

HOWTO  
write an External  
for *pure-data*

johannes m zmölnig

*institut für elektronische musik und akustik*

## Zusammenfassung

pd ist ein graphisches Computermusiksystem in der Tradition von IRCAMs *ISPW-max*.

Obwohl eine Fülle von Funktionen von pd selbst zur Verfügung gestellt werden, stößt man doch manchmal an die Grenzen dessen, das mit diesen Primitiven und ihren Kombinationen möglich ist.

Deswegen bietet pd die Möglichkeit, eigene Primitive (“objects”, Objekte) in komplexen Programmiersprachen wie C/C++ zu erstellen.

In diesem Dokument soll beschrieben werden, wie man solche Primitive mit Hilfe der Sprache C, in der auch pd selbst realisiert wurde, schreibt.

# Inhaltsverzeichnis

<b>1</b>	<b>Voraussetzungen und Begriffsbestimmungen</b>	<b>2</b>
1.1	Klassen, Instanzen und Objekte . . . . .	2
1.2	Internals, Externals und Libraries . . . . .	2
<b>2</b>	<b>mein erstes External: helloworld</b>	<b>3</b>
2.1	die Schnittstelle zu pd . . . . .	3
2.2	eine Klasse und ihr Datenraum . . . . .	4
2.3	Methodenraum . . . . .	4
2.4	Generierung einer neuen Klasse . . . . .	5
2.5	Konstruktor: Instanziierung eines Objektes . . . . .	6
2.6	der Code: <code>helloworld</code> . . . . .	7
<b>3</b>	<b>ein komplexes External: counter</b>	<b>7</b>
3.1	Variablen eines Objektes . . . . .	8
3.2	Übergabeargumente . . . . .	8
3.3	Konstruktor . . . . .	9
3.4	die Zählermethode . . . . .	9
3.5	der Code: <code>counter</code> . . . . .	10
<b>4</b>	<b>ein komplexeres External: counter</b>	<b>11</b>
4.1	erweiterter Datenraum . . . . .	11
4.2	Erweiterung der Klasse . . . . .	12
4.3	Konstruktion von In- und Outlets . . . . .	13
4.4	erweiterter Methodenraum . . . . .	15
4.5	der Code: <code>counter</code> . . . . .	16
<b>5</b>	<b>ein Signal-External: pan~</b>	<b>18</b>
5.1	Variablen einer Signalklasse . . . . .	19
5.2	Signalklassen . . . . .	19
5.3	Konstruktion von Signal-In- und Outlets . . . . .	20
5.4	DSP-Methode . . . . .	20
5.5	perform-Routine . . . . .	21
5.6	der Code: <code>pan~</code> . . . . .	22
<b>A</b>	<b>das Message-System von <i>pd</i></b>	<b>24</b>
A.1	Atome . . . . .	24
A.2	Selectoren . . . . .	24
<b>B</b>	<b>pd-Typen</b>	<b>25</b>

<b>C</b>	<b>Wichtige Funktionen aus “m_pd.h”</b>	<b>26</b>
C.1	Funktionen: Atome . . . . .	26
C.1.1	SETFLOAT . . . . .	26
C.1.2	SETSYMBOL . . . . .	26
C.1.3	SETPOINTER . . . . .	26
C.1.4	atom_getfloat . . . . .	27
C.1.5	atom_getfloatarg . . . . .	27
C.1.6	atom_getint . . . . .	27
C.1.7	atom_getsymbol . . . . .	27
C.1.8	atom_gensym . . . . .	27
C.1.9	atom_string . . . . .	27
C.1.10	gensym . . . . .	28
C.2	Funktionen: Klassen . . . . .	28
C.2.1	class_new . . . . .	28
C.2.2	class_addmethod . . . . .	29
C.2.3	class_addbang . . . . .	29
C.2.4	class_addfloat . . . . .	29
C.2.5	class_addsymbol . . . . .	30
C.2.6	class_addpointer . . . . .	30
C.2.7	class_addlist . . . . .	30
C.2.8	class_addanything . . . . .	30
C.2.9	class_addcreator . . . . .	31
C.2.10	class_sethelpsymbol . . . . .	31
C.2.11	pd_new . . . . .	31
C.3	Funktionen: In- und Outlets . . . . .	31
C.3.1	inlet_new . . . . .	31
C.3.2	floatinlet_new . . . . .	32
C.3.3	symbolinlet_new . . . . .	32
C.3.4	pointerinlet_new . . . . .	32
C.3.5	outlet_new . . . . .	33
C.3.6	outlet_bang . . . . .	33
C.3.7	outlet_float . . . . .	33
C.3.8	outlet_symbol . . . . .	33
C.3.9	outlet_pointer . . . . .	34
C.3.10	outlet_list . . . . .	34
C.3.11	outlet_anything . . . . .	34
C.4	Funktionen: DSP . . . . .	34
C.4.1	CLASS_MAINSIGNALIN . . . . .	35
C.4.2	dsp_add . . . . .	36
C.4.3	sys_getsr . . . . .	36
C.5	Funktion: Memory . . . . .	36

