw_buffer * buf);
```

93. The function *pw_cable_make_block* will set up a block cable using a buffer stack. Block size must also be given here. This code has been refactored from *pw_node_set_block* in order to make it easier to build standalone cables for things like sends and throws. Block size will have to be specified here. This can be retrieved with *pw_node_blksize* or *pw_patch_blksize*.

On success, *pw_cable_make_block* will return `PW_OK`. Otherwise, it will return `PW_NOT_OK`.

\langle Cable Top 66 $\rangle + \equiv$

```
int pw_cable_make_block(pw_cable *cable, pw_stack * stack, int blksize)
{
    pw_buffer * buf;
    PWFLT * blk;
    buf =  $\Lambda$ ;
    if (pw_stack_push(stack, &buf)  $\neq$  PW_OK) {
        return PW_NOT_OK;
    }
    blk = pw_buffer_data(buf);
    pw_cable_set_block(cable, blk, blksize);
    pw_cable_set_buffer(cable, buf);
    return PW_OK;
}
```

94. \langle Header 6 $\rangle + \equiv$

```
int pw_cable_make_block(pw_cable *cable, pw_stack * stack, int blksize);
```

95. The function *pw_cable_clear* zeros out a block cable. If the cable is not a block cable, it will return `PW_NOT_OK`.

\langle Cable Top 66 $\rangle + \equiv$

```
int pw_cable_clear(pw_cable *cab)
{
    int i;
    if ( $\neg$ pw_cable_is_block(cab)) return PW_NOT_OK;
    for (i = 0; i < cab-blksize; i++) {
        cab-val[i] = 0;
    }
    return PW_OK;
}
```

96. \langle Header 6 $\rangle + \equiv$

```
int pw_cable_clear(pw_cable *cab);
```

97. The function *pw_cable_mix* will mix the signal contained in the block cable *in* to the block cable *sum* by some scaling amount *mix*.

⟨ Cable Top 66 ⟩ +=

```
int pw_cable_mix(pw_cable *in, pw_cable *sum, PWFLT mix)
{
    int i;
    if (¬pw_cable_is_block(in) ∨ ¬pw_cable_is_block(sum)) return PW_NOT_OK;
    for (i = 0; i < sum->blksize; i++) {
        sum->val[i] += in->val[i] * mix;
    }
    return PW_OK;
}
```

98. ⟨ Header 6 ⟩ +=

```
int pw_cable_mix(pw_cable *in, pw_cable *sum, PWFLT mix);
```

99. The function *pw_cable_blksize* returns the block size of the cable.

⟨ Cable Top 66 ⟩ +=

```
int pw_cable_blksize(pw_cable *cable)
{
    return cable->blksize;
}
```

100. ⟨ Header 6 ⟩ +=

```
int pw_cable_blksize(pw_cable *cable);
```

101. The function *pw_cable_override* overrides the value pointer of cable *c2* to be the value of cable *c1*, without setting the type flag. This can be thought of as a more low-level component of *pw_cable_connect_nocheck*. ■

102. ⟨ Header 6 ⟩ +=

```
void pw_cable_override(pw_cable *c1, pw_cable *c2);
```

103. Note below that block size value is copied alongside the other values. This is a deliberate choice. Dummy cables are cables initialized without knowing any global state information, and meant to be overridden by other cables. By default, cables are set to a block size of 0. Having the block size set to an incorrect value make *pw_cable_clear* work incorrectly, if at all.

⟨ Cable Top 66 ⟩ +=

```
void pw_cable_override(pw_cable *c1, pw_cable *c2)
{
    c2->val = c1->val;
    c2->pcable = c1;
    c2->blksize = c1->blksize;
}
```

104. The function *pw_cable_copy* copies the contents of one cable (*c1*) to another cable (*c2*).

105. ⟨ Header 6 ⟩ +=

```
void pw_cable_copy(pw_cable *c1, pw_cable *c2);
```


106. $\langle \text{Cable Top } 66 \rangle + \equiv$
void *pw_cable_copy*(**pw_cable** **c1*, **pw_cable** **c2*)
{
 int *blksize*;
 int *n*;
 PWFLT *tmp*;
 blksize = *c1*→*blksize*;
 for (*n* = 0; *n* < *blksize*; *n*++) {
 tmp = *pw_cable_get*(*c1*, *n*);
 pw_cable_set(*c2*, *n*, *tmp*);
 }
}

107. Error handling. When something goes wrong, it is helpful to know why. This implements a reasonably minimal error handling interface.

108. The error codes are supplied in an enum. This is to be included in the header.

```
<Header 6> +≡
enum {
    <Error Codes 109>
};
```

109. The error codes are as follows:

PW_OK indicates that things completed successfully without error.

PW_NOT_OK is a general error code.

PW_INVALID_CABLE indicates a nonexistent or invalid cable.

PW_INVALID_NODE indicates a nonexistent or invalid node.

PW_INVALID_BUFFER indicates a nonexistent or invalid buffer.

PW_ALREADY_ALLOCATED when a patch is attempted to be reallocated.

PW_CONNECTION_MISMATCH happens when a cable attempts to connect to a cable belonging to a lower ranking node.

PW_NOT_ENOUGH_NODES happens in *pw_patch_new_node* when the nodes in the patch are filled up and there are no nodes left.

PW_NULL_VALUE happens when a value attempted to be read is NULL.

```
<Error Codes 109> ≡
PW_OK,
PW_NOT_OK,
PW_INVALID_CABLE,
PW_INVALID_NODE,
PW_INVALID_BUFFER,
PW_ALREADY_ALLOCATED,
PW_CONNECTION_MISMATCH,
PW_INVALID_ENTRY,
PW_NOT_ENOUGH_NODES,
PW_NULL_VALUE,
PW_I_DONT_KNOW,
PW_POOL_FULL,
PW_STACK_OVERFLOW
```

This code is used in section 108.

110. Each error code corresponds to a entry inside of an array of constant string literals.

```
<Top 8> +≡
static const char *errmsg[] = {"Everything is okay!", "Oops! Something went wrong.",
    "Invalid cable.", "Invalid node.", "Invalid buffer.",
    "This thing has already been allocated.",
    "Cables can only connect to nodes with a higher id.",
    "Position out of range in ugen list.",
    "Patch is out of nodes. Consider allocating more nodes.",
    "Attempted to read a NULL value.", "I'm not actually sure what happened.",
    "Buffer Pool is Full.", "Stack Overflow."};
```

111. The function *pw_error* returns the string at a particular index position.

```
<Header 6> +≡
const char *pw_error(int rc);
```

112. \langle Top 8 $\rangle + \equiv$
const char *pw_error(**int** rc)
{
 if (rc \geq (**sizeof** (errmsg)/**sizeof** (*errmsg)) \vee rc < 0) {
 return errmsg[PW_I_DONT_KNOW];
 }
 else {
 return errmsg[rc];
 }
}

113. Error handling implements a macro which does an early return on bad return codes.

\langle Header 6 $\rangle + \equiv$
#define PW_ERROR_CHECK(rc) **if** (rc \neq PW_OK) **return** rc

114. Pointer. A pointer in Patchwerk is a wrapper around a generic **void *** pointer in C. It is used to store chunks of data, such as tables to be used in table-lookup oscillators. Memory allocation is handled by the pointer type. Pointers are linked together in a linked-list, and are called a pointer list.

⟨ Top 8 ⟩ +≡
⟨ Pointer Top 118 ⟩

115. Pointer Data and Type Declarations.

116. The pointer is implemented as an opaque pointer type called **pw_pointer**.

117. A callback interface is used to implement wrappers for memory allocation and freeing.

⟨Type Declarations 56⟩ +≡
typedef void(**pw_pointer_function*)(**pw_pointer** **p*);

118. The opaque pointer has a private struct definition, also called **pw_pointer**.

⟨Pointer Top 118⟩ ≡
struct pw_pointer {
 ⟨Pointer Struct Data 119⟩
};

See also sections 129, 131, 133, 135, 137, 139, 142, 144, 146, and 148.

This code is cited in section 348.

This code is used in section 114.

119. Most of the time, **pw_pointers** are not directly accessible to the user. Instead, they indirectly used via top-level patch functions. The **pw_pointer** type struct definition begins with a reference to the top-level patch it belongs to.

⟨Pointer Struct Data 119⟩ ≡
pw_patch * *patch*;

See also sections 120, 121, and 122.

This code is used in section 118.

120. Naturally, a pointer wrapper needs to have a **void** * pointer somewhere. This is included, alongside an optional *type* variable for a weakly enforced type-checking system.

⟨Pointer Struct Data 119⟩ +≡
int *type*;
void **ud*;

121. The *pw_pointer_function* callback prototype is used for freeing memory at the end of performance.

⟨Pointer Struct Data 119⟩ +≡
*pw_pointer_function**free*;

122. The **pw_pointer** types can be strung together in a linked list, allowing for many points of data to be automatically freed.

⟨Pointer Struct Data 119⟩ +≡
pw_pointer **next*;

123. A collection of **pw_pointer** values is contained in a list called a *pw_pointerlist*.

⟨Type Declarations 56⟩ +≡
typedef struct {
 ⟨Pointer List Struct Data 124⟩
} **pw_pointerlist**;

124. The **pw_pointerlist** implements a singly linked list. It has a *root* value and a *last* value. Values are appended to the end of a pointer list.

⟨Pointer List Struct Data 124⟩ ≡

```
pw_pointer *root;  
pw_pointer *last;
```

See also section 125.

This code is used in section 123.

125. The **pw_pointerlist** implementation keeps track of how many items there are in the list. This is used for list iteration.

⟨Pointer List Struct Data 124⟩ +≡

```
unsigned int size;
```

126. Pointer Functions. The following functions below describe operations on individual **pw_pointer** types.

127. A **pw_pointer** is created with the function *pw_pointer_create*.

128. $\langle \text{Header 6} \rangle + \equiv$

```
int pw_pointer_create(pw_patch * patch, pw_pointer **pointer, pw_pointer_functionfree, void *ud);
```

129. $\langle \text{Pointer Top 118} \rangle + \equiv$

```
int pw_pointer_create(pw_patch * patch, pw_pointer **pointer, pw_pointer_functionfree, void *ud)
{
    pw_pointer *pptr;
    int rc;

    rc = pw_memory_alloc(patch, sizeof(pw_pointer), (void **) &pptr);
    if (rc != PW_OK) return rc;
    *pointer = pptr;
    pptr->patch = patch;
    pptr->next =  $\Lambda$ ;
    pptr->type = 0;
    pptr->ud = ud;
    pptr->free = free;
    return PW_OK;
}
```

130. A **pw_pointer** is destroyed with the function *pw_pointer_destroy*. This will not only free the pointer itself, but the generic **void** * pointer contained inside of it.

$\langle \text{Header 6} \rangle + \equiv$

```
void pw_pointer_destroy(pw_pointer **pointer);
```

131. $\langle \text{Pointer Top 118} \rangle + \equiv$

```
void pw_pointer_destroy(pw_pointer **pointer)
{
    (*pointer)->free(*pointer);
    pw_memory_free((*pointer)->patch, (void **) pointer);
}
```

132. The function *pw_pointer_data* will return the user data contained inside of **pw_pointer**.

$\langle \text{Header 6} \rangle + \equiv$

```
void *pw_pointer_data(pw_pointer *pointer);
```

133. $\langle \text{Pointer Top 118} \rangle + \equiv$

```
void *pw_pointer_data(pw_pointer *pointer)
{
    return pointer->ud;
}
```

134. The function *pw_pointer_set_type* and *pw_pointer_get_type* will set and get the optional type id, which is set to 0 by default.

$\langle \text{Header 6} \rangle + \equiv$

```
void pw_pointer_set_type(pw_pointer *pointer, int id);
int pw_pointer_get_type(pw_pointer *pointer);
```

135. \langle Pointer Top 118 $\rangle + \equiv$

```
void pw_pointer_set_type(pw_pointer *pointer, int id)
{
    pointer->type = id;
}

int pw_pointer_get_id(pw_pointer *pointer)
{
    return pointer->type;
}
```

136. The function *pw_pointer_get_next* returns the value of the next pointer in the linked list.

\langle Header 6 $\rangle + \equiv$

```
pw_pointer *pw_pointer_get_next(pw_pointer *p);
```

137. \langle Pointer Top 118 $\rangle + \equiv$

```
pw_pointer *pw_pointer_get_next(pw_pointer *p)
{
    return p->next;
}
```

138. The function *pw_pointer_set_next* sets the next value of the pointer.

\langle Header 6 $\rangle + \equiv$

```
void pw_pointer_set_next(pw_pointer *p, pw_pointer *next);
```

139. \langle Pointer Top 118 $\rangle + \equiv$

```
void pw_pointer_set_next(pw_pointer *p, pw_pointer *next)
{
    p->next = next;
}
```


140. Pointer List Functions. The following functions below describe operations on pointer lists, otherwise known as **pw_pointerlist**.

141. A pointer list is initialized with the function *pw_pointerlist_init*.

⟨Header 6⟩ +≡

```
void pw_pointerlist_init(pw_pointerlist *plist);
```

142. ⟨Pointer Top 118⟩ +≡

```
void pw_pointerlist_init(pw_pointerlist *plist)
{
    plist->root = Λ;
    plist->last = Λ;
    plist->size = 0;
}
```

143. The function *pw_pointerlist_top* returns the top of the list.

⟨Header 6⟩ +≡

```
pw_pointer *pw_pointerlist_top(pw_pointerlist *plist);
```

144. ⟨Pointer Top 118⟩ +≡

```
pw_pointer *pw_pointerlist_top(pw_pointerlist *plist)
{
    return plist->root;
}
```

145. The function *pw_pointerlist_append* appends a pointer to the end of the list. It is assumed that the pointer value has already been allocated with *pw_pointer_create*.

⟨Header 6⟩ +≡

```
void pw_pointerlist_append(pw_pointerlist *plist, pw_pointer *p);
```

146. The function *pw_pointerlist_append* appends a value to the end of a linked list. It has an edge case for when the list is zero. When it is zero, the value is assigned to be *root* and *last*. Otherwise, the value is tacked on the the *last* next value, and then the *last* value is updated. The size of the list is incremented.

⟨Pointer Top 118⟩ +≡

```
void pw_pointerlist_append(pw_pointerlist *plist, pw_pointer *p)
{
    if (plist->size == 0) {
        plist->root = p;
    }
    else {
        pw_pointer_set_next(plist->last, p);
    }
    plist->last = p;
    plist->size++;
}
```

147. The function *pw_pointerlist_free* frees memory for all pointers contained inside of the pointer list.

⟨Header 6⟩ +≡

```
void pw_pointerlist_free(pw_pointerlist *plist);
```

148. \langle Pointer Top 118 $\rangle + \equiv$

```
void pw_pointerlist_free(pw_pointerlist *plist)
{
    unsigned int i;
    pw_pointer *next;
    pw_pointer *val;
    val = pw_pointerlist_top(plist);
    for (i = 0; i < plist->size; i++) {
        next = pw_pointer_get_next(val);
        pw_pointer_destroy(&val);
        val = next;
    }
}
```

149. Buffer Pool. In Patchwerk, audio-rate signals are read and written to a set of buffers. The buffer pool implements a scalable memory-efficient approach to handling buffers.

⟨ Top [8](#) ⟩ +≡
⟨ Buffer Pool Top [150](#) ⟩

150. Data. The buffer pool data consists of a buffer pool, which is a collection of underlying buffers.

```

⟨ Buffer Pool Top 150 ⟩ ≡
  ⟨ A Single Buffer 152 ⟩
  ⟨ The Buffer Pool 155 ⟩

```

This code is cited in section 265.

This code is used in section 149.

151. Both the buffer *pw_buffer* and the buffer pool *pw_bufferpool* are forward-declared as opaque pointers.

```

⟨ Type Declarations 56 ⟩ +≡
  typedef struct pw_buffer pw_buffer;
  typedef struct pw_bufferpool pw_bufferpool;

```

152. A single buffer contained inside of the buffer pool **pw_bufferpool** is known as a **pw_buffer**.

```

⟨ A Single Buffer 152 ⟩ ≡
  struct pw_buffer {
    ⟨ Data for a Single Buffer 153 ⟩
  };

```

See also sections 161, 163, 166, 168, 170, 172, 174, 176, 179, 181, 182, and 184.

This code is cited in section 63.

This code is used in section 150.

153. Every **pw_buffer** contains an unique *id*, which is used with the **pw_bufferpool** and *pw_stack* to keep track of the last release buffer.

```

⟨ Data for a Single Buffer 153 ⟩ ≡
  int id;

```

See also section 154.

This code is used in section 152.

154. The contents of **pw_buffer** are a pointer to the actual buffer, as well as a counter variable *read*, which keeps track of how many cables are actively reading from the particular buffer.

```

⟨ Data for a Single Buffer 153 ⟩ +≡
  int read;
  PWFLT * buf;

```

155. A buffer pool is known as a **pw_bufferpool**.

```

⟨ The Buffer Pool 155 ⟩ ≡
  struct pw_bufferpool {
    ⟨ Data for a Buffer Pool 156 ⟩
  };

```

See also sections 186, 188, 190, 192, 194, 196, 199, 207, 209, 211, 213, and 215.

This code is cited in section 63.

This code is used in section 150.

156. The core element of the buffer pool is an array of *buffers*.

```

⟨ Data for a Buffer Pool 156 ⟩ ≡
  pw_buffer * buffers;

```

See also sections 157, 158, 159, and 206.

This code is used in section 155.

157. The size of the buffers is contained is a signed integer called *size*.

⟨Data for a Buffer Pool 156⟩ +≡

int *size*;

158. The number of actively used buffers is kept track of in the variable called *nactive*.

⟨Data for a Buffer Pool 156⟩ +≡

int *nactive*;

159. To speed up search for next available buffers, the indice for the last freed buffer is stored in a signed variable called *last_free*. If such information isn't available, the value is set to be negative.

⟨Data for a Buffer Pool 156⟩ +≡

int *last_free*;

160. Buffer Functions. The following functions below describe operations on individual buffers.

161. Since the `pw_buffer` is an opaque type, the size of the it can be known with the function `pw_buffer_size` ■

```
⟨ A Single Buffer 152 ⟩ +=
size_t pw_buffer_size(void)
{
    return sizeof(pw_buffer);
}
```

162. ⟨ Header 6 ⟩ +=
`size_t pw_buffer_size(void);`

163. The function `pw_buffer_alloc` allocates memory for a single buffer with a provided buffer size.

```
⟨ A Single Buffer 152 ⟩ +=
void pw_buffer_alloc(pw_patch * patch, pw_buffer * buf, int size)
{
    pw_memory_alloc(patch, sizeof (PWFLT) * size, (void **) &buf->buf);
}
```

164. ⟨ Header 6 ⟩ +=
`void pw_buffer_alloc(pw_patch * patch, pw_buffer * buf, int size);`

165. The function `pw_buffer_free` frees memory contained inside of the buffer.

166.

```
⟨ A Single Buffer 152 ⟩ +=
void pw_buffer_free(pw_patch * patch, pw_buffer * buf)
{
    pw_memory_free(patch, (void **) &buf->buf);
}
```

167. ⟨ Header 6 ⟩ +=
`void pw_buffer_free(pw_patch * patch, pw_buffer * buf);`

168. The function `pw_buffer_init` initializes a `pw_buffer`. This function is desgined to be called before or after `pw_buffer_alloc`.

```
⟨ A Single Buffer 152 ⟩ +=
void pw_buffer_init(pw_buffer * buf)
{
    buf->id = -1;
    pw_buffer_reinit(buf);
}
```

169. ⟨ Header 6 ⟩ +=
`void pw_buffer_init(pw_buffer * buf);`

170. A buffer can be re-initialized with *pw_buffer_reinit*. This will zero out variables while retaining the buffer id.

```
⟨ A Single Buffer 152 ⟩ +=
void pw_buffer_reinit(pw_buffer *buf)
{
    buf->read = 0;
}
```

171. ⟨ Header 6 ⟩ +=
void pw_buffer_reinit(**pw_buffer** *buf);

172. The function *pw_buffer_mark* marks the *read* variable by incrementing it.

This only will happen when *=read=* is a positive or zero value. Negative values are used for buffers that are held indefinitely.

```
⟨ A Single Buffer 152 ⟩ +=
void pw_buffer_mark(pw_buffer *buf)
{
    if (buf->read ≥ 0) buf->read++;
}
```

173. ⟨ Header 6 ⟩ +=
void pw_buffer_mark(**pw_buffer** *buf);

174. The function *pw_buffer_unmark* unmarks the buffer by decrementing the *read* variable. If it is already zero, nothing happens and -2 is returned.

The after the read value has been decreased, it will once again check to see if the *read* variable is zero. A zero-value indicates that the buffer is not being ready by any nodes and is a free agent, and will return the buffer id. This information is useful for buffer pool and the buffer stack, as recently freed values can speed the search process for the next available free. Otherwise, -1 is returned.

```
⟨ A Single Buffer 152 ⟩ +=
int pw_buffer_unmark(pw_buffer *buf)
{
    if (buf->read < 0) return -3;
    if (buf->read ≡ 0) return -2;
    buf->read--;
    if (buf->read ≡ 0) return buf->id;
    return -1;
}
```

175. ⟨ Header 6 ⟩ +=
int pw_buffer_unmark(**pw_buffer** *buf);

176. The function *pw_buffer_data* returns the buffer pointer.

```
⟨ A Single Buffer 152 ⟩ +=
PWFLT * pw_buffer_data(pw_buffer *buf)
{
    return buf->buf;
}
```

177. ⟨ Header 6 ⟩ +=
PWFLT * pw_buffer_data(**pw_buffer** *buf);

178. When a buffer needs to be read indefinitely, it needs to be what is known as a held buffer. This is done with *pw_buffer_hold*.

⟨Header 6⟩ +≡
void *pw_buffer_hold*(**pw_buffer** *buf);

179. Generally speaking, a buffer is marked to be held buffer by setting the read parameter to be a negative value. The convention is to use -1.

⟨A Single Buffer 152⟩ +≡
void *pw_buffer_hold*(**pw_buffer** *buf)
{
 buf→read = -1;
}

180. User-space held buffers are noted as having a specific read value of -2. This can be marked using the function *pw_buffer_holdu*.

⟨Header 6⟩ +≡
void *pw_buffer_holdu*(**pw_buffer** *buf);

181.

⟨A Single Buffer 152⟩ +≡
void *pw_buffer_holdu*(**pw_buffer** *buf)
{
 buf→read = -2;
}

182.

⟨A Single Buffer 152⟩ +≡
int *pw_buffer_unhold*(**pw_buffer** *buf)
{
 if (buf→read < 0) {
 buf→read = 0;
 return 1;
 }
 else {
 return 0;
 }
}

183. ⟨Header 6⟩ +≡
int *pw_buffer_unhold*(**pw_buffer** *buf);

184. TODO: write words here.

⟨A Single Buffer 152⟩ +≡
int *pw_buffer_id*(**pw_buffer** *buf)
{
 return buf→id;
}

185. ⟨Header 6⟩ +≡
int *pw_buffer_id*(**pw_buffer** *buf);

186. Initialization Functions. The following functions below describe operations related to creation and initialization on the buffer pool.

⟨The Buffer Pool 155⟩ +≡

```
void pw_bufferpool_init(pw_bufferpool *pool)
{
    pool->size = 0;
    pool->nactive = 0;
    pool->usrnactive = 0;
}
```

187. ⟨Header 6⟩ +≡

```
void pw_bufferpool_init(pw_bufferpool *pool);
```

188. The function *pw_bufferpool_create* allocates and initializes a buffer pool. The arguments required are the number of buffers allocated *nbuf*, then the block size *blksize*.

⟨The Buffer Pool 155⟩ +≡

```
void pw_bufferpool_create(pw_patch *patch, pw_bufferpool *pool, int nbuf, int blksize)
{
    int i;
    pool->size = nbuf;
    pool->nactive = 0;
    pw_memory_alloc(patch, sizeof(pw_buffer) * nbuf, (void **) &pool->buffers);
    for (i = 0; i < nbuf; i++) {
        pw_buffer_alloc(patch, &pool->buffers[i], blksize);
        pw_buffer_init(&pool->buffers[i]);
        pool->buffers[i].id = i;
    }
}
```

189. ⟨Header 6⟩ +≡

```
void pw_bufferpool_create(pw_patch *patch, pw_bufferpool *pool, int nbuf, int blksize);
```

190. ⟨The Buffer Pool 155⟩ +≡

```
void pw_bufferpool_reset(pw_bufferpool *pool)
{
    int i;
    pool->last_free = -1;
    pool->nactive = 0;
    for (i = 0; i < pool->size; i++) {
        if (pool->buffers[i].read ≥ 0) {
            pw_buffer_reinit(&pool->buffers[i]);
        }
        else {
            pool->nactive++;
        }
    }
}
```

191. ⟨Header 6⟩ +≡

```
void pw_bufferpool_reset(pw_bufferpool *pool);
```

192. The function *pw_bufferpool_destroy* frees any memory allocated by *pw_bufferpool_create*.

⟨The Buffer Pool 155⟩ +≡

```
void pw_bufferpool_destroy(pw_patch * patch, pw_bufferpool * pool)
{
    int i;
    for (i = 0; i < pool->size; i++) {
        pw_buffer_free(patch, &pool->buffers[i]);
    }
    if (pool->size > 0) {
        pw_memory_free(patch, (void **) &pool->buffers);
    }
}
```

193. ⟨Header 6⟩ +≡

```
void pw_bufferpool_destroy(pw_patch * patch, pw_bufferpool * pool);
```

194. The function *pw_bufferpool_nactive* returns the number of active buffers in the buffer pool.

⟨The Buffer Pool 155⟩ +≡

```
int pw_bufferpool_nactive(pw_bufferpool * pool)
{
    return pool->nactive;
}
```

195. ⟨Header 6⟩ +≡

```
int pw_bufferpool_nactive(pw_bufferpool * pool);
```

196. The function *pw_bufferpool_unhold* is a wrapper around *pw_buffer_unhold*. It is used to both unhold the buffer, and update the number of active buffers in the buffer pool.

Some buffers are not managed by a pool. When they are initialized, they are given a negative value. Any buffers with a non-zero id will be skipped.

Note: At the moment there are no checks on if the buffer actually belongs to the buffer pool. The function assumes that the buffer is valid, making this procedure fragile in the current state.

⟨The Buffer Pool 155⟩ +≡

```
int pw_bufferpool_unhold(pw_bufferpool * pool, pw_buffer * buf)
{
    if (buf->id < 0) return 0;
    if (pw_buffer_unhold(buf)) {
        pool->nactive--;
        return 1;
    }
    else {
        return 0;
    }
}
```

197. ⟨Header 6⟩ +≡

```
int pw_bufferpool_unhold(pw_bufferpool * pool, pw_buffer * buf);
```

198. Finding the next free buffer. The function *pw_bufferpool_nextfree* returns the next free buffer. This function will return `PW_OK` on success and `PW_NOT_OK` on failure.

⟨Header 6⟩ +≡

```
int pw_bufferpool_nextfree(pw_bufferpool *pool, pw_buffer **buf);
```

199.

⟨The Buffer Pool 155⟩ +≡

```
int pw_bufferpool_nextfree(pw_bufferpool *pool, pw_buffer **buf)
{
    int i;
    if (⟨Check if Buffer Pool is Full 200⟩) {
        return PW_POOL_FULL;
    }
    else if (⟨Check for previously freed buffer 201⟩) {
        ⟨Use recently freed buffer 202⟩
    }
    else {
        ⟨Use brute force to find next free buffer 203⟩
    }
    ⟨Successful Finalization 204⟩
    return PW_OK;
}
```

200. If the pool is full when the number of actively used buffers matches the size.

⟨Check if Buffer Pool is Full 200⟩ ≡

$$pool\text{-}nactive \geq pool\text{-}size$$

This code is used in section 199.

201. To avoid brute force lookup found in ⟨Use brute force to find next free buffer 203⟩, there is a check for any recently freed-up buffers. This is done by checking the *last_free* variable for a indice greater than or equal to zero.

⟨Check for previously freed buffer 201⟩ ≡

$$pool\text{-}last_free \geq 0$$

This code is used in section 199.

202. Assuming there is indeed a valid value stored in *last_free*, the buffer from that indice is used as the next free value. Since it is claimed, it is no longer free, so *last_free* is set to be an arbitray negative value.

⟨Use recently freed buffer 202⟩ ≡

$$*buf = \&pool\text{-}buffers[pool\text{-}last_free];$$

$$pool\text{-}last_free = -1;$$

This code is used in section 199.

203. If all else fails, brute force is used to find the next value. The program steps through the array of buffers and checks for the next available free buffer. A buffer is considered free if the *read* variable is 0.

⟨ Use brute force to find next free buffer 203 ⟩ ≡

```

for (i = 0; i < pool→size; i++) {
    if (pool→buffers[i].read ≡ 0) {
        *buf = &pool→buffers[i];
        break;
    }
}

```

This code is cited in section 201.

This code is used in section 199.

204. On successful completion of finding an available buffer, the number of active buffers is incremented by one, and the buffer is marked as read.

⟨ Successful Finalization 204 ⟩ ≡

```

pool→nactive++;
pw→buffer→mark(*buf);

```

This code is used in section 199.

205. User-space buffers.

Userspace buffers refer to buffers that are explicitly held at user-level. At the time of writing, this refers to using the Runt patchwerk interface.

Buffers that are held cannot be re-used until they are explicitly unheld. While this is helpful for constructing more complicating patches, it can be easy enough to forget about a unholding a buffer. If a buffer fails to be unheld before the end of a patch, that buffer becomes permanently lost, and a resource leak occurs as a result of this.

To mitigate so-called "buffer leaks", a user-space buffer is explicitly kept track of during the population of a new patch. If at the end of a patch there are still active user-space buffers, then patchwerk has the ability to issue a warning and potentially close those buffers.

This section outlines the specific data structures and functions associated with user-space buffers in the buffer pool.

206. A user-space buffer is just like any other buffer, except that it is kept track of in a special counter variable. This variable is called *usrnactive*.

⟨Data for a Buffer Pool 156⟩ +≡

```
int usrnactive;
```

207. The function *pw_bufferpool_holdu* marks a buffer as being held at user-level. This will be kept track of, and marked separately from other held buffers. The *usrnactive* counter will move upwards, in addition to the regular *nactive* counter.

⟨The Buffer Pool 155⟩ +≡

```
int pw_bufferpool_holdu(pw_bufferpool *pool, pw_buffer *buf)
{
    if (buf == Λ) return PW_NULL_VALUE;
    if (buf->read ≥ 0) { /* make sure the buffer isn't marked already */
        pw_buffer_holdu(buf);
        pool->usrnactive++;
        pool->nactive++;
        return PW_OK;
    }
    return PW_INVALID_BUFFER;
}
```

208. ⟨Header 6⟩ +≡

```
int pw_bufferpool_holdu(pw_bufferpool *pool, pw_buffer *buf);
```

209. The function *pw_bufferpool_unholdu* unholds a buffer being held at user-level. If the buffer is not marked as being a user-level buffer, the function returns *PW_INVALID_BUFFER*. Otherwise the buffer is unheld, and the *nactive* and *usrnactive* fields are respectively decreased.

When buffers have an initialized value of -1, they are assumed to be self-managed and are ignored. For now, this will return *PW_OK*. In the future, a *PW_IGNORE* error code may be introduced if needed.

⟨The Buffer Pool 155⟩ +≡

```
int pw_bufferpool_unholdu(pw_bufferpool *pool, pw_buffer *buf)
{
    if (buf == Λ) return PW_NULL_VALUE;
    if (buf->id == -1) return PW_OK;
    if (buf->read != -2) return PW_INVALID_BUFFER;
    if (¬pw_buffer_unhold(buf)) return PW_NOT_OK;
    pool->nactive--;
    pool->usrnactive--;
    return PW_OK;
}
```

210. ⟨Header 6⟩ +≡

```
int pw_bufferpool_unholdu(pw_bufferpool *pool, pw_buffer *buf);
```

211. The function *pw_bufferpool_unholdu_all* unholds all the buffers marked as being held by the user. If there are no user-level buffers being held, then the function immediately returns. At the moment, this is a rather brute-force *O(n)* operation that simply iterates through all the buffers.

Note that *pw_bufferpool_unholdu* is called on every buffer. This should be harmless, since the function immediately returns if the buffer is not a marked user-space buffer.

⟨The Buffer Pool 155⟩ +≡