C.5.1	getbytes	36
C.5.2	copybytes	36
C.5.3	freebytes	36
C.6	Funktionen: Ausgabe	37
C.6.1	post	37
C.6.2	error	37

# 1 Voraussetzungen und Begriffsbestimmungen

pd bezieht sich auf das graphische Echtzeit-Computermusikprogramm von Miller S. Puckette. *puredata*.

Zum Verständnis dieses Dokumentes wird der Umgang mit pd sowie Verständnis von Programmier-Techniken, insbesondere C vorausgesetzt.

Zum Schreiben von eigenen Primitiven wird weiters ein C-Compiler, der dem ANSI-C-Standard genügt, notwendig sein. Solche Compiler sind beispielsweise der *Gnu C-Compiler* (*gcc*) auf linux-Systemen oder *Visual-C++ 6.0* (*vc6*) auf Windows-Systemen.

## 1.1 Klassen, Instanzen und Objekte

pd ist in der Programmiersprache C geschrieben. Allerdings ist pd auf Grund seiner graphischen Natur ein *objektorientiertes* System. Da C die Verwendung von Klassen nicht sehr gut unterstützt, ist der resultierende Quellcode nicht so elegant wie er zum Beispiel unter C++ wäre.

Der Ausdruck *Klasse* bezieht sich in diesem Dokument auf die Realisierung eines Konzeptes, bei dem Daten und Manipulatoren eine Einheit bilden.

Konkrete *Instanzen einer Klasse* sind *Objekte*.

## 1.2 Internals, Externals und Libraries

Um Begriffsverwirrungen von vorneherein auszuschließen, seien hier kurz die Ausdrücke *Internal*, *External* und *Library* erklärt.

**Internal** Ein *Internal* ist eine Klasse, die in pd eingebaut ist. Viele Primitive wie "+", "pack" oder "sig~" sind *Internals*

**External** Ein *External* ist eine Klasse, die nicht in pd eingebaut ist und erst zur Laufzeit nachgeladen wird. Sind sie einmal im Speicher von pd, so sind *Externals* nicht mehr von *Internals* zu unterscheiden.

**Library** Eine *Library* bezeichnet eine Sammlung von *Externals*, die gemeinsam in eine Binärdatei kompiliert werden.

*Library*-Dateien müssen eine betriebssystemabhängige Namenskonvention einhalten:

Bibliothek	linux	irix	Win32
my_lib	my_lib.pd_linux	my_lib.pd_irix	my_lib.dll

Die einfachste Form einer *Library* beinhaltet genau ein *External*, das den selben Name trägt, wie auch die *Library*

Im Gegensatz zu *Externals* können *Libraries* mit bestimmten Befehlen von pd importiert werden. Ist eine *Library* importiert worden, so sind alle *Externals*, die sie beinhaltet, in den Speicher geladen und stehen als Objekte zur Verfügung.

pd stellt zwei Methoden zur Verfügung, um *Libraries* zu laden:

- mit der commandline-Option “-lib my\_lib”
- durch Kreieren eines Objektes “my\_lib”

Die erste Methode lädt die *Library* sofort beim Starten von pd. Dies ist die zu bevorzugende Methode für *Libraries*, die mehrere *Externals* beinhalten.

Die zweite Methode ist für *Libraries* zu bevorzugen, die genau ein *External* mit dem selben Namen beinhalten. Bei der zweiten Methode wird zuerst geprüft, ob eine Klasse namens “my\_lib” bereits in den Speicher geladen ist. Ist dies nicht der Fall<sup>1</sup> so werden alle Pfade untersucht, ob darin eine Datei namens “my\_lib.pd\_linux”<sup>2</sup> existiert. Wird eine solche Datei gefunden, so werden alle in ihr enthaltenen *Externals* in den Speicher geladen. Danach wird nachgesehen, ob nun eine Klasse namens “my\_lib” als (neu geladenes) *External* im Speicher existiert. Ist dies der Fall, so wird eine Instanz dieser Klasse geschaffen. Ansonsten wird eine Fehlermeldung ausgegeben, die Instanziierung ist gescheitert.