```
int pw_bufferpool_unholdu_all(pw_bufferpool *pool)
{
    int i;
    if (pool->usrnactive == 0) return PW_NOT_OK; /* TODO: better error handling */
    for (i = 0; i < pool->size; i++) {
        pw_bufferpool_unholdu(pool, &pool->buffers[i]);
    }
    return PW_OK;
}
```

212. ⟨Header 6⟩ +≡

```
int pw_bufferpool_unholdu_all(pw_bufferpool *pool);
```

213. The function *pw_bufferpool_uactive* returns the number of currently active userspace buffers.

⟨The Buffer Pool 155⟩ +≡

```
int pw_bufferpool_uactive(pw_bufferpool *pool)
{
    return pool->usrnactive;
}
```

214. ⟨Header 6⟩ +≡

```
int pw_bufferpool_uactive(pw_bufferpool *pool);
```

215. The function *pw_bufferpool_clear_last_free* sets the *last_free* value in the pool to be negative. This is called by the *bhold* word in Runt after popping the buffer off the stack. This is done so that a held buffer is taken out of the buffer pool.

⟨The Buffer Pool 155⟩ +≡

```
void pw_bufferpool_clear_last_free(pw_bufferpool *pool)
{
    pool-last_free = -1;
}
```

216. ⟨Header 6⟩ +≡

```
void pw_bufferpool_clear_last_free(pw_bufferpool *pool);
```

217. Buffer Stack. Buffers from the buffer pool are accessed via a stack-like interface known as the **buffer stack**.

$\langle \text{Top } 8 \rangle + \equiv$
 $\langle \text{Stack Top } 218 \rangle$

218. Buffer Stack Data.

⟨Stack Top 218⟩ ≡
⟨Stack Data 219⟩

See also sections 222, 224, 226, 228, 230, 232, 234, 236, 238, 240, 242, 244, and 248.

This code is cited in section 265.

This code is used in section 217.

219.

⟨Stack Data 219⟩ ≡
struct pw_stack {
 int *pos*;
 int *size*;
 pw_bufferpool **pool*;
 pw_buffer ***buffers*;
};

This code is used in section 218.

220. ⟨Type Declarations 56⟩ +≡
 typedef struct pw_stack pw_stack;

221. Buffer Stack Functions.

222. *pw_stack_init* initializes data and variables in the stack. It does not allocate any memory. In addition to supplying a **pw_stack**, an allocated **pw_bufferpool** must also be supplied.

The size of the stack by default is 0. It must be allocated with a function like *pw_stack_alloc*.

```

⟨Stack Top 218⟩ +=
void pw_stack_init(pw_stack *stack, pw_bufferpool *pool)
{
    stack->pool = pool;
    stack->pos = 0;
    stack->size = 0;
}

```

223. ⟨Header 6⟩ +=
void *pw_stack_init*(**pw_stack** *stack, **pw_bufferpool** *pool);

224. The function *pw_stack_alloc* allocates memory a static stack. The stack holds an array of buffer pointers. On success, **PW_OK** will be returned. On failure, **PW_NOT_OK** will be returned.

```

⟨Stack Top 218⟩ +=
int pw_stack_alloc(pw_patch *patch, pw_stack *stack, int size)
{
    stack->size = size;
    return pw_memory_alloc(patch, sizeof(pw_buffer *) * size, (void **) &stack->buffers);
}

```

225. ⟨Header 6⟩ +=
int *pw_stack_alloc*(*pw_patch* *patch, **pw_stack** *stack, **int** size);

226. The function **pw_stack** frees the data previously allocated by *malloc* in *pw_stack_alloc*.

```

⟨Stack Top 218⟩ +=
int pw_stack_free(pw_patch *patch, pw_stack *stack)
{
    return pw_memory_free(patch, (void **) &stack->buffers);
}

```

227. ⟨Header 6⟩ +=
int *pw_stack_free*(*pw_patch* *patch, **pw_stack** *stack);

228. A buffer is pushed to the stack with *pw_stack_push*. The stack will search the buffer pool for the next free buffer, which will then get pushed on to the stack. On success, the buffer will then get saved to the variable *buf*, and the buffer will be stored in the next position in the stack. The stack position is one-indexed, so the current position can be used before incrementing the stack position.

If the stack or pool is full, the function will return a value of `PW_NOT_OK`.

⟨Stack Top 218⟩ +≡

```
int pw_stack_push(pw_stack *stack, pw_buffer **buf)
{
    pw_buffer *pbuf;
    int rc;
    pbuf = Λ;
    if (stack->pos ≥ stack->size) {
        return PW_STACK_OVERFLOW;
    }
    rc = pw_bufferpool_nextfree(stack->pool, &pbuf);
    if (rc ≠ PW_OK) return rc;
    stack->buffers[stack->pos] = pbuf;
    stack->pos++;
    if (buf ≠ Λ) *buf = pbuf;
    return PW_OK;
}
```

229. ⟨Header 6⟩ +≡

```
int pw_stack_push(pw_stack *stack, pw_buffer **buf);
```

230. To push a specific buffer onto the stack, use the function *pw_stack_push_buffer*. This particular function is used with the *cable* word defined in the runt patchwerk interface for buffers that have been held.

⟨Stack Top 218⟩ +≡

```
int pw_stack_push_buffer(pw_stack *stack, pw_buffer *buf)
{
    if (stack->pos ≥ stack->size) {
        return PW_STACK_OVERFLOW;
    }
    stack->buffers[stack->pos] = buf;
    stack->pos++;
    return PW_OK;
}
```

231. ⟨Header 6⟩ +≡

```
int pw_stack_push_buffer(pw_stack *stack, pw_buffer *buf);
```

232. A buffer is popped from the stack using *pw_stack_pop*. First, the stack is checked for any items on the stack. Then, the last item on the stack is assigned to the variable *buf*.

One thing to note is that the stack position uses one-indexing as opposed to C's zero-indexing scheme, so the position in the array needs to be subtracted by one.

Once the stack successfully pops a buffer, the buffer must be unmarked to indicate it is being read by one less node with *pw_buffer_unmark*. If the buffer is not actively being read, it will the return value id of the buffer which will set the *last_free* value.

The *pw_stack_pop* function will save the popped buffer value to the variable *buf* if it not NULL.

⟨Stack Top 218⟩ +=

```
static int pop_from_stack(pw_stack *stack, pw_buffer **buf)
{
    if (stack->pos == 0) return PW_NOT_OK;
    *buf = stack->buffers[stack->pos - 1];
    stack->pos--;
    return PW_OK;
}

int pw_stack_pop(pw_stack *stack, pw_buffer **buf)
{
    int rc;
    pw_buffer *tmp;
    rc = pop_from_stack(stack, &tmp);
    if (rc != PW_OK) return PW_NOT_OK;
    rc = pw_buffer_unmark(tmp);
    if (rc >= 0) {
        stack->pool->last_free = rc;
        stack->pool->nactive--;
    }
    if (buf != Λ) *buf = tmp;
    return PW_OK;
}
```

233. ⟨Header 6⟩ +=

```
int pw_stack_pop(pw_stack *stack, pw_buffer **buf);
```

234. The last buffer on the stack can be duplicated with *pw_stack_dup*. This operation copies the buffer memory address, rather than copying the contents to a new buffer. This is done through a direct assignment operation. Before this is done, the edge cases are checked to make sure that there are items on the stack to duplicate *and* that there are enough spaces on the stack.

Since the buffer memory location is being duplicated instead of the contents of the buffer, the buffer must be marked as being read again.

```

⟨Stack Top 218⟩ +=
int pw_stack_dup(pw_stack *stack)
{
    pw_buffer *buf;
    if (stack->pos == 0) {
        return PW_NOT_OK;    /* No items on the stack */
    }
    if (stack->pos ≥ stack->size - 1) {
        return PW_NOT_OK;    /* Not enough room on the stack */
    }
    stack->buffers[stack->pos] = stack->buffers[stack->pos - 1];
    buf = stack->buffers[stack->pos];
    pw_buffer_mark(buf);
    stack->pos++;
    return PW_OK;
}

```

235. ⟨Header 6⟩ +=
int pw_stack_dup(**pw_stack** *stack);

236. The last item on the stack is removed with *pw_stack_drop*. This is done by subtracting the stack position by one. This function will return an error of PW_NOT_OK if there are no items to drop on the stack.

```

⟨Stack Top 218⟩ +=
int pw_stack_drop(pw_stack *stack)
{
    if (stack->pos == 0) return PW_NOT_OK;
    stack->pos--;
    return PW_OK;
}

```

237. ⟨Header 6⟩ +=
int pw_stack_drop(**pw_stack** *stack);

238. The last two items can swap positions using the function *pw_stack_swap*. This operation will fail if there aren't enough items on the stack.

```

⟨Stack Top 218⟩ +=
int pw_stack_swap(pw_stack *stack)
{
    pw_buffer *tmp;
    if (stack->pos < 2) return PW_NOT_OK;
    tmp = stack->buffers[stack->pos - 1];
    stack->buffers[stack->pos - 1] = stack->buffers[stack->pos - 2];
    stack->buffers[stack->pos - 2] = tmp;
    return PW_OK;
}

```

239. \langle Header 6 $\rangle + \equiv$

```
int pw_stack_swap(pw_stack *stack);
```

240. Sometimes, there are some cases where a particular buffer needs to be held indefinitely. The function *pw_stack_hold* implements a hold operation, which pops the last buffer on the stack and marks it to be held. This buffer will not be written to again until the buffer is unheld with the function *pw_buffer_unhold*.

pw_stack_hold will save the popped **pw_buffer** into the value *buf*.

\langle Stack Top 218 $\rangle + \equiv$

```
int pw_stack_hold(pw_stack *stack, pw_buffer **buf)
{
    if (pop_from_stack(stack, buf)  $\neq$  PW_OK) {
        return PW_NOT_OK;
    }
    pw_buffer_hold(*buf);
    return PW_OK;
}
```

241. \langle Header 6 $\rangle + \equiv$

```
int pw_stack_hold(pw_stack *stack, pw_buffer **buf);
```

242. \langle Stack Top 218 $\rangle + \equiv$

```
int pw_stack_size(pw_stack *stack)
{
    return stack->size;
}
```

243. \langle Header 6 $\rangle + \equiv$

```
int pw_stack_size(pw_stack *stack);
```

244. \langle Stack Top 218 $\rangle + \equiv$

```
int pw_stack_pos(pw_stack *stack)
{
    return stack->pos;
}
```

245. The function *pw_stack_pos* returns the current stack position.

246. \langle Header 6 $\rangle + \equiv$

```
int pw_stack_pos(pw_stack *stack);
```

247. The function *pw_stack_reset* resets the stack back to zero. This function is used internally, and is used in for live-coding environments where on-the-fly re-evaluation required.

\langle Header 6 $\rangle + \equiv$

```
void pw_stack_reset(pw_stack *stack);
```

248.

\langle Stack Top 218 $\rangle + \equiv$

```
void pw_stack_reset(pw_stack *stack)
{
    stack->pos = 0;
}
```

249. Patch. The patch interface provides an interface for controlling and connecting multiple nodes and cables.

⟨ Top [8](#) ⟩ +≡
 ⟨ Patch Data [250](#) ⟩
 ⟨ Patch Top [264](#) ⟩

250. Data. The struct for patch data is contained in a struct called *pw_patch*.

```

< Patch Data 250 > ≡
    struct pw_patch {
        < Variables in Patch Data 252 >
    };

```

This code is used in section 249.

251. The **pw_patch** struct is forward declared as an opaque pointer.

```

< Type Declarations 56 > +≡
    typedef struct pw_patch pw_patch;

```

252.

```

< Variables in Patch Data 252 > ≡
    pw_node *nodes;
    pw_node *last;
    int nnodes;

```

See also sections 253, 254, 255, 256, 257, 258, 259, 260, 261, 389, and 398.

This code is used in section 250.

253. A patch has a delegate output cable. By default, this is set to be a constant of zero via an internal cable.

```

< Variables in Patch Data 252 > +≡
    pw_cable *out;
    pw_cable zero;

```

254. The global blocksize is stored in the patch.

```

< Variables in Patch Data 252 > +≡
    int blksize;

```

255. When dealing with single-sample rendering environments like Sporth or Soundpipe, a counter is need to keep track of when to render a new block. This function is kept in a variable called *counter*.

```

< Variables in Patch Data 252 > +≡
    int counter;

```

256. Node ID management can be abstracted away inside of a patch through an internal variable called *nodepos*, which keeps track of the current node position.

```

< Variables in Patch Data 252 > +≡
    int nodepos;

```

257. Any user data is contained in a linked list-type called a **pw_pointerlist**.

```

< Variables in Patch Data 252 > +≡
    pw_pointerlist plist;

```

258. Contained inside the patch data is a buffer pool.

```

< Variables in Patch Data 252 > +≡
    pw_bufferpool pool;

```

259. The buffer pool is controlled via a LIFO (last in, first out) stack interface.

```

< Variables in Patch Data 252 > +≡
    pw_stack stack;

```


260. A variable storing the internal sampling rate can be used by nodes created via the patchwerk interface.

⟨ Variables in Patch Data 252 ⟩ +≡

int *sr*;

261. A generic **void** * pointer is used to store any persistent user data needed by nodes. In the Runt interface, this is the Soundpipe data struct.

⟨ Variables in Patch Data 252 ⟩ +≡

void **ud*;

262. Functions.

263. The function *pw_patch_init* initializes and zeros out variables inside of **pw_patch**. No memory allocation happens here.

⟨Header 6⟩ +≡

```
void pw_patch_init(pw_patch *patch, int blksize);
```

264. ⟨Patch Top 264⟩ ≡

```
void pw_patch_init(pw_patch *patch, int blksize)
{
    patch->blksize = blksize;
    pw_bufferpool_init(&patch->pool);
    pw_patch_srate_set(patch, 44100);    /* default sampling rate of 44.1kHz */
    pw_memory_defaults(patch);
    pw_print_init(patch);
    pw_patch_reinit(patch);
}
```

See also sections 266, 268, 270, 272, 274, 276, 278, 280, 282, 285, 287, 289, 292, 294, 296, 298, 300, 302, 304, 306, 397, 400, and 403.

This code is used in section 249.

265. The function *pw_patch_alloc* does all the memory allocation needed for the patch. This includes allocating the buffer pool ⟨Buffer Pool Top 150⟩ and buffer stack ⟨Stack Top 218⟩.

The variable *nbuffers* is the number of buffers to be allocated in the buffer pool, and the variable *stack_size* is the number of stack slots to be allocated for the buffer stack.

Reasonable numbers for *nbuffers* and *stack_size* are 8 and 10, respectively.

⟨Header 6⟩ +≡

```
void pw_patch_alloc(pw_patch *patch, int nbuffers, int stack_size);
```

266. ⟨Patch Top 264⟩ +≡

```
void pw_patch_alloc(pw_patch *patch, int nbuffers, int stack_size)
{
    pw_bufferpool_create(patch, &patch->pool, nbuffers, patch->blksize);
    pw_bufferpool_reset(&patch->pool);
    pw_stack_init(&patch->stack, &patch->pool);
    pw_stack_alloc(patch, &patch->stack, stack_size);
}
```

267. The function *pw_patch_realloc* will re-allocate the buffer pool and stack.

⟨Header 6⟩ +≡

```
void pw_patch_realloc(pw_patch *patch, int nbuffers, int stack_size, int blksize);
```

268. Because the internal memory interface has no bindings for *realloc*, the patch will free the existing stack and pool, and allocate it. For this reason, use caution when calling this!

⟨Patch Top 264⟩ +≡

```
void pw_patch_realloc(pw_patch *patch, int nbuffers, int stack_size, int blksize)
{
    pw_bufferpool_destroy(patch, &patch->pool);
    pw_stack_free(patch, &patch->stack);
    patch->blksize = blksize;
    pw_patch_alloc(patch, nbuffers, stack_size);
}
```

269. The function *pw_patch_reinit* can reinitialize the **pw_patch** interface. This function is particularly useful for implementing live coding environments.

⟨Header 6⟩ +≡

```
void pw_patch_reinit(pw_patch *patch);
```

270. ⟨Patch Top 264⟩ +≡

```
void pw_patch_reinit(pw_patch *patch)
{
    pw_cable_init( $\Lambda$ , &patch->zero);
    patch->counter = 0;
    patch->nodepos = 0;
    pw_patch_clear(patch);
    pw_stack_reset(&patch->stack);
    pw_bufferpool_reset(&patch->pool);
}
```

271. The function *pw_patch_clear* will partially reset a **pw_patch**. Clearing is useful in some cases involving subpatches.

⟨Header 6⟩ +≡

```
void pw_patch_clear(pw_patch *patch);
```

272. Clearing a patch with *pw_patch_clear* is similar to reinitializing a patch with *pw_patch_reinit*, except that it will not reset the buffer stack or buffer pool. This function is useful in some situations involving subpatches. One such case of this would be using subpatches to build a polyphonic synthesizer inside of a larger patch. If patchwerk is reinitialized *pw_patch_reinit* instead of cleared with *pw_patch_clear*, audio-rate signals existing on the buffer stack can be at risk of being overwritten.

⟨Patch Top 264⟩ +≡

```
void pw_patch_clear(pw_patch *patch)
{
    patch->out = &patch->zero;
    patch->nodes =  $\Lambda$ ;
    patch->last =  $\Lambda$ ;
    patch->nnodes = 0;
    pw_pointerlist_init(&patch->plist);
}
```

273. Any nodes allocated with *pw_patch_new_node* must be freed with *pw_patch_free_nodes*. Memory will only be freed if the number of nodes is non-zero.

⟨Header 6⟩ +≡

```
void pw_patch_free_nodes(pw_patch *patch);
```

274. \langle Patch Top 264 $\rangle + \equiv$

```
void pw_patch_free_nodes(pw_patch *patch)
{
    int i;
    pw_node *node;
    pw_node *next;
    node = patch->nodelist;
    for (i = 0; i < patch->nnodes; i++) {
        next = pw_node_get_next(node);
        free(node);
        node = next;
    }
    patch->nnodes = 0;
}
```

275. The functions define below will call the setup, compute, and destroy functions for each individual node.

\langle Header 6 $\rangle + \equiv$

```
void pw_patch_setup(pw_patch *patch);
void pw_patch_destroy(pw_patch *patch);
void pw_patch_compute(pw_patch *patch);
```

276. \langle Patch Top 264 $\rangle + \equiv$

```

void pw_patch_setup(pw_patch *patch)
{
    int n;
    pw_node *node;
    pw_node *next;
    node = patch->nodes;
    for (n = 0; n < patch->nnodes; n++) {
        next = pw_node_get_next(node);
        pw_node_setup(node);
        node = next;
    }
}

void pw_patch_destroy(pw_patch *patch)
{
    int n;
    pw_node *node;
    pw_node *next;
    node = patch->nodes;
    for (n = 0; n < patch->nnodes; n++) {
        next = pw_node_get_next(node);
        pw_node_destroy(node);
        node = next;
    }
    pw_pointerlist_free(&patch->plist);
    pw_bufferpool_destroy(patch, &patch->pool);
    pw_stack_free(patch, &patch->stack);
}

void pw_patch_compute(pw_patch *patch)
{
    int n;
    pw_node *node;
    pw_node *next;
    node = patch->nodes;
    for (n = 0; n < patch->nnodes; n++) {
        next = pw_node_get_next(node);
        pw_node_compute(node);
        node = next;
    }
}

```

277. The function `pw_patch_set_out` sets the output cable of the entire patch.

\langle Header 6 $\rangle + \equiv$

```

void pw_patch_set_out(pw_patch *patch, pw_cable *cable);

```

278. \langle Patch Top 264 $\rangle + \equiv$

```

void pw_patch_set_out(pw_patch *patch, pw_cable *cable)
{
    patch->out = cable;
}

```

279. The function *pw_patch_get_out* returns the output pointer of the patch.

⟨Header 6⟩ +≡

```
pw_cable *pw_patch_get_out(pw_patch *patch);
```

280. ⟨Patch Top 264⟩ +≡

```
pw_cable *pw_patch_get_out(pw_patch *patch)
{
    return patch->out;
}
```

281. The function *pw_patch_size* returns the size of the **pw_patch** data struct.

⟨Header 6⟩ +≡

```
size_t pw_patch_size();
```

282. ⟨Patch Top 264⟩ +≡

```
size_t pw_patch_size()
{
    return sizeof(pw_patch);
}
```

283. High Level Functions. These are functions that provide small abstractions around common tasks.

284. The function *pw_patch_tick* is a tick function that can be used inside of single-sample DSP function environments such as Soundpipe or Sporth. It uses an internal counter that keeps track of when to render a new block of audio.

⟨Header 6⟩ +≡

```
PWFLTpw_patch_tick(pw_patch *patch);
```

285. ⟨Patch Top 264⟩ +≡

```
PWFLTpw_patch_tick(pw_patch *patch)
{
    PWFLTsmp;
    pw_cable *out;
    if (patch->counter ≡ 0) {
        pw_patch_compute(patch);
    }
    out = pw_patch_get_out(patch);
    smp = pw_cable_get(out, patch->counter);
    patch->counter = (patch->counter + 1) % patch->blksize;
    return smp;
}
```

286. The function *pw_patch_new_node* allocates a node and stores in the pointer *node*.

⟨Header 6⟩ +≡

```
int pw_patch_new_node(pw_patch *patch, pw_node **node);
```

287. ⟨Patch Top 264⟩ +≡

```
int pw_patch_new_node(pw_patch *patch, pw_node **node)
{
    pw_node *tmp;
    int rc;
    if (patch ≡ Λ) return PW_NULL_VALUE;
    rc = pw_memory_alloc(patch, sizeof(pw_node), (void **) &tmp);
    if (rc ≠ PW_OK) return rc;
    pw_node_init(tmp, patch->blksize);
    pw_node_set_id(tmp, patch->nnodes);
    pw_node_set_patch(tmp, patch);
    if (patch->nnodes ≡ 0) {
        patch->nodes = tmp;
    }
    else {
        pw_node_set_next(patch->last, tmp);
    }
    patch->nnodes++;
    patch->last = tmp;
    *node = tmp;
    return PW_OK;
}
```

288. The function *pw_patch_new_cable* allocates and initializes a new cable that is independent from a node.

⟨Header 6⟩ +≡

```
int pw_patch_new_cable(pw_patch *patch, pw_cable **cable);
```

289. ⟨Patch Top 264⟩ +≡

```
static void delete_cable(pw_pointer *p)
{
    pw_cable *c;
    c = pw_pointer_data(p);
    pw_memory_free(patch, (void **) &c);
}

int pw_patch_new_cable(pw_patch *patch, pw_cable **cable)
{
    pw_cable *tmp;
    int rc;

    rc = pw_memory_alloc(patch, sizeof(pw_cable), (void **) &tmp);
    if (rc != PW_OK) return rc;
    pw_cable_init(patch, tmp);
    pw_patch_append_userdata(patch, delete_cable, tmp);
    *cable = tmp;
    return PW_OK;
}
```

290. The function *pw_patch_append_userdata* creates and appends a **pw_pointer** to the internal **pw_pointerlist** inside of the patch.

291. ⟨Header 6⟩ +≡

```
int pw_patch_append_userdata(pw_patch *patch, pw_pointer_function dfun, void *ud);
```

292. ⟨Patch Top 264⟩ +≡

```
int pw_patch_append_userdata(pw_patch *patch, pw_pointer_function dfun, void *ud)
{
    pw_pointer *ptr;
    int rc;

    rc = pw_pointer_create(patch, &ptr, dfun, ud);
    if (rc != PW_OK) return rc;
    pw_pointerlist_append(&patch->plist, ptr);
    return PW_OK;
}
```

293. ⟨Header 6⟩ +≡

```
pw_stack *pw_patch_stack(pw_patch *patch);
```

294. The function *pw_patch_stack* returns the **pw_stack** contained inside of a **pw_patch**.

⟨Patch Top 264⟩ +≡

```
pw_stack *pw_patch_stack(pw_patch *patch)
{
    return &patch->stack;
}
```


295. The audio block size can be returned using *pw_patch_blksize*.

```
⟨Header 6⟩ +≡
    int pw_patch_blksize(pw_patch *patch);
```

296.

```
⟨Patch Top 264⟩ +≡
    int pw_patch_blksize(pw_patch *patch)
    {
        return patch-blksize;
    }
```

297. The function *pw_patch_pool* returns the bufferpool inside of a **pw_patch**

```
⟨Header 6⟩ +≡
    pw_bufferpool *pw_patch_pool(pw_patch *patch);
```

298.

```
⟨Patch Top 264⟩ +≡
    pw_bufferpool *pw_patch_pool(pw_patch *patch)
    {
        return &patch-pool;
    }
```

299. The sampling rate can be set and retrieved using *pw_patch_srate_set* and *pw_patch_srate_get*, respectively.

```
⟨Header 6⟩ +≡
    void pw_patch_srate_set(pw_patch *patch, int sr);
    int pw_patch_srate_get(pw_patch *patch);
```

```
300.  ⟨Patch Top 264⟩ +≡
    void pw_patch_srate_set(pw_patch *patch, int sr)
    {
        patch-sr = sr;
    }
    int pw_patch_srate_get(pw_patch *patch)
    {
        return patch-sr;
    }
```

301. The user data can be set and get using *pw_patch_data_set* and *pw_patch_data_get*, respectively.

```
⟨Header 6⟩ +≡
    void pw_patch_data_set(pw_patch *patch, void *ud);
    void *pw_patch_data_get(pw_patch *patch);
```

302. \langle Patch Top 264 $\rangle + \equiv$

```
void pw_patch_data_set(pw_patch *patch, void *ud)
{
    patch->ud = ud;
}

void *pw_patch_data_get(pw_patch *patch)
{
    return patch->ud;
}
```

303. For holding buffers via cables, the functions *pw_patch_holdbuf* and *pw_patch_unholdbuf* can be used. These helper functions will access and hold the buffer inside of a patchwerk cable.

\langle Header 6 $\rangle + \equiv$

```
void pw_patch_holdbuf(pw_patch *patch, pw_cable *c);
void pw_patch_unholdbuf(pw_patch *patch, pw_cable *c);
```

304.

After the buffer is held, the *last_free* variable in the bufferpool is reset with *pw_bufferpool_clear_last_free*, which forces the bufferpool to query for a free buffer instead of using the cached value. This ensures that the buffer being held doesn't accidentally get reclaimed.

\langle Patch Top 264 $\rangle + \equiv$

```
void pw_patch_holdbuf(pw_patch *patch, pw_cable *c)
{
    pw_bufferpool *pool;
    pw_buffer *buf;
    if (pw_cable_is_constant(c)) return;
    pool = pw_patch_pool(patch);
    buf = pw_cable_get_buffer(c->pcable);
    pw_bufferpool_holdu(pool, buf);
    pw_bufferpool_clear_last_free(pool);
}

void pw_patch_unholdbuf(pw_patch *patch, pw_cable *c)
{
    pw_bufferpool *pool;
    pw_buffer *buf;
    if (pw_cable_is_constant(c)) return;
    pool = pw_patch_pool(patch);
    buf = pw_cable_get_buffer(c->pcable);
    pw_bufferpool_unholdu(pool, buf);
}
```

305. The functions *pw_patch_bhold* and *pw_patch_bunhold* patchwerk C functions built to somewhat mimic the behaviors of the runt words *bhold* and *bunhold*.

The *bhold* word will pop the last buffer off the buffer stack and hold it. The output buffer will be stored in the variable *b*, assuming it is not a null value.

The *bunhold* word will take in a held buffer *b* to be unheld.

Both functions will return `PW_OK` on success, and an error code on failure (`PW_INVALID_BUFFER` will be used on null buffers, and `PW_NOT_OK` for everything else).

⟨Header 6⟩ +≡

```
int pw_patch_bhold(pw_patch *patch, pw_buffer **b);
int pw_patch_bunhold(pw_patch *patch, pw_buffer *b);
```

306. It should be noted that in *pw_patch_bhold*, *pw_stack_pop* makes the buffer available to be reassigned. This is cleared with *pw_bufferpool_clear_last_free*.

⟨Patch Top 264⟩ +≡

```
int pw_patch_bhold(pw_patch *patch, pw_buffer **b)
{
    int rc;
    pw_stack *stack;
    pw_buffer *buf;
    pw_bufferpool *pool;
    stack = pw_patch_stack(patch);
    pool = pw_patch_pool(patch);
    rc = pw_stack_pop(stack, &buf);
    if (rc != PW_OK) return rc;
    pw_bufferpool_clear_last_free(pool);
    pw_bufferpool_holdu(pool, buf);
    if (b != Λ) *b = buf;
    return PW_OK;
}

int pw_patch_bunhold(pw_patch *patch, pw_buffer *b)
{
    pw_bufferpool *pool;
    int rc;
    pool = pw_patch_pool(patch);
    rc = pw_bufferpool_unholdu(pool, b);
    if (rc != PW_OK) return rc;
    return PW_OK;
}
```

307. Subpatch. A subpatch can thought up as an abstraction of a patch. A subpatch is first populated via **pw_patch**, then exported as a subpatch. A subpatch contains the necessary state information to compute and destroy a self-contained audio graph. Subpatches were created to support on-the-fly hotswapping for live coding.

⟨ Top 8 ⟩ +≡
⟨ Subpatch Top 313 ⟩

308. Data.

309. The data contents of a *pw_subpatch* are contained in a **typedef** inside of the \langle Header 6 \rangle . The contents of this patch are the core elements needed to represent an audio graph created without the need of **pw_patch**:

- The variables *nodes* and *last* refer to the root and last item of the linked list.
- The number of nodes is contained in the variable *nnodes*.
- The output cable *out* is the output of the audio graph.
- The pointer list *plist*, containing data allocated inside of the patch.

\langle Type Declarations 56 $\rangle + \equiv$

```
typedef struct {  
    pw_node *nodes;  
    pw_node *last;  
    int nnodes;  
    pw_cable *out;  
    pw_pointerlist plist;  
} pw_subpatch;
```

310. Functions.

311. The function *pw_subpatch_init* initializes a subpatch by setting the number of nodes *nnodes* to 0.

312. \langle Header 6 $\rangle + \equiv$
void *pw_subpatch_init*(**pw_subpatch** **subpatch*);

313. \langle Subpatch Top 313 $\rangle \equiv$
void *pw_subpatch_init*(**pw_subpatch** **subpatch*)
{
 subpatch→*nnodes* = 0;
 pw_pointerlist_init(&*subpatch*→*plist*);
}

See also sections 316, 319, 321, 323, 325, and 327.

This code is used in section 307.

314. The function *pw_subpatch_save* copies variables from **pw_patch** to a **pw_subpatch** variable, thus producing a snapshot of the **pw_patch** state. In practice, it is best to call *pw_patch_clear* or *pw_patch_reinit* after saving a patch, as it resets the values in the **pw_patch** instance.

315. \langle Header 6 $\rangle + \equiv$
void *pw_subpatch_save*(**pw_patch** **patch*, **pw_subpatch** **subpatch*);

316. \langle Subpatch Top 313 $\rangle + \equiv$
void *pw_subpatch_save*(**pw_patch** **patch*, **pw_subpatch** **subpatch*)
{
 subpatch→*nodes* = *patch*→*nodes*;
 subpatch→*last* = *patch*→*last*;
 subpatch→*nnodes* = *patch*→*nnodes*;
 subpatch→*out* = *patch*→*out*;
 subpatch→*plist* = *patch*→*plist*;
}

317. The function *pw_subpatch_restore* copies variables from a **pw_subpatch** instance to a **pw_patch** instance.

318. \langle Header 6 $\rangle + \equiv$
void *pw_subpatch_restore*(**pw_patch** **patch*, **pw_subpatch** **subpatch*);

319. \langle Subpatch Top 313 $\rangle + \equiv$
void *pw_subpatch_restore*(**pw_patch** **patch*, **pw_subpatch** **subpatch*)
{
 patch→*nodes* = *subpatch*→*nodes*;
 patch→*last* = *subpatch*→*last*;
 patch→*nnodes* = *subpatch*→*nnodes*;
 patch→*out* = *subpatch*→*out*;
 patch→*plist* = *subpatch*→*plist*;
}

320. The function *pw_subpatch_compute* computes a sample block for a patch contained inside of a subpatch.

⟨Header 6⟩ +≡
void *pw_subpatch_compute*(**pw_subpatch** **subpatch*);

321. ⟨Subpatch Top 313⟩ +≡
void *pw_subpatch_compute*(**pw_subpatch** **subpatch*)
{
 int *n*;
 pw_node **node*;
 pw_node **next*;
 node = *subpatch*→*nodes*;
 for (*n* = 0; *n* < *subpatch*→*nnodes*; *n*++) {
 next = *pw_node_get_next*(*node*);
 pw_node_compute(*node*);
 node = *next*;
 }
}

322. The function *pw_subpatch_destroy* calls the destroy function inside of each node contained in the **pw_subpatch**.

⟨Header 6⟩ +≡
void *pw_subpatch_destroy*(**pw_subpatch** **subpatch*);

323. ⟨Subpatch Top 313⟩ +≡
void *pw_subpatch_destroy*(**pw_subpatch** **subpatch*)
{
 int *n*;
 pw_node **node*;
 pw_node **next*;
 node = *subpatch*→*nodes*;
 for (*n* = 0; *n* < *subpatch*→*nnodes*; *n*++) {
 next = *pw_node_get_next*(*node*);
 pw_node_destroy(*node*);
 node = *next*;
 }
}

324. The function *pw_subpatch_free* frees the nodes inside of a subpatch. In addition, this function also frees and re-initializes the internal pointer list.

⟨Header 6⟩ +≡
void *pw_subpatch_free*(**pw_subpatch** **subpatch*);

325. \langle Subpatch Top 313 $\rangle + \equiv$