## 2 mein erstes External: helloworld

Wie das beim Erlernen von Programmiersprachen so üblich ist, beginnen wir mit “Hello world”.

Ein Objekt soll geschaffen werden, dass jedesmal, wenn es mit “bang” getriggert wird, die Zeile “Hello world!” auf die Standardausgabe schreibt.

### 2.1 die Schnittstelle zu pd

Um ein pd-External zu schreiben, braucht man eine wohldefinierte Schnittstelle. Diese wird in der Datei “m\_pd.h” zur Verfügung gestellt.

```
#include "m_pd.h"
```

---

<sup>1</sup>Ist eine solche Klasse bereits im Speicher, wird ein Objekt namens “my\_lib” instanziiert und der Vorgang bricht ab. Es wird also keine neue *Library* geladen. Man kann daher keine *Libraries* mit bereits verwendeten Klassennamen, wie zum Beispiel “abs”, laden.

<sup>2</sup> oder einer anderen betriebssystemabhängigen Dateinamenerweiterung (s.o.)

## 2.2 eine Klasse und ihr Datenraum

Als nächstes muß eine neue Klasse vorbereitet und der Datenraum für diese Klasse definiert werden.

```
static t_class *helloworld_class;

typedef struct _helloworld {
    t_object  x_obj;
} t_helloworld;
```

`hello_worldclass` wird der Zeiger auf die neue Klasse.

Die Struktur `t_helloworld` (vom Typ `_helloworld`) stellt den Datenraum der Klasse dar. Ein unverzichtbares Element ist dabei eine Variable des Type `t_object`. In ihr werden interne Objekteigenschaften abgelegt, wie zum Beispiel die Größe der Objekt-Box bei der graphischen Darstellung, aber auch Daten über Inlets und Outlets. `t_object` muss der erste Eintrag in die Struktur sein !

Da bei einer einfachen “Hello world”-Anwendung keine Variablen gebraucht werden, ist die Struktur ansonsten leer.

## 2.3 Methodenraum

Zu einer Klasse gehören neben einem Datenraum auch ein Satz von Manipulatoren (Methoden) mit denen diese Daten manipuliert werden können.

Wird eine Message an eine Instanz unserer Klasse geschickt, so wird eine Methoden aufgerufen. Diese Methoden, die die Schnittstelle zum Messagesystem von pd bilden, haben grundsätzlich kein Rückgabeargument, sind also vom Typ `void`.

```
void helloworld_bang(t_helloworld *x)
{
    post("Hello world !!");
}
```

Diese Methode hat ein Übergabeargument vom Typ `t_helloworld`, sodass wir also unseren Datenraum manipulieren könnten.

Da wir nur “Hello world!” ausgeben wollen (und ausserdem unser Datenraum recht spärlich ist), verzichten wir auf eine Manipulation.

Mit dem Befehl `post(char *c, ...)` wird eine Meldung an die Standardausgabe geschickt. Ein Zeilenumbruch wird automatisch angehängt. Ansonsten funktioniert `post()` gleich wie der C-Befehl `printf()`.

## 2.4 Generierung einer neuen Klasse

Um eine neue Klasse zu generieren, müssen Angaben über den Datenraum und den Methodenraum dieser Klasse beim Laden einer Library an pd übergeben werden.

Wird eine neue Library “my\_lib” geladen, so versucht pd eine Funktion “my\_lib\_setup()” aufzurufen. Diese Funktion (oder von ihr aufgerufene Funktionen) teilt pd mit, welche Eigenschaften die neuen Klassen haben. Sie wird nur einmal, beim Laden der Library aufgerufen.

```
void helloworld_setup(void)
{
    helloworld_class = class_new(gensym("helloworld"),
        (t_newmethod)helloworld_new,
        0, sizeof(t_helloworld),
        CLASS_DEFAULT, 0);

    class_addbang(helloworld_class, helloworld_bang);
}
```

**class\_new** Der Befehl `class_new` kreiert eine neue Klasse und gibt einen Zeiger auf diesen Prototyp zurück.

Das erste Argument ist der symbolische Name der Klasse.