```
void pw_subpatch_free(pw_subpatch *subpatch)
{
    int n;
    pw_node *node;
    pw_node *next;
    node = subpatch->nodelist;
    for (n = 0; n < subpatch->nnodes; n++) {
        next = pw_node_get_next(node);
        free(node);
        node = next;
    }
    subpatch->nnodes = 0;
    pw_pointerlist_free(&subpatch->plist);
    pw_pointerlist_init(&subpatch->plist);
}
```

326. The function *pw_subpatch_out* returns the output cable in the *pw_subpatch_out*.

\langle Header 6 $\rangle + \equiv$

```
pw_cable *pw_subpatch_out(pw_subpatch *subpatch);
```

327. \langle Subpatch Top 313 $\rangle + \equiv$

```
pw_cable *pw_subpatch_out(pw_subpatch *subpatch)
{
    return subpatch->out;
}
```


328. Event Graphs. Event graphs are a complimentary data structure to accompany the main patchwerk audio graph. While audio graphs represent the signal flow of a given patch, event graphs represent discrete events that happen during the duration of the patch.

⟨ Top [8](#) ⟩ +≡
⟨ Event Graph Top [332](#) ⟩

329. A Single Event Node. An event graph is composed of interconnected nodes called event nodes, otherwise known as a *pw_evnode*.

⟨Type Declarations 56⟩ +≡

```
typedef struct pw_evnode pw_evnode;
```

330. The **pw_evnode** is centered around a callback which is the event associated with the node. The callback is a **typedef** called *pw_evnode_cb*.

⟨Header 6⟩ +≡

```
typedef pw_evnode *(*pw_evnode_cb)(pw_evnode *, int);
```

331. The default **pw_evnode_cb** returns the next entry in the linked list. However, this does not always have to be the case. Other event nodes, such as ones stored in userdata, also be returned.

⟨An Empty Event Callback 331⟩ ≡

```
static pw_evnode *empty_event(pw_evnode *evn, int pos)
{
    return pw_evnode_next(evn);
}
```

This code is used in section 337.

332. The **pw_evnode** contains the following data:

- A callback for the actual event.
- An optional callback for cleanup.
- The duration of the event, in ticks.
- A void pointer for any user data.
- A prev and next entry, needed for bidirectional linked list

⟨Event Graph Top 332⟩ ≡

```
struct pw_evnode {
    pw_evnode_cb event;
    void(*clean)(pw_evnode *);
    int dur;
    void *ud;
    pw_evnode *prev;
    pw_evnode *next;
};
```

See also sections 335, 337, 340, 342, 344, 346, 348, 350, 352, 354, 356, 358, and 359.

This code is used in section 328.

333. Event Node Functions.

334. To make room for future memory allocation options in patchwerk, top level patchwerk functions *pw_evnode_new* and *pw_evnode_bye* have been created. These functions take in a **pw_patch** as an argument, and are designed to have clearer error checking.

⟨Header 6⟩ +≡

```
int pw_evnode_new(pw_patch *patch, pw_evnode **evn);
int pw_evnode_bye(pw_patch *patch, pw_evnode **evn);
```

335. These functions use the internal memory allocation routines inside of the main **pw_patch** instance.

⟨Event Graph Top 332⟩ +≡

```
int pw_evnode_new(pw_patch *patch, pw_evnode **evn)
{
    pw_evnode *e;
    int rc;
    e = Λ;
    rc = pw_memory_alloc(patch, sizeof(pw_evnode), (void **) &e);
    if (e ≠ Λ) {
        pw_evnode_init(e);
        *evn = e;
    }
    return rc;
}
int pw_evnode_bye(pw_patch *patch, pw_evnode **evn)
{
    return pw_memory_free(patch, (void **) evn);
}
```

336. Once a **pw_evnode** has been allocated, it can be initialized with *pw_evnode_init*.

⟨Header 6⟩ +≡

```
void pw_evnode_init(pw_evnode *evn);
```

337. ⟨Event Graph Top 332⟩ +≡

⟨An Empty Event Callback 331⟩

⟨An Empty Cleanup Callback 338⟩

```
void pw_evnode_init(pw_evnode *evn)
{
    evn-dur = 0;
    evn-ud = Λ;
    evn-event = empty_event;
    evn-clean = empty_clean;
    evn-prev = Λ;
    evn-next = Λ;
}
```

338. Like *empty_event*, *empty_clean* supplies a default empty callback for cleanup.

⟨An Empty Cleanup Callback 338⟩ ≡

```
static void empty_clean(pw_evnode *evn)
{ }
```

This code is used in section 337.

339. The callback inside of node can be called with the function *pw_evnode_fire*.

⟨Header 6⟩ +≡

```
pw_evnode *pw_evnode_fire(pw_evnode *evn, int pos);
```

340. ⟨Event Graph Top 332⟩ +≡

```
pw_evnode *pw_evnode_fire(pw_evnode *evn, int pos)
{
    return evn->event(evn, pos);
}
```

341. The event callback can be set using the function *pw_evnode_callback*

⟨Header 6⟩ +≡

```
void pw_evnode_callback(pw_evnode *evn, pw_evnode_cb cb);
```

342. ⟨Event Graph Top 332⟩ +≡

```
void pw_evnode_callback(pw_evnode *evn, pw_evnode_cb cb)
{
    evn->event = cb;
}
```

343. The clean callback inside of node can be called with the function *pw_evnode_clean*.

⟨Header 6⟩ +≡

```
void pw_evnode_clean(pw_evnode *evn);
```

344. ⟨Event Graph Top 332⟩ +≡

```
void pw_evnode_clean(pw_evnode *evn)
{
    evn->clean(evn);
}
```

345. The clean callback can be set using the function *pw_evnode_clean_set*

⟨Header 6⟩ +≡

```
void pw_evnode_clean_set(pw_evnode *evn, void(*cb)(pw_evnode *));
```

346. ⟨Event Graph Top 332⟩ +≡

```
void pw_evnode_clean_set(pw_evnode *evn, void(*cb)(pw_evnode *))
{
    evn->clean = cb;
}
```

347. User data can be associated to an event node using setter/getter functions *pw_evnode_data_set* and *pw_evnode_data_get*, respectively.

⟨Header 6⟩ +≡

```
void pw_evnode_data_set(pw_evnode *evn, void *ud);
void *pw_evnode_data_get(pw_evnode *evn);
```

348. User data is contained as a generic **void** * pointer. If data is to be managed by an instance of an event node, a cleanup function should be provided via *pw_evnode_clean_set*.

Alternatively, one could indirectly have memory handled by using the **pw_pointer** and **pw_pointerlist** interfaces attached with a **pw_patch** or **pw_subpatch**. See the pointer section in [\(Pointer Top 118\)](#) for more information.

```

< Event Graph Top 332 > +≡
    void pw_evnode_data_set(pw_evnode *evn, void *ud)
    {
        evn->ud = ud;
    }
    void *pw_evnode_data_get(pw_evnode *evn)
    {
        return evn->ud;
    }

```

349. The next event node in the list (if it exists) is returned using the function *pw_evnode_next*.

```

< Header 6 > +≡
    pw_evnode *pw_evnode_next(pw_evnode *evn);

```

350. Note that *pw_evnode_next* does not check for null values.

```

< Event Graph Top 332 > +≡
    pw_evnode *pw_evnode_next(pw_evnode *evn)
    {
        return evn->next;
    }

```

351. The function *pw_evnode_dur* sets the duration.

```

< Header 6 > +≡
    void pw_evnode_dur(pw_evnode *evn, int dur);

```

```

352. < Event Graph Top 332 > +≡
    void pw_evnode_dur(pw_evnode *evn, int dur)
    {
        evn->dur = dur;
    }

```

353. The function *pw_evnode_dur_get* returns the duration.

```

< Header 6 > +≡
    int pw_evnode_dur_get(pw_evnode *evn);

```

```

354. < Event Graph Top 332 > +≡
    int pw_evnode_dur_get(pw_evnode *evn)
    {
        return evn->dur;
    }

```

355. The function *pw_evnode_is_terminal* checks if an event node is a terminal node.

```

< Header 6 > +≡
    int pw_evnode_is_terminal(pw_evnode *evn);

```

356. A node is terminal if the *next* entry is null.

```

⟨Event Graph Top 332⟩ +≡
  int pw_evnode_is_terminal(pw_evnode *evn)
  {
    return evn→next ≡ Λ;
  }

```

357. TODO: Words, please.

```

⟨Header 6⟩ +≡
  void pw_evnode_set_next(pw_evnode *evn, pw_evnode *next);

```

358. TODO: Words, please.

```

⟨Event Graph Top 332⟩ +≡
  void pw_evnode_set_next(pw_evnode *evn, pw_evnode *next)
  {
    evn→next = next;
  }

```

359. The Event Walker. An event walker is used to step through the event graph, starting at any node. Each event walker contains internal state information about the current location in the graph. Event walkers are designed such that multiple walkers can occur at the same time on the same graph.

$\langle \text{Event Graph Top } 332 \rangle + \equiv$
 $\langle \text{Event Walker Top } 364 \rangle$

360. Event Walker Data. The event walker is contained in a typedef struct called *pw_evwalker*. The type declaration is contained inside the header file, so dynamic memory allocation is not required.

⟨Type Declarations 56⟩ +≡

```
typedef struct {  
    ⟨Struct Contents in Event Walker 361⟩  
} pw_evwalker;
```

361. The data in the **pw_evwalker** struct contains all the necessary state information to locally know and keep track of the current position in the graph.

These elements include:

- The current event node.
- A counter, to keep track of duration.
- A boolean flag to indicate if it has reached a terminal node.

⟨Struct Contents in Event Walker 361⟩ ≡

```
pw_evnode *node;  
int count;  
int terminal;
```

This code is used in section 360.

362. Event Walker Functions.

363. The function *pw_evwalker_reset* zeros out the event walker data contained inside of **pw_evwalker**. This will overwrite any previous data.

⟨Header 6⟩ +≡

```
void pw_evwalker_reset(pw_evwalker *walker);
```

364. Note that there are no calls to malloc required, making it safe to call more than once.

⟨Event Walker Top 364⟩ ≡

```
void pw_evwalker_reset(pw_evwalker *walker)
{
    walker->node = Λ;
    walker->count = 0;
    walker->terminal = 1;    /* set terminal flag on by default */
}
```

See also sections 366 and 368.

This code is used in section 359.

365. A walker can be initialized with a starting event node using the function *pw_evwalker_init*. This function sets up the walker with a starting event node. If the starting event node argument is NULL, it initializes an empty event walker. This function does not handle any memory allocation, so it is safe to call again for reinitialization.

⟨Header 6⟩ +≡

```
void pw_evwalker_init(pw_evwalker *walker, pw_evnode *start, int delay);
```

366. The task of initializing an event walker can be broken down into the following subtasks:

- Set the initial node to be the starter node.
- Set the counter duration to be initial delay. If the initial delay is zero, the first node function will be called immediately. A delay of one will wait one step before firing, a delay of two steps will wait two steps before firing, etc.
- Set the terminal flag to the offstate.
- If the starting node is NULL, then it is an empty event walker. The terminal flag is flipped on.

⟨Event Walker Top 364⟩ +≡

```
void pw_evwalker_init(pw_evwalker *walker, pw_evnode *start, int delay)
{
    walker->count = delay;
    walker->node = start;
    walker->terminal = 0;
    if (start ≡ Λ) {
        walker->terminal = 1;
    }
}
```

367. Once an event walker has been initialized with *pw_evwalker_init*, it can begin stepping through the event graph with the function *pw_evwalker_walk*. This will return 0 if it has reached a terminal node. Otherwise, it will return a 1.

The argument *pos* is for situations where sample accuracy is desired. This refers to the current sample position in the audio render loop. A value zero is fine if you want to ignore this.

⟨Header 6⟩ +≡

```
int pw_evwalker_walk(pw_evwalker *walker, int pos);
```

368. The Walker Algorithm can be broken down into the following steps:

- First, the walker checks for the terminal flag. If it has been set, it immediately returns 0. This happens first so that no null reads occur.
- After the terminal flag check, there is a counter a check. A counter greater than zero will result in the counter variable decreasing by one and then exiting.
- When the counter is zero, it indicates that it is time to jump to the next event node.
- The current event node is checked for being terminal. If it is, the terminal flag is set. The return code is set to be zero.
- The counter is set to be the duration the of current event node. This can be thought of as sort of delay until the next node in the graph.
- Finally, the event function of the current procedure is called. This function replaces the current event node in the walker.
- While nodes are zero and non-terminal, step through and fire off nodes in the graph. This particular feature allows for multiple nodes to be fired in a single step.
- An event node that returns NULL is said to be treated the same way as a terminal event node.
- The end of the function returns a boolean check on if the terminal flag is set or not.
- When a new non-zero counter event has occurred, time has passed for this note, and thus the counter must be decreased by one when it breaks out of the loop.

(Event Walker Top 364) +=

```

int pw_evwalker_walk(pw_evwalker *walker, int pos)
{
    if (walker->terminal) return 0;
    if (walker->count > 0) {
        walker->count --;
        return 1;
    }
    else if (walker->count == 0) {
        while (walker->count == 0) {
            if (pw_evnode_is_terminal(walker->node)) {
                pw_evnode_fire(walker->node, pos);
                walker->terminal = 1;
                return 0;
            }
            else {
                walker->count = walker->node->dur;
                walker->node = pw_evnode_fire(walker->node, pos);
                if (walker->node ==  $\Lambda$ ) {
                    walker->terminal = 1;
                    return 0;
                }
            }
        }
    }
    walker->count --;
    return !walker->terminal;
}

```

369. Data Dumping. When debugging Patchwerk, it can be helpful to probe the inner contents. This issue of needing to see information has come up enough times in production to warrant a formal interface for printing contents.

The following functions can be used to print contents to file or standard output. The data format is JSON, which can be scanned or prettified using external utilities like jq or Python.

⟨ Top 8 ⟩ +≡
 ⟨ Data Dumpers 371 ⟩

370. The function *pw_dump_cable* dumps the contents of a cable.

⟨ Header 6 ⟩ +≡
void *pw_dump_cable*(**FILE** **fp*, **pw_cable** **c*);

371. ⟨ Data Dumpers 371 ⟩ ≡
void *pw_dump_cable*(**FILE** **fp*, **pw_cable** **c*)
 {
 fprintf(*fp*, "{");
 fprintf(*fp*, "\"type\":");
 if (*c*-type ≡ CABLE_BLOCK) {
 fprintf(*fp*, "\"block\",");
 }
 else if (*c*-type ≡ CABLE_IVAL) {
 fprintf(*fp*, "\"constant\",");
 }
 fprintf(*fp*, "\"node_id\":");
 if (*c*-node ≡ Λ) {
 fprintf(*fp*, "-1,");
 }
 else {
 fprintf(*fp*, "%d,", *pw_node_get_id*(*c*-node));
 }
 fprintf(*fp*, "\"ival\":%g,", *c*-ival);
 fprintf(*fp*, "\"buffer\":");
 if (*c*-buf ≠ Λ) {
 fprintf(*fp*, "%d,", *pw_buffer_id*(*c*-buf));
 }
 else {
 fprintf(*fp*, "-1,");
 }
 fprintf(*fp*, "\"pcable\":");
 if (*c*-pcable ≠ *c*) {
 pw_dump_cable(*fp*, *c*-pcable);
 }
 else {
 fprintf(*fp*, "{}");
 }
 fprintf(*fp*, "}");
 }

See also sections 373, 375, 377, 379, 381, 383, and 385.

This code is used in section 369.

372. The function *pw_dump_node* dumps the contents of a node.

⟨Header 6⟩ +≡

```
void pw_dump_node(FILE *fp, pw_node *n, int print_cables);
```

373. ⟨Data Dumpers 371⟩ +≡

```
void pw_dump_node(FILE *fp, pw_node *n, int print_cables)
{
    int c;
    fprintf(fp, "{");
    fprintf(fp, "\"id\":%d,", n->id);
    fprintf(fp, "\"ncables\":%d,", n->ncables);
    fprintf(fp, "\"type\":%d", n->type);
    if (print_cables) {
        fprintf(fp, ",");
        fprintf(fp, "\"cables\": [");
        for (c = 0; c < n->ncables; c++) {
            pw_dump_cable(fp, &n->cables[c]);
            if (c ≠ n->ncables - 1) {
                fprintf(fp, ",");
            }
        }
        fprintf(fp, "]");
    }
    fprintf(fp, "}");
}
```

374. The function *pw_dump_nodes* is a function that dumps a linked list of nodes. It is used inside *pw_dump_nodelist* and *pw_dump_subpatch*. It normally shouldn't be called directly.

⟨Header 6⟩ +≡

```
void pw_dump_nodes(FILE *fp, pw_node *nodes, int nnodes, int print_cables);
```

375. \langle Data Dumpers 371 $\rangle + \equiv$

```
void pw_dump_nodes(FILE *fp, pw_node *nodes, int nnodes, int print_cables)
{
    int n;
    pw_node *nd;
    pw_node *nxt;
    nd = nodes;
    fprintf(fp, "{");
    fprintf(fp, "\"nnodes\":%d", nnodes);
    fprintf(fp, "\"nodes\":[");
    for (n = 0; n < nnodes; n++) {
        nxt = nd->next;
        pw_dump_node(fp, nd, print_cables);
        if (n != nnodes - 1) {
            fprintf(fp, ",");
        }
        nd = nxt;
    }
    fprintf(fp, "]");
    fprintf(fp, "}");
}
```

376. The function *pw_dump_nodelist* dumps the list of nodes in the current patch.

\langle Header 6 $\rangle + \equiv$

```
void pw_dump_nodelist(FILE *fp, pw_patch *p, int print_cables);
```

377. \langle Data Dumpers 371 $\rangle + \equiv$

```
void pw_dump_nodelist(FILE *fp, pw_patch *p, int print_cables)
{
    pw_dump_nodes(fp, p->nodes, p->nnodes, print_cables);
}
```

378. The function *pw_dump_subpatch* dumps the contents of a subpatch.

\langle Header 6 $\rangle + \equiv$

```
void pw_dump_subpatch(FILE *fp, pw_subpatch *s, int print_cables);
```

379. \langle Data Dumpers 371 $\rangle + \equiv$

```
void pw_dump_subpatch(FILE *fp, pw_subpatch *s, int print_cables)
{
    pw_dump_nodes(fp, s->nodes, s->nnodes, print_cables);
}
```

380. The function *pw_dump_bufferpool* dumps the contents of the bufferpool.

\langle Header 6 $\rangle + \equiv$

```
void pw_dump_bufferpool(FILE *fp, pw_bufferpool *bp, int print_buffers);
```

381. `<Data Dumpers 371> +≡`

```
void pw_dump_bufferpool(FILE *fp, pw_bufferpool *bp, int print_buffers)
{
    int b;
    int pos;

    fprintf(fp, "{");
    fprintf(fp, "\"size\":%d", bp->size);
    fprintf(fp, "\"nactive\":%d", bp->nactive);
    fprintf(fp, "\"user_buffers\":%d", pw_bufferpool_uactive(bp));
    fprintf(fp, "\"last_free\":%d", bp->last_free);
    if (print_buffers) {
        fprintf(fp, "\",\"buffers\":[");
        pos = 0;
        for (b = 0; b < bp->size; b++) {
            if (bp->buffers[b].read != 0) {
                pw_dump_buffer(fp, &bp->buffers[b]);
                pos++;
                if (pos != bp->nactive) fprintf(fp, ",");
            }
        }
        fprintf(fp, "]");
    }
    fprintf(fp, "}");
}
```

382. The function `pw_dump_stack` dumps the contents of the buffer stack.

`<Header 6> +≡`

```
void pw_dump_stack(FILE *fp, pw_stack *s, int print_buffers);
```

383. `<Data Dumpers 371> +≡`

```
void pw_dump_stack(FILE *fp, pw_stack *s, int print_buffers)
{
    int b;

    fprintf(fp, "{");
    fprintf(fp, "\"size\":%d", s->size);
    fprintf(fp, "\"pos\":%d", s->pos);
    if (print_buffers) {
        fprintf(fp, "\",\"buffers\":[");
        for (b = 0; b < s->pos; b++) {
            pw_dump_buffer(fp, s->buffers[b]);
            if (b != s->pos - 1) {
                fprintf(fp, ",");
            }
        }
        fprintf(fp, "]");
    }
    fprintf(fp, "}");
}
```

384. The function *pw_dump_buffer* dumps the contents of a buffer.

⟨Header 6⟩ +≡

```
void pw_dump_buffer(FILE *fp, pw_buffer *b);
```

385. ⟨Data Dumpers 371⟩ +≡

```
void pw_dump_buffer(FILE *fp, pw_buffer *b)
{
    fprintf(fp, "{");
    fprintf(fp, "\"id\":\t%d", b->id);
    fprintf(fp, "\"read\":\t%d", b->read);
    fprintf(fp, "}");
}
```

386. Memory Handling.

Many parts of Patchwerk require dynamically allocated memory. When porting the Patchwerk API to other languages and systems, memory allocation schemes may need to change. For example, bindings to a scripting language like Lua may want to take advantage of that garbage collector.

This section describes the memory allocation system process.

⟨Top 8⟩ +≡
 ⟨Memory Handling 388⟩

387. The memory handling functions in Patchwerk are *pw_memory_alloc* and *pw_memory_free* for allocating and freeing memory, respectively.

The function *pw_memory_alloc* will allocate a chunk of memory that is *size* bytes and store it in the the pointer address in *ud*.

The function *pw_memory_free* will free a chunk of memory in *ud* previously allocated by *pw_memory_alloc*.

⟨Header 6⟩ +≡
int *pw_memory_alloc*(**pw_patch** **p*, **size_t** *size*, **void** ***ud*);
int *pw_memory_free*(**pw_patch** **p*, **void** ***ud*);

388.

⟨Memory Handling 388⟩ ≡
int *pw_memory_alloc*(**pw_patch** **p*, **size_t** *size*, **void** ***ud*)
 {
void **ptr*;
ptr = *malloc*(*size*);
if (*ptr* ≡ Λ) **return** PW_NOT_OK;
 **ud* = *ptr*;
return PW_OK;
 }
int *pw_memory_free*(**pw_patch** **p*, **void** ***ud*)
 {
free(**ud*);
return PW_OK;
 }

See also sections 392 and 394.

This code is used in section 386.

389. Internally, these functions call on internal function pointers which are stored in the top-level **pw_patch** struct.

⟨Variables in Patch Data 252⟩ +≡
pw_mallocfun *malloc*;
pw_freefun *free*;

390. These function callbacks mirror *pw_memory_alloc* and *pw_memory_free*.

⟨Type Declarations 56⟩ +≡
typedef int (**pw_mallocfun*)(**pw_patch** *, **size_t**, **void** **);
typedef int (**pw_freefun*)(**pw_patch** *, **void** **);

391. These two functions can be overridden by the user at init-time. With the function *pw_memory_override*.

⟨Header 6⟩ +≡
void *pw_memory_override*(**pw_patch** **p*, *pw_mallocfun* *m*, *pw_freefun* *f*);

392.

⟨Memory Handling 388⟩ +≡

```
void pw_memory_override(pw_patch *p, pw_mallocfun m, pw_freefun f)
{
    p->malloc = m;
    p->free = f;
}
```

393. If left alone, the default memory allocation functions are wrappers around *free* and *malloc*. This can also be set with the function *pw_memory_defaults*.

⟨Header 6⟩ +≡

```
void pw_memory_defaults(pw_patch *p);
```

394.

⟨Memory Handling 388⟩ +≡

```
static int default_malloc(pw_patch *p, size_t size, void **ud)
{
    void *ptr;
    ptr = malloc(size);
    if (ptr ==  $\Lambda$ ) return PW_NOT_OK;
    *ud = ptr;
    return PW_OK;
}

static int default_free(pw_patch *p, void **ud)
{
    free(*ud);
    return PW_OK;
}

void pw_memory_defaults(pw_patch *p)
{
    pw_memory_override(p, default_malloc, default_free);
}
```

395. Printing (WIP).

Printing messages to the screen is particularly useful tool for troubleshooting. An internal printing system has been designed to accomodate different platforms. This is done by having an internal printing callback that can be overridden with a user-supplied callback function.

Most of this functionality has been borrowed from the Runt system.

396. The core print callback is called *pw_print*. It is designed to function similarly to *printf* with a **pw_patch** argument.

⟨Header 6⟩ +≡
void *pw_print*(**pw_patch** **p*, **const char** **fmt*, ...);

397. Variadic macro lists are used to set up the internally stored print callback.

⟨Patch Top 264⟩ +≡
void *pw_print*(**pw_patch** **p*, **const char** **fmt*, ...)
 {
 va_list *args*;
 va_start(*args*, *fmt*);
 p-print(*p*, *fmt*, *args*);
 va_end(*args*);
 }

398. Stored inside of the **pw_patch** struct is the print function pointer.

⟨Variables in Patch Data 252⟩ +≡
void(**print*)(**pw_patch** *, **const char** **fmt*, **va_list**);

399. By default, it is set to use *pw_print_default*, via the function *pw_print_init*.

⟨Header 6⟩ +≡
void *pw_print_init*(**pw_patch** **p*);

400.

⟨Patch Top 264⟩ +≡
 ⟨Default Print Function 401⟩
void *pw_print_init*(**pw_patch** **p*)
 {
 p-print = *pw_print_default*;
 }

401. The function *pw_print_default* uses *vprintf*, to get the equivalent functionality of *printf* with variadic arguments.

⟨Default Print Function 401⟩ ≡
static void *pw_print_default*(**pw_patch** **p*, **const char** **str*, **va_list** *args*)
 {
 vprintf(*str*, *args*);
 }

This code is used in section 400.

402. The default print function can be overridden after patchwerk is initialized using the function *pw_print_set*. ■

⟨Header 6⟩ +≡
void *pw_print_set*(**pw_patch** **p*, **void**(**print*)(**pw_patch** *, **const char** *, **va_list**));

403.

⟨Patch Top 264⟩ +=