Die nächsten beiden Argumente definieren Konstruktor und Destruktor der Klasse. Wenn in einen pd-Patch ein Objekt kreiert wird, instanziiert der Konstruktor (`t_newmethod`)`helloworld_new` dieses Objekt und initialisiert den Datenraum. Wird ein pd-Patch geschlossen oder ein Objekt daraus entfernt, so gibt der Destruktor, wenn notwendig, dynamisch reservierten Speicher wieder frei. Der Speicherplatz für den Datenraum selbst wird von pd automatisch freigegeben. Deshalb kann in diesem Beispiel auf einen Destruktor verzichtet werden, folglich wird dieses Argument auf “0” gesetzt.

Damit pd genug Speicher für den Datenraum allozieren und wieder freigeben kann, wird die Größe dieser Datenstruktur als viertes Argument übergeben.

Das fünfte Argument bestimmt, wie Klasseninstanzen graphisch dargestellt werden und ob sie mit anderen Objekten verknüpfbar sind. Der Standardwert `CLASS_DEFAULT` (oder einfacher: “0”) bezieht sich auf ein Objekt mit mindestens einem Inlet. Würde man keinen Eingang wollen (wie zum Beispiel beim Internal “receive”), so kann man diesen Wert auf `CLASS_NOINLET` setzen.

Die restlichen Argumente definieren die Übergabeargumente eines Objektes und deren Typ.

Bis zu sechs numerische und symbolische Objektargumente können in beliebiger Reihenfolge mit `A_DEFFLOAT` und `A_DEFSYMBOL` angegeben werden. Sollen mehr Argumente übergeben werden oder die Atomtyp-Reihenfolge flexibler sein, so bietet `A_GIMME` die Übergabe einer beliebigen Liste von Atomen.

Die Objektargumentliste wird mit "0" terminiert. In unserem Beispiel sind also keine Übergabeargumente für die Klasse vorgesehen.

**class\_addbang** Jetzt muss zur Klasse noch ein Methodenraum hinzugefügt werden.

Mit `class_addbang` wird der durch das erste Argument definierten Klasse eine Methode für eine "bang"-Message hinzugefügt. Diese Methode ist das zweite Argument.

## 2.5 Konstruktor: Instanziierung eines Objektes

Jedesmal, wenn in einem pd-Patch ein Objekt einer Klasse kreiert wird, schafft der mit `class_new` angegebene Konstruktor eine neue Instanz der Klasse.

Der Konstruktor ist immer vom Typ `void *`

```
void *helloworld_new(void)
{
    t_helloworld *x = (t_helloworld *)pd_new(helloworld_class);

    return (void *)x;
}
```

Die Übergabeargumente der Konstruktorfunktion hängen von den mit `class_new` angegebenen Objektargumenten ab.

<code>class_new</code> -Argument	Konstruktorargument
<code>A_DEFFLOAT</code>	<code>t_floatarg f</code>
<code>A_DEFSYMBOL</code>	<code>t_symbol *s</code>
<code>A_GIMME</code>	<code>t_symbol *s, int argc, t_atom *argv</code>

Da in diesem Beispiel keine Objektargumente existieren, hat auch der Konstruktor keine.

Die Funktion `pd_new` reserviert Speicher für den Datenraum, initialisiert die objektinternen Variablen und gibt einen Zeiger auf den Datenraum zurück.

Der Typ-Cast auf den Datenraum ist notwendig.

Normalerweise würden im Konstruktor auch die Objektvariablen initialisiert werden. In diesem Beispiel ist dies aber nicht notwendig.

Der Konstruktor muss einen Zeiger auf den instanziierten Datenraum zurückgeben.

## 2.6 der Code: helloworld

```
#include "m_pd.h"

static t_class *helloworld_class;

typedef struct _helloworld {
    t_object x_obj;
} t_helloworld;

void helloworld_bang(t_helloworld *x)
{
    post("Hello world !!");
}

void *helloworld_new(void)
{
    t_helloworld *x = (t_helloworld *)pd_new(helloworld_class);

    return (void *)x;
}

void helloworld_setup(void) {
    helloworld_class = class_new(gensym("helloworld"),
        (t_newmethod)helloworld_new,
        0, sizeof(t_helloworld),
        CLASS_DEFAULT, 0);
    class_addbang(helloworld_class, helloworld_bang);
}
```

## 3 ein komplexes External: counter

Als nächstes soll ein einfacher Zähler als External geschrieben werden. Ein "bang"-Trigger soll den aktuellen Zählerstand am Outlet ausgeben und anschließend um 1 erhöhen.

Diese Klasse unterscheidet sich nicht sonderlich von der vorherigen, ausser dass nun eine interne Variable "Zählerstand" benötigt wird und das Ergebnis nicht mehr auf die Standardausgabe geschrieben sondern als Message zu einem Outlet geschickt wird.

### 3.1 Variablen eines Objektes

Ein Zähler braucht natürlich eine Zustandsvariable, in der der aktueller Zählerstand gespeichert ist.

Solche zum Objekt gehörigen Zustandsvariablen werden im Datenraum abgelegt.

```
typedef struct _counter {
    t_object  x_obj;
    t_int  i_count;
} t_counter;
```

Die Ganzzahlvariable `i_count` beschreibt den Zählerstand. Natürlich könnte man sie auch als Gleitkommawert realisieren, doch traditionell werden Zähler ganzzahlig ausgeführt.

### 3.2 Übergabeargumente

Für einen Zähler ist es durchaus sinnvoll, wenn man den Startwert festlegen kann. Hier soll der Startwert dem Objekt bei der Kreation übergeben werden.

```
void counter_setup(void) {
    counter_class = class_new(gensym("counter"),
        (t_newmethod)counter_new,
        0, sizeof(t_counter),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

    class_addbang(counter_class, counter_bang);
}
```

Es ist also ein Argument zur Funktion `class_new` hinzugekommen:

`A_DEFFLOAT` teilt mit, dass das Objekt ein Übergabeargument vom Typ `t_floatarg` hat.

### 3.3 Konstruktor

Dem Konstruktor kommen nun mehrere neue Aufgaben zu. Zum ersten muss eine Variable initialisiert werden, zum anderen muss auch ein Outlet für das Objekt geschaffen werden.

```
void *counter_new(t_floatarg f)
{
    t_counter *x = (t_counter *)pd_new(counter_class);

    x->i_count=f;
    outlet_new(&x->x_obj, &s_float);

    return (void *)x;
}
```

Die Konstruktorfunktion hat jetzt ein Argument vom Typ `t_floatarg`, wie es in der Setup-Routine `class_new` deklariert worden ist. Dieses Argument initialisiert den Zähler.

Einer neuer Outlet wird mit der Funktion `outlet_new` geschaffen. Das erste Argument ist ein Zeiger auf die Objektinterna, in denen der neue Ausgang geschaffen wird.

Das zweite Argument ist eine symbolische Typbeschreibung des Ausgangs. Da der Zähler numerische Werte ausgeben soll, ist er vom Typ "float". Sollte der Ausgang für Messages mit verschiedenen Selectoren verwendet werden, so ist dieser Wert "0".

`outlet_new` gibt einen Zeiger auf den neuen Outlet zurück und speichert diesen Zeiger in der `t_object`-Variablen `x_obj.ob_outlet`. Wird nur ein Outlet verwendet, muss daher der Zeiger nicht extra im Datenraum gespeichert werden. Werden mehrere Outlets verwendet, so müssen diese Zeiger im Datenraum gespeichert werden.

### 3.4 die Zählermethode

Bei einem Triggerevent soll der alte Zählerstand ausgegeben und um eins inkrementiert werden.

```
void counter_bang(t_counter *x)
{
    t_float f=x->i_count;
    x->i_count++;
    outlet_float(x->x_obj.ob_outlet, f);
}
```

Die Funktion `outlet_float` gibt an dem Outlet, auf den das erste Argument verweist, eine Gleitkommazahl (zweites Argument) aus.

Hier wird zuerst der Zählerstand in eine Gleitkomma-Buffervariable gespeichert. Danach wird er inkrementiert und dann wird erst die Buffervariable ausgegeben.

Was auf den ersten Blick unnötig erscheint, macht bei näherer Betrachtung Sinn: Die Buffervariable wurde gleich als `t_float` realisiert, da sich `outlet_float` sowieso einen Gleitkommawert erwartet und ein Cast unvermeidlich ist.

Würde der Zählerstand zuerst an den Outlet geschickt werden und danach erst inkrementiert werden, würde dies unter Umständen zu einem etwas seltsamen Verhalten führen. Wenn nämlich der Zählerausgang wieder an den Inlet zurückgeführt würde, der Zähler sich also selbst triggerte, so würde die Zählermethode erneut aufgerufen, ohne dass der Zählerstand inkrementiert worden wäre. Dies ist im Allgemeinen aber unerwünscht.

Man kann übrigens das gleiche Ergebnis wie hier mit nur einer einzigen Zeile erreichen, doch sieht man das *Reentrant*-Problem dann nicht sehr gut.

### 3.5 der Code: counter

```
#include "m_pd.h"

static t_class *counter_class;

typedef struct _counter {
    t_object  x_obj;
    t_int  i_count;
} t_counter;

void counter_bang(t_counter *x)
{
    t_float f=x->i_count;
    x->i_count++;
    outlet_float(x->x_obj.ob_outlet, f);
}

void *counter_new(t_floatarg f)
{
    t_counter *x = (t_counter *)pd_new(counter_class);

    x->i_count=f;
}
```

```

    outlet_new(&x->x_obj, &s_float);

    return (void *)x;
}

void counter_setup(void) {
    counter_class = class_new(gensym("counter"),
        (t_newmethod)counter_new,
        0, sizeof(t_counter),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

    class_addbang(counter_class, counter_bang);
}

```

## 4 ein komplexeres External: counter

Man kann natürlich auch einen einfache Zähler ein bißchen komplexer gestalten. Es wäre zum Beispiel sinnvoll, wenn der Zählerstand auf einen Startwert zurückgesetzt werden könnte, wenn man Start- und Endwert bestimmen könnte und auch die Schrittweite variabel wäre.

Bei jedem Zählerüberlauf soll ein zweiter Outlet eine “bang”-Message schicken und der Zähler auf den Startwert zurückgesetzt werden.

### 4.1 erweiterter Datenraum

```

typedef struct _counter {
    t_object x_obj;
    t_int i_count;
    t_float step;
    t_int i_down, i_up;
    t_outlet *f_out, *b_out;
} t_counter;

```

Der Datenraum wurde also erweitert um Variablen für Schrittweite und Start- bzw. Stopwert. Weiters werden Zeiger auf zwei Outlets zur Verfügung gestellt.

## 4.2 Erweiterung der Klasse

Da nun die Klassenobjekte verschiedene Messages, wie “set” und “reset”, verstehen können sollen, müssen der Methodenraum entsprechend erweitert werden.

```
counter_class = class_new(gensym("counter"),
    (t_newmethod)counter_new,
    0, sizeof(t_counter),
    CLASS_DEFAULT,
    A_GIMME, 0);
```

Der Klassengenerator `class_new` ist um das Objektübergabeargument `A_GIMME` erweitert. Damit kann eine dynamische Anzahl von Argumenten bei der Objektinstanziierung verwaltet werden.

```
class_addmethod(counter_class,
    (t_method)counter_reset,
    gensym("reset"), 0);
```

`class_addmethod` fügt einer Klasse eine Methode mit für einen beliebigen Selector hinzu.

Das erste Argument ist die Klasse, zu der die Methode (zweites Argument) hinzugefügt wird.

Das dritte Argument ist der symbolische Selector, der mit der Methode assoziiert wird.

Die restlichen “0”-terminierten Argumente beschreiben die Atomliste, die dem Selector folgt.

```
class_addmethod(counter_class,
    (t_method)counter_set, gensym("set"),
    A_DEFFLOAT, 0);
class_addmethod(counter_class,
    (t_method)counter_bound, gensym("bound"),
    A_DEFFLOAT, A_DEFFLOAT, 0);
```

Eine Methode für den Selector “set”, gefolgt von einem numerischen Wert, wird hinzugefügt.

Für den Selector “bound”, gefolgt von zwei numerischen Werten, wird ebenfalls eine Methode zur Klasse hinzugefügt.

```
class_sethelpsymbol(counter_class, gensym("help-counter"));
```

Clickt man mit der rechten Maustaste auf ein pd-Objekt, so kann man sich einen Hilfe-Patch für die zugehörige Objektklasse anzeigen lassen. Standardmäßig wird dies ein Patch mit dem symbolischen Klassennamen im Verzeichnis “*doc/5.reference/*” gesucht. Mit dem Befehl `class_sethelpsymbol` kann ein alternativer Patch angegeben werden.

### 4.3 Konstruktion von In- und Outlets

Bei der Objektkreation sollten dem Objekt verschiedene Argumente übergeben werden.

```
void *counter_new(t_symbol *s, int argc, t_atom *argv)
```

Durch die Argumentendeklaration in der `class_new`-Funktion mit `A_GIMME`, werden dem Konstruktor folgende Argumente übergeben:

<code>t_symbol *s</code>	der symbolische Namen, mit dem das Objekt kreiert wurde
<code>int argc</code>	die Anzahl, der dem Objekt übergebenen Argumente
<code>t_atom *argv</code>	ein Zeiger auf eine Liste von <code>argc</code> Atomen

```
t_float f1=0, f2=0;

x->step=1;
switch(argc){
default:
case 3:
    x->step=atom_getfloat(argv+2);
case 2:
    f2=atom_getfloat(argv+1);
case 1:
    f1=atom_getfloat(argv);
    break;
case 0:
}
if (argc<2)f2=f1;
x->i_down = (f1<f2)?f1:f2;
x->i_up   = (f1>f2)?f1:f2;

x->i_count=x->i_down;
```

Werden drei Argumente übergeben, so sollten dies *untere Zählergrenze*, *obere Zählergrenze* und *Schrittgröße* sein. Werden nur zwei Argumente über-

geben, so wird die Schrittgröße standardmäßig auf “1” gesetzt. Bei nur einem Argument, sei dies der *Startwert* des Zählers, die *Schrittgröße* sei “1”.

```
inlet_new(&x->x_obj, &x->x_obj.ob_pd,  
         gensym("list"), gensym("bound"));
```

Die Funktion `inlet_new` erzeugt einen neuen “aktiven” Inlet. “Aktiv” heißt, dass eine Klassenmethode ausgeführt wird, wenn eine Message in den einen “aktiven” Inlet geschickt wird.

Von der Software-Architektur her ist der erste Inlet immer “aktiv”.

Die ersten beiden Argumente der `inlet_new`-Funktion sind Zeiger auf die Objektinterna und die graphische Darstellung des Objektes.

Der symbolische Selector, der durch das dritte Argument spezifiziert wird, wird für diesen Inlet durch einen anderen symbolischen Selector (viertes Argument) substituiert.

Durch die Substitution von Selectoren kann eine Message an einem bestimmten rechten Eingang wie eine Message mit einem bestimmten Selector am linken Eingang betrachtet werden.

Dies bedeutet

- Der substituierende Selector muss mit `class_addmethod` angegeben werden.
- Man kann einen bestimmten rechten Eingang simulieren, indem man dem ersten Eingang eine Message mit dem Selector dieses Eingangs schickt.
- Es ist nicht möglich, einem rechten Eingang Methoden für mehr als einen Selector zuzuweisen. Insbesondere ist es nicht möglich, ihm eine allgemeine Methode für einen beliebigen Selector zuzuweisen.

```
floatinlet_new(&x->x_obj, &x->step);
```

`floatinlet_new` generiert einen “passiven” Inlet für numerische Werte. “Passive” Eingänge erlauben, dass ein Speicherplatz bestimmten Typs im Variablenraum des Objektes von außen direkt beschrieben werden kann. Dadurch ist zum Beispiel eine Abfrage nach illegalen Eingaben nicht möglich. Das erste Argument ist dabei ein Zeiger auf die interne Objektinfrastruktur. Das zweite Argument ist ein Zeiger auf den Speicherplatz, auf den geschrieben wird.

Es können “passive” Eingänge für numerische (Gleitkomma<sup>3</sup>)-Werte, symbolische Werte und Pointer geschaffen werden.

---

<sup>3</sup> Deswegen ist der `step`-Wert des Klassendatenraums als `t_float` realisiert.

```
x->f_out = outlet_new(&x->x_obj, &s_float);
x->b_out = outlet_new(&x->x_obj, &s_bang);
```

Die von `outlet_new` zurückgegebenen Zeiger auf die geschaffenen Outlets, müssen im Klassendatenraum gespeichert werden, damit sie später von den Ausgaberroutinen angesprochen werden.

Die Reihenfolge der Generierung von In- und Outlets ist wichtig, da sie der Reihenfolge der Ein- und Ausgänge der graphischen Repräsentation des Objektes entsprechen.

## 4.4 erweiterter Methodenraum

Der