```
void pw_print_set(pw_patch *p, void(*print)(pw_patch *, const char *, va_list))
{
    p-print = print;
}
```

args: [397](#), [401](#).

b: [305](#), [306](#), [381](#), [383](#), [384](#), [385](#).

blk: [59](#), [66](#), [69](#), [70](#), [93](#).

blksize: [17](#), [24](#), [25](#), [43](#), [45](#), [59](#), [66](#), [68](#), [69](#), [70](#), [93](#),
[94](#), [95](#), [97](#), [99](#), [103](#), [106](#), [188](#), [189](#), [254](#), [263](#),
[264](#), [266](#), [267](#), [268](#), [285](#), [287](#), [296](#).

bp: [380](#), [381](#).

buf: [63](#), [66](#), [91](#), [92](#), [93](#), [154](#), [163](#), [164](#), [166](#), [167](#),
[168](#), [169](#), [170](#), [171](#), [172](#), [173](#), [174](#), [175](#), [176](#),
[177](#), [178](#), [179](#), [180](#), [181](#), [182](#), [183](#), [184](#), [185](#),
[196](#), [197](#), [198](#), [199](#), [202](#), [203](#), [204](#), [207](#), [208](#),
[209](#), [210](#), [228](#), [229](#), [230](#), [231](#), [232](#), [233](#), [234](#),
[240](#), [241](#), [304](#), [306](#), [371](#).

buffers: [156](#), [188](#), [190](#), [192](#), [202](#), [203](#), [211](#), [219](#),
[224](#), [226](#), [228](#), [230](#), [232](#), [234](#), [238](#), [381](#), [383](#).

c: [73](#), [74](#), [289](#), [303](#), [304](#), [370](#), [371](#), [373](#).

cab: [83](#), [84](#), [85](#), [86](#), [87](#), [88](#), [89](#), [90](#), [91](#), [92](#), [95](#), [96](#).

cable: [40](#), [41](#), [65](#), [66](#), [67](#), [68](#), [69](#), [70](#), [71](#), [72](#), [75](#), [76](#),
[77](#), [78](#), [93](#), [94](#), [99](#), [100](#), [277](#), [278](#), [288](#), [289](#).

CABLE_BLOCK: [61](#), [69](#), [87](#), [371](#).

CABLE_IVAL: [61](#), [66](#), [72](#), [75](#), [76](#), [78](#), [89](#), [371](#).

CABLE_OVERRIDE: [61](#).

cables: [16](#), [37](#), [39](#), [41](#), [45](#), [373](#).

cb: [341](#), [342](#), [345](#), [346](#).

clean: [332](#), [337](#), [344](#), [346](#).

compute: [14](#), [25](#), [31](#), [33](#).

count: [361](#), [364](#), [366](#), [368](#).

counter: [255](#), [270](#), [285](#).

c1: [79](#), [80](#), [81](#), [82](#), [101](#), [102](#), [103](#), [105](#), [106](#).

c2: [79](#), [80](#), [81](#), [82](#), [101](#), [102](#), [103](#), [105](#), [106](#).

data: [34](#), [35](#).

default_free: [394](#).

default_malloc: [394](#).

delay: [365](#), [366](#).

delete_cable: [289](#).

destroy: [13](#), [25](#), [31](#), [33](#).

dfun: [291](#), [292](#).

dur: [332](#), [337](#), [351](#), [352](#), [354](#), [368](#).

e: [335](#).

empty: [25](#).

empty_clean: [337](#), [338](#).

empty_event: [331](#), [337](#), [338](#).

errmsg: [110](#), [112](#).

event: [332](#), [337](#), [340](#), [342](#).

evn: [331](#), [334](#), [335](#), [336](#), [337](#), [338](#), [339](#), [340](#), [341](#),

[342](#), [343](#), [344](#), [345](#), [346](#), [347](#), [348](#), [349](#), [350](#), [351](#),
[352](#), [353](#), [354](#), [355](#), [356](#), [357](#), [358](#).

fmt: [396](#), [397](#), [398](#).

fp: [370](#), [371](#), [372](#), [373](#), [374](#), [375](#), [376](#), [377](#), [378](#),
[379](#), [380](#), [381](#), [382](#), [383](#), [384](#), [385](#).

fprintf: [371](#), [373](#), [375](#), [381](#), [383](#), [385](#).

free: [121](#), [128](#), [129](#), [131](#), [274](#), [325](#), [388](#), [389](#),
[392](#), [393](#), [394](#).

free_cables: [25](#).

fun: [30](#), [31](#).

group: [19](#), [25](#).

i: [95](#), [97](#), [148](#), [188](#), [190](#), [192](#), [199](#), [211](#), [274](#).

id: [11](#), [25](#), [27](#), [28](#), [29](#), [40](#), [41](#), [44](#), [45](#), [134](#), [135](#), [153](#),
[168](#), [174](#), [184](#), [188](#), [196](#), [209](#), [373](#), [385](#).

id1: [80](#).

id2: [80](#).

in: [97](#), [98](#).

int: [390](#).

ival: [58](#), [66](#), [72](#), [371](#).

last: [124](#), [142](#), [146](#), [252](#), [272](#), [287](#), [309](#), [316](#), [319](#).

last_free: [159](#), [190](#), [201](#), [202](#), [215](#), [232](#), [304](#), [381](#).

malloc: [226](#), [388](#), [389](#), [392](#), [393](#), [394](#).

mix: [97](#), [98](#).

n: [37](#), [39](#), [106](#), [276](#), [321](#), [323](#), [325](#), [372](#), [373](#), [375](#).

nactive: [158](#), [186](#), [188](#), [190](#), [194](#), [196](#), [200](#), [204](#),
[207](#), [209](#), [232](#), [381](#).

nbuf: [188](#), [189](#).

nbuffers: [265](#), [266](#), [267](#), [268](#).

ncables: [16](#), [25](#), [36](#), [37](#), [39](#), [41](#), [45](#), [47](#), [373](#).

nd: [375](#).

next: [20](#), [25](#), [52](#), [53](#), [122](#), [129](#), [137](#), [138](#), [139](#),
[148](#), [274](#), [276](#), [321](#), [323](#), [325](#), [332](#), [337](#), [350](#),
[356](#), [357](#), [358](#), [375](#).

nnodes: [252](#), [272](#), [274](#), [276](#), [287](#), [309](#), [311](#), [313](#),
[316](#), [319](#), [321](#), [323](#), [325](#), [374](#), [375](#), [377](#), [379](#).

node: [24](#), [25](#), [26](#), [27](#), [28](#), [29](#), [30](#), [31](#), [32](#), [33](#), [34](#),
[35](#), [36](#), [37](#), [38](#), [39](#), [40](#), [41](#), [42](#), [43](#), [44](#), [45](#), [46](#),
[47](#), [48](#), [49](#), [50](#), [51](#), [52](#), [53](#), [57](#), [65](#), [66](#), [80](#), [83](#),
[85](#), [274](#), [276](#), [286](#), [287](#), [321](#), [323](#), [325](#), [361](#),
[364](#), [366](#), [368](#), [371](#).

nodepos: [256](#), [270](#).

nodes: [252](#), [272](#), [274](#), [276](#), [287](#), [309](#), [316](#), [319](#), [321](#),
[323](#), [325](#), [374](#), [375](#), [377](#), [379](#).

next: [375](#).

out: [253](#), [272](#), [278](#), [280](#), [285](#), [309](#), [316](#), [319](#), [327](#).

p: [117](#), [136](#), [137](#), [138](#), [139](#), [145](#), [146](#), [289](#), [376](#),
[377](#), [387](#), [388](#), [391](#), [392](#), [393](#), [394](#), [396](#), [397](#),
[399](#), [400](#), [401](#), [402](#), [403](#).
patch: [10](#), [37](#), [39](#), [45](#), [50](#), [51](#), [83](#), [85](#), [119](#), [128](#), [129](#),
[131](#), [163](#), [164](#), [166](#), [167](#), [188](#), [189](#), [192](#), [193](#), [224](#),
[225](#), [226](#), [227](#), [263](#), [264](#), [265](#), [266](#), [267](#), [268](#), [269](#),
[270](#), [271](#), [272](#), [273](#), [274](#), [275](#), [276](#), [277](#), [278](#), [279](#),
[280](#), [284](#), [285](#), [286](#), [287](#), [288](#), [289](#), [291](#), [292](#), [293](#),
[294](#), [295](#), [296](#), [297](#), [298](#), [299](#), [300](#), [301](#), [302](#), [303](#),
[304](#), [305](#), [306](#), [315](#), [316](#), [318](#), [319](#), [334](#), [335](#).
patchwerk: [3](#).
PATCHWERK_H: [5](#).
pbuf: [228](#).
pcable: [62](#), [66](#), [103](#), [304](#), [371](#).
plist: [141](#), [142](#), [143](#), [144](#), [145](#), [146](#), [147](#), [148](#), [257](#),
[272](#), [276](#), [292](#), [309](#), [313](#), [316](#), [319](#), [325](#).
pointer: [128](#), [129](#), [130](#), [131](#), [132](#), [133](#), [134](#), [135](#).
pool: [186](#), [187](#), [188](#), [189](#), [190](#), [191](#), [192](#), [193](#), [194](#),
[195](#), [196](#), [197](#), [198](#), [199](#), [200](#), [201](#), [202](#), [203](#),
[204](#), [207](#), [208](#), [209](#), [210](#), [211](#), [212](#), [213](#), [214](#),
[215](#), [216](#), [219](#), [222](#), [223](#), [228](#), [232](#), [258](#), [264](#),
[266](#), [268](#), [270](#), [276](#), [298](#), [304](#), [306](#).
pop_from_stack: [232](#), [240](#).
pos: [75](#), [76](#), [77](#), [78](#), [219](#), [222](#), [228](#), [230](#), [232](#),
[234](#), [236](#), [238](#), [244](#), [248](#), [331](#), [339](#), [340](#), [367](#),
[368](#), [381](#), [383](#).
pptr: [129](#).
prev: [332](#), [337](#).
print: [397](#), [398](#), [400](#), [402](#), [403](#).
print_buffers: [380](#), [381](#), [382](#), [383](#).
print_cables: [372](#), [373](#), [374](#), [375](#), [376](#), [377](#), [378](#), [379](#).
printf: [401](#).
ptr: [292](#), [388](#), [394](#).
PW_ALREADY_ALLOCATED: [109](#).
pw_buffer: [3](#), [63](#), [83](#), [85](#), [91](#), [92](#), [93](#), [151](#), [152](#),
[153](#), [154](#), [156](#), [161](#), [163](#), [164](#), [166](#), [167](#), [168](#),
[169](#), [170](#), [171](#), [172](#), [173](#), [174](#), [175](#), [176](#), [177](#),
[178](#), [179](#), [180](#), [181](#), [182](#), [183](#), [184](#), [185](#), [188](#),
[196](#), [197](#), [198](#), [199](#), [207](#), [208](#), [209](#), [210](#), [219](#),
[224](#), [228](#), [229](#), [230](#), [231](#), [232](#), [233](#), [234](#), [238](#),
[240](#), [241](#), [304](#), [305](#), [306](#), [384](#), [385](#).
pw_buffer_alloc: [163](#), [164](#), [168](#), [188](#).
pw_buffer_data: [93](#), [176](#), [177](#).
pw_buffer_free: [165](#), [166](#), [167](#), [192](#).
pw_buffer_hold: [178](#), [179](#), [240](#).
pw_buffer_holdu: [180](#), [181](#), [207](#).
pw_buffer_id: [184](#), [185](#), [371](#).
pw_buffer_init: [168](#), [169](#), [188](#).
pw_buffer_mark: [172](#), [173](#), [204](#), [234](#).
pw_buffer_reinit: [168](#), [170](#), [171](#), [190](#).
pw_buffer_size: [161](#), [162](#).
pw_buffer_unhold: [182](#), [183](#), [196](#), [209](#), [240](#).

pw_buffer_unmark: [174](#), [175](#), [232](#).
pw_bufferpool: [3](#), [151](#), [152](#), [153](#), [155](#), [186](#), [187](#),
[188](#), [189](#), [190](#), [191](#), [192](#), [193](#), [194](#), [195](#), [196](#),
[197](#), [198](#), [199](#), [207](#), [208](#), [209](#), [210](#), [211](#), [212](#),
[213](#), [214](#), [215](#), [216](#), [219](#), [222](#), [223](#), [258](#), [297](#),
[298](#), [304](#), [306](#), [380](#), [381](#).
pw_bufferpool_clear_last_free: [215](#), [216](#), [304](#), [306](#).
pw_bufferpool_create: [188](#), [189](#), [192](#), [266](#).
pw_bufferpool_destroy: [192](#), [193](#), [268](#), [276](#).
pw_bufferpool_holdu: [207](#), [208](#), [304](#), [306](#).
pw_bufferpool_init: [186](#), [187](#), [264](#).
pw_bufferpool_nactive: [194](#), [195](#).
pw_bufferpool_nextfree: [198](#), [199](#), [228](#).
pw_bufferpool_reset: [190](#), [191](#), [266](#), [270](#).
pw_bufferpool_uactive: [213](#), [214](#), [381](#).
pw_bufferpool_unhold: [196](#), [197](#).
pw_bufferpool_unholdu: [209](#), [210](#), [211](#), [304](#), [306](#).
pw_bufferpool_unholdu_all: [211](#), [212](#).
pw_cable: [3](#), [16](#), [37](#), [40](#), [41](#), [55](#), [56](#), [57](#), [62](#), [65](#),
[66](#), [67](#), [68](#), [69](#), [70](#), [71](#), [72](#), [73](#), [74](#), [75](#), [76](#), [77](#),
[78](#), [79](#), [80](#), [81](#), [82](#), [83](#), [84](#), [85](#), [86](#), [87](#), [88](#), [89](#),
[90](#), [91](#), [92](#), [93](#), [94](#), [95](#), [96](#), [97](#), [98](#), [99](#), [100](#), [102](#),
[103](#), [105](#), [106](#), [253](#), [277](#), [278](#), [279](#), [280](#), [285](#), [288](#),
[289](#), [303](#), [304](#), [309](#), [326](#), [327](#), [370](#), [371](#).
pw_cable_blksize: [99](#), [100](#).
pw_cable_clear: [95](#), [96](#), [103](#).
pw_cable_connect: [79](#), [80](#).
pw_cable_connect_nocheck: [80](#), [81](#), [82](#), [101](#).
pw_cable_copy: [104](#), [105](#), [106](#).
pw_cable_free: [39](#), [67](#), [68](#).
pw_cable_get: [75](#), [76](#), [106](#), [285](#).
pw_cable_get_buffer: [91](#), [92](#), [304](#).
pw_cable_init: [37](#), [65](#), [66](#), [270](#), [289](#).
pw_cable_is_block: [83](#), [87](#), [88](#), [95](#), [97](#).
pw_cable_is_constant: [89](#), [90](#), [304](#).
pw_cable_make_block: [45](#), [93](#), [94](#).
pw_cable_mix: [97](#), [98](#).
pw_cable_override: [82](#), [101](#), [102](#), [103](#).
pw_cable_pop: [83](#), [84](#).
pw_cable_push: [85](#), [86](#).
pw_cable_set: [77](#), [78](#), [106](#).
pw_cable_set_block: [69](#), [70](#), [93](#).
pw_cable_set_buffer: [91](#), [92](#), [93](#).
pw_cable_set_constant: [71](#), [72](#), [73](#).
pw_cable_set_value: [73](#), [74](#).
PW_CONNECTION_MISMATCH: [79](#), [80](#), [109](#).
pw_dump_buffer: [381](#), [383](#), [384](#), [385](#).
pw_dump_bufferpool: [380](#), [381](#).
pw_dump_cable: [370](#), [371](#), [373](#).
pw_dump_node: [372](#), [373](#), [375](#).
pw_dump_nodelist: [374](#), [376](#), [377](#).
pw_dump_nodes: [374](#), [375](#), [377](#), [379](#).

pw_dump_stack: [382](#), [383](#).
pw_dump_subpatch: [374](#), [378](#), [379](#).
pw_egraph: [3](#).
pw_error: [111](#), [112](#).
PW_ERROR_CHECK: [113](#).
pw_evnode: [329](#), [330](#), [331](#), [332](#), [334](#), [335](#), [336](#),
[337](#), [338](#), [339](#), [340](#), [341](#), [342](#), [343](#), [344](#), [345](#),
[346](#), [347](#), [348](#), [349](#), [350](#), [351](#), [352](#), [353](#), [354](#),
[355](#), [356](#), [357](#), [358](#), [361](#), [365](#), [366](#).
pw_evnode_bye: [334](#), [335](#).
pw_evnode_callback: [341](#), [342](#).
pw_evnode_cb: [330](#), [331](#), [332](#), [341](#), [342](#).
pw_evnode_clean: [343](#), [344](#).
pw_evnode_clean_set: [345](#), [346](#), [348](#).
pw_evnode_data_get: [347](#), [348](#).
pw_evnode_data_set: [347](#), [348](#).
pw_evnode_dur: [351](#), [352](#).
pw_evnode_dur_get: [353](#), [354](#).
pw_evnode_fire: [339](#), [340](#), [368](#).
pw_evnode_init: [335](#), [336](#), [337](#).
pw_evnode_is_terminal: [355](#), [356](#), [368](#).
pw_evnode_new: [334](#), [335](#).
pw_evnode_next: [331](#), [349](#), [350](#).
pw_evnode_set_next: [357](#), [358](#).
pw_evwalker: [360](#), [361](#), [363](#), [364](#), [365](#), [366](#),
[367](#), [368](#).
pw_evwalker_init: [365](#), [366](#), [367](#).
pw_evwalker_reset: [363](#), [364](#).
pw_evwalker_walk: [367](#), [368](#).
pw_freefun: [389](#), [390](#), [391](#), [392](#).
pw_function: [6](#), [12](#), [13](#), [14](#), [30](#), [31](#).
PW_I_DONT_KNOW: [109](#), [112](#).
PW_IGNORE: [209](#).
PW_INVALID_BUFFER: [109](#), [207](#), [209](#), [305](#).
PW_INVALID_CABLE: [40](#), [41](#), [44](#), [45](#), [109](#).
PW_INVALID_ENTRY: [109](#).
PW_INVALID_NODE: [109](#).
pw_mallocfun: [389](#), [390](#), [391](#), [392](#).
pw_memory_alloc: [37](#), [129](#), [163](#), [188](#), [224](#), [287](#),
[289](#), [335](#), [387](#), [388](#), [390](#).
pw_memory_defaults: [264](#), [393](#), [394](#).
pw_memory_free: [39](#), [131](#), [166](#), [192](#), [226](#), [289](#),
[335](#), [387](#), [388](#), [390](#).
pw_memory_override: [391](#), [392](#), [394](#).
pw_node: [3](#), [6](#), [9](#), [20](#), [22](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#),
[29](#), [30](#), [31](#), [32](#), [33](#), [34](#), [35](#), [36](#), [37](#), [38](#), [39](#), [40](#),
[41](#), [42](#), [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#), [52](#),
[53](#), [57](#), [65](#), [66](#), [83](#), [85](#), [252](#), [274](#), [276](#), [286](#), [287](#),
[309](#), [321](#), [323](#), [325](#), [372](#), [373](#), [374](#), [375](#).
pw_node_blksize: [42](#), [43](#), [93](#).
pw_node_cables_alloc: [36](#), [37](#), [38](#).
pw_node_cables_free: [24](#), [25](#), [38](#), [39](#).
pw_node_compute: [32](#), [33](#), [276](#), [321](#).
pw_node_destroy: [32](#), [33](#), [276](#), [323](#).
pw_node_get_cable: [40](#), [41](#).
pw_node_get_data: [34](#), [35](#).
pw_node_get_id: [26](#), [27](#), [80](#), [371](#).
pw_node_get_ncables: [46](#), [47](#).
pw_node_get_next: [52](#), [53](#), [274](#), [276](#), [321](#), [323](#), [325](#).
pw_node_get_patch: [50](#), [51](#).
pw_node_get_type: [48](#), [49](#).
pw_node_init: [24](#), [25](#), [287](#).
pw_node_set_block: [44](#), [45](#), [93](#).
pw_node_set_compute: [30](#), [31](#).
pw_node_set_data: [34](#), [35](#).
pw_node_set_destroy: [30](#), [31](#).
pw_node_set_id: [28](#), [29](#), [287](#).
pw_node_set_next: [52](#), [53](#), [287](#).
pw_node_set_patch: [50](#), [51](#), [287](#).
pw_node_set_setup: [30](#), [31](#).
pw_node_set_type: [48](#), [49](#).
pw_node_setup: [32](#), [33](#), [276](#).
pw_node_size: [22](#), [23](#).
pw_nodfun: [6](#).
PW_NOT_ENOUGH_NODES: [109](#).
PW_NOT_OK: [37](#), [39](#), [83](#), [93](#), [95](#), [97](#), [109](#), [198](#),
[209](#), [211](#), [224](#), [228](#), [232](#), [234](#), [236](#), [238](#), [240](#),
[305](#), [388](#), [394](#).
PW_NULL_VALUE: [109](#), [207](#), [209](#), [287](#).
PW_OK: [37](#), [40](#), [41](#), [44](#), [51](#), [80](#), [83](#), [93](#), [95](#), [97](#),
[109](#), [113](#), [129](#), [198](#), [199](#), [207](#), [209](#), [211](#), [224](#),
[228](#), [230](#), [232](#), [234](#), [236](#), [238](#), [240](#), [287](#), [289](#),
[292](#), [305](#), [306](#), [388](#), [394](#).
pw_patch: [3](#), [10](#), [50](#), [51](#), [59](#), [119](#), [128](#), [129](#), [163](#),
[164](#), [166](#), [167](#), [188](#), [189](#), [192](#), [193](#), [224](#), [225](#), [226](#),
[227](#), [250](#), [251](#), [263](#), [264](#), [265](#), [266](#), [267](#), [268](#), [269](#),
[270](#), [271](#), [272](#), [273](#), [274](#), [275](#), [276](#), [277](#), [278](#), [279](#),
[280](#), [281](#), [282](#), [284](#), [285](#), [286](#), [287](#), [288](#), [289](#), [291](#),
[292](#), [293](#), [294](#), [295](#), [296](#), [297](#), [298](#), [299](#), [300](#), [301](#),
[302](#), [303](#), [304](#), [305](#), [306](#), [307](#), [309](#), [314](#), [315](#),
[316](#), [317](#), [318](#), [319](#), [334](#), [335](#), [348](#), [376](#), [377](#),
[387](#), [388](#), [389](#), [390](#), [391](#), [392](#), [393](#), [394](#), [396](#),
[397](#), [398](#), [399](#), [400](#), [401](#), [402](#), [403](#).
pw_patch_alloc: [265](#), [266](#), [268](#).
pw_patch_append_userdata: [289](#), [290](#), [291](#), [292](#).
pw_patch_bhold: [305](#), [306](#).
pw_patch_blksize: [93](#), [295](#), [296](#).
pw_patch_bunhold: [305](#), [306](#).
pw_patch_clear: [270](#), [271](#), [272](#), [314](#).
pw_patch_compute: [275](#), [276](#), [285](#).
pw_patch_data_get: [301](#), [302](#).
pw_patch_data_set: [301](#), [302](#).
pw_patch_destroy: [275](#), [276](#).
pw_patch_free_nodes: [273](#), [274](#).

pw_patch_get_out: [279](#), [280](#), [285](#).
pw_patch_holdbuf: [303](#), [304](#).
pw_patch_init: [263](#), [264](#).
pw_patch_new_cable: [288](#), [289](#).
pw_patch_new_node: [109](#), [273](#), [286](#), [287](#).
pw_patch_pool: [297](#), [298](#), [304](#), [306](#).
pw_patch_realloc: [267](#), [268](#).
pw_patch_reinit: [264](#), [269](#), [270](#), [272](#), [314](#).
pw_patch_set_out: [277](#), [278](#).
pw_patch_setup: [275](#), [276](#).
pw_patch_size: [281](#), [282](#).
pw_patch_srate_get: [299](#), [300](#).
pw_patch_srate_set: [264](#), [299](#), [300](#).
pw_patch_stack: [45](#), [83](#), [85](#), [293](#), [294](#), [306](#).
pw_patch_tick: [284](#), [285](#).
pw_patch_unholdbuf: [303](#), [304](#).
pw_pointer: [3](#), [6](#), [116](#), [117](#), [118](#), [119](#), [122](#), [123](#),
[124](#), [126](#), [127](#), [128](#), [129](#), [130](#), [131](#), [132](#), [133](#),
[134](#), [135](#), [136](#), [137](#), [138](#), [139](#), [143](#), [144](#), [145](#),
[146](#), [148](#), [289](#), [290](#), [292](#), [348](#).
pw_pointer_create: [127](#), [128](#), [129](#), [145](#), [292](#).
pw_pointer_data: [132](#), [133](#), [289](#).
pw_pointer_destroy: [130](#), [131](#), [148](#).
pw_pointer_function: [117](#), [121](#), [128](#), [129](#), [291](#), [292](#).
pw_pointer_get_id: [135](#).
pw_pointer_get_next: [136](#), [137](#), [148](#).
pw_pointer_get_type: [134](#).
pw_pointer_set_next: [138](#), [139](#), [146](#).
pw_pointer_set_type: [134](#), [135](#).
pw_pointerlist: [3](#), [123](#), [124](#), [125](#), [140](#), [141](#), [142](#),
[143](#), [144](#), [145](#), [146](#), [147](#), [148](#), [257](#), [290](#), [309](#), [348](#).
pw_pointerlist_append: [145](#), [146](#), [292](#).
pw_pointerlist_free: [147](#), [148](#), [276](#), [325](#).
pw_pointerlist_init: [141](#), [142](#), [272](#), [313](#), [325](#).
pw_pointerlist_top: [143](#), [144](#), [148](#).
PW_POOL_FULL: [109](#), [199](#).
pw_print: [396](#), [397](#).
pw_print_default: [399](#), [400](#), [401](#).
pw_print_init: [264](#), [399](#), [400](#).
pw_print_set: [402](#), [403](#).
pw_stack: [3](#), [83](#), [85](#), [93](#), [94](#), [153](#), [219](#), [220](#), [222](#),
[223](#), [224](#), [225](#), [226](#), [227](#), [228](#), [229](#), [230](#), [231](#),
[232](#), [233](#), [234](#), [235](#), [236](#), [237](#), [238](#), [239](#), [240](#),
[241](#), [242](#), [243](#), [244](#), [246](#), [247](#), [248](#), [259](#), [293](#),
[294](#), [306](#), [382](#), [383](#).
pw_stack_alloc: [222](#), [224](#), [225](#), [226](#), [266](#).
pw_stack_drop: [236](#), [237](#).
pw_stack_dup: [234](#), [235](#).
pw_stack_free: [226](#), [227](#), [268](#), [276](#).
pw_stack_hold: [240](#), [241](#).
pw_stack_init: [222](#), [223](#), [266](#).
PW_STACK_OVERFLOW: [109](#), [228](#), [230](#).
pw_stack_pop: [83](#), [232](#), [233](#), [306](#).
pw_stack_pos: [244](#), [245](#), [246](#).
pw_stack_push: [85](#), [93](#), [228](#), [229](#).
pw_stack_push_buffer: [230](#), [231](#).
pw_stack_reset: [247](#), [248](#), [270](#).
pw_stack_size: [242](#), [243](#).
pw_stack_swap: [238](#), [239](#).
pw_subpatch: [3](#), [309](#), [312](#), [313](#), [314](#), [315](#), [316](#),
[317](#), [318](#), [319](#), [320](#), [321](#), [322](#), [323](#), [324](#), [325](#),
[326](#), [327](#), [348](#), [378](#), [379](#).
pw_subpatch_compute: [320](#), [321](#).
pw_subpatch_destroy: [322](#), [323](#).
pw_subpatch_free: [324](#), [325](#).
pw_subpatch_init: [311](#), [312](#), [313](#).
pw_subpatch_out: [326](#), [327](#).
pw_subpatch_restore: [317](#), [318](#), [319](#).
pw_subpatch_save: [314](#), [315](#), [316](#).
PWFLT: [6](#), [58](#), [59](#), [69](#), [70](#), [71](#), [72](#), [73](#), [74](#), [75](#), [76](#), [77](#),
[78](#), [93](#), [97](#), [98](#), [106](#), [154](#), [163](#), [176](#), [177](#), [284](#), [285](#).
rc: [37](#), [111](#), [112](#), [113](#), [129](#), [228](#), [232](#), [287](#), [289](#),
[292](#), [306](#), [335](#).
read: [154](#), [170](#), [172](#), [174](#), [179](#), [181](#), [182](#), [190](#), [203](#),
[207](#), [209](#), [381](#), [385](#).
realloc: [268](#).
root: [124](#), [142](#), [144](#), [146](#).
s: [378](#), [379](#), [382](#), [383](#).
setup: [12](#), [25](#), [31](#), [33](#).
size: [125](#), [142](#), [146](#), [148](#), [157](#), [163](#), [164](#), [186](#), [188](#),
[190](#), [192](#), [200](#), [203](#), [211](#), [219](#), [222](#), [224](#), [225](#), [228](#),
[230](#), [234](#), [242](#), [381](#), [383](#), [387](#), [388](#), [394](#).
smtp: [285](#).
sr: [260](#), [299](#), [300](#).
stack: [83](#), [85](#), [93](#), [94](#), [222](#), [223](#), [224](#), [225](#), [226](#), [227](#),
[228](#), [229](#), [230](#), [231](#), [232](#), [233](#), [234](#), [235](#), [236](#), [237](#),
[238](#), [239](#), [240](#), [241](#), [242](#), [243](#), [244](#), [246](#), [247](#), [248](#),
[259](#), [266](#), [268](#), [270](#), [276](#), [294](#), [306](#).
stack_size: [265](#), [266](#), [267](#), [268](#).
start: [365](#), [366](#).
str: [401](#).
subpatch: [312](#), [313](#), [315](#), [316](#), [318](#), [319](#), [320](#), [321](#),
[322](#), [323](#), [324](#), [325](#), [326](#), [327](#).
sum: [97](#), [98](#).
terminal: [361](#), [364](#), [366](#), [368](#).
tmp: [83](#), [85](#), [106](#), [232](#), [238](#), [287](#), [289](#).
type: [18](#), [25](#), [48](#), [49](#), [60](#), [66](#), [69](#), [72](#), [76](#), [78](#), [82](#),
[87](#), [89](#), [120](#), [129](#), [135](#), [371](#), [373](#).
ud: [15](#), [35](#), [120](#), [128](#), [129](#), [133](#), [261](#), [291](#), [292](#), [301](#),
[302](#), [332](#), [337](#), [347](#), [348](#), [387](#), [388](#), [394](#).
useractive: [206](#).
usrnactive: [186](#), [206](#), [207](#), [209](#), [211](#), [213](#).
va_end: [397](#).
va_start: [397](#).

val: [58](#), [66](#), [69](#), [71](#), [72](#), [73](#), [74](#), [76](#), [77](#), [78](#), [95](#),
[97](#), [103](#), [148](#).

void: [6](#), [117](#).

vprintf: [401](#).

walker: [363](#), [364](#), [365](#), [366](#), [367](#), [368](#).

zero: [253](#), [270](#), [272](#).

- ⟨ A Single Buffer 152, 161, 163, 166, 168, 170, 172, 174, 176, 179, 181, 182, 184 ⟩ Cited in section 63. Used in section 150.
- ⟨ An Empty Cleanup Callback 338 ⟩ Used in section 337.
- ⟨ An Empty Event Callback 331 ⟩ Used in section 337.
- ⟨ Buffer Pool Top 150 ⟩ Cited in section 265. Used in section 149.
- ⟨ Cable Data 55 ⟩ Used in section 7.
- ⟨ Cable Enums 61 ⟩ Cited in section 60. Used in section 54.
- ⟨ Cable Top 66, 68, 69, 72, 74, 76, 78, 80, 82, 83, 85, 87, 89, 91, 93, 95, 97, 99, 103, 106 ⟩ Used in section 54.
- ⟨ Check for previously freed buffer 201 ⟩ Used in section 199.
- ⟨ Check if Buffer Pool is Full 200 ⟩ Used in section 199.
- ⟨ Data Dumpers 371, 373, 375, 377, 379, 381, 383, 385 ⟩ Used in section 369.
- ⟨ Data for a Buffer Pool 156, 157, 158, 159, 206 ⟩ Used in section 155.
- ⟨ Data for a Single Buffer 153, 154 ⟩ Used in section 152.
- ⟨ Default Print Function 401 ⟩ Used in section 400.
- ⟨ Error Codes 109 ⟩ Used in section 108.
- ⟨ Event Graph Top 332, 335, 337, 340, 342, 344, 346, 348, 350, 352, 354, 356, 358, 359 ⟩ Used in section 328.
- ⟨ Event Walker Top 364, 366, 368 ⟩ Used in section 359.
- ⟨ Header 6, 7, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 65, 67, 70, 71, 73, 75, 77, 79, 81, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 105, 108, 111, 113, 128, 130, 132, 134, 136, 138, 141, 143, 145, 147, 162, 164, 167, 169, 171, 173, 175, 177, 178, 180, 183, 185, 187, 189, 191, 193, 195, 197, 198, 208, 210, 212, 214, 216, 223, 225, 227, 229, 231, 233, 235, 237, 239, 241, 243, 246, 247, 263, 265, 267, 269, 271, 273, 275, 277, 279, 281, 284, 286, 288, 291, 293, 295, 297, 299, 301, 303, 305, 312, 315, 318, 320, 322, 324, 326, 330, 334, 336, 339, 341, 343, 345, 347, 349, 351, 353, 355, 357, 363, 365, 367, 370, 372, 374, 376, 378, 380, 382, 384, 387, 391, 393, 396, 399, 402 ⟩ Cited in section 309. Used in section 5.
- ⟨ Memory Handling 388, 392, 394 ⟩ Used in section 386.
- ⟨ Node Data 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ⟩ Used in section 9.
- ⟨ Node Top 9, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53 ⟩ Used in section 8.
- ⟨ Patch Data 250 ⟩ Used in section 249.
- ⟨ Patch Top 264, 266, 268, 270, 272, 274, 276, 278, 280, 282, 285, 287, 289, 292, 294, 296, 298, 300, 302, 304, 306, 397, 400, 403 ⟩ Used in section 249.
- ⟨ Pointer List Struct Data 124, 125 ⟩ Used in section 123.
- ⟨ Pointer Struct Data 119, 120, 121, 122 ⟩ Used in section 118.
- ⟨ Pointer Top 118, 129, 131, 133, 135, 137, 139, 142, 144, 146, 148 ⟩ Cited in section 348. Used in section 114.
- ⟨ Stack Data 219 ⟩ Used in section 218.
- ⟨ Stack Top 218, 222, 224, 226, 228, 230, 232, 234, 236, 238, 240, 242, 244, 248 ⟩ Cited in section 265. Used in section 217.
- ⟨ Struct Contents in Event Walker 361 ⟩ Used in section 360.
- ⟨ Subpatch Top 313, 316, 319, 321, 323, 325, 327 ⟩ Used in section 307.
- ⟨ Successful Finalization 204 ⟩ Used in section 199.
- ⟨ The Buffer Pool 155, 186, 188, 190, 192, 194, 196, 199, 207, 209, 211, 213, 215 ⟩ Cited in section 63. Used in section 150.
- ⟨ Top 8, 54, 110, 112, 114, 149, 217, 249, 307, 328, 369, 386 ⟩ Used in section 4.
- ⟨ Type Declarations 56, 117, 123, 151, 220, 251, 309, 329, 360, 390 ⟩ Used in section 6.
- ⟨ Use brute force to find next free buffer 203 ⟩ Cited in section 201. Used in section 199.
- ⟨ Use recently freed buffer 202 ⟩ Used in section 199.
- ⟨ Variables in Cable Data 57, 58, 59, 60, 62, 63 ⟩ Used in section 55.
- ⟨ Variables in Patch Data 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 389, 398 ⟩ Used in section 250.
- ⟨ patchwerk.h 5 ⟩ Cited in section 56.

PATCHWERK

	Section	Page
Introduction	1	1
An Overview of the System	2	2
Overview of Sections	3	3
Header	5	4
Type Definitions	6	5
C structs	7	6
Node	8	7
Data	9	8
Functions	21	10
Cable	54	16
Data	55	17
Functions	64	19
Error handling	107	26
Pointer	114	28
Pointer Data and Type Declarations	115	29
Pointer Functions	126	31
Pointer List Functions	140	33
Buffer Pool	149	35
Data	150	36
Buffer Functions	160	38
Intialization Functions	186	41
Finding the next free buffer	198	43
User-space buffers	205	45
Buffer Stack	217	48
Buffer Stack Data	218	49
Buffer Stack Functions	221	50
Patch	249	55
Data	250	56
Functions	262	58
High Level Functions	283	63
Subpatch	307	68
Data	308	69
Functions	310	70
Event Graphs	328	73
A Single Event Node	329	74
Event Node Functions	333	75
The Event Walker	359	79
Event Walker Data	360	80
Event Walker Functions	362	81
Data Dumping	369	83
Memory Handling	386	88
Printing (WIP)	395	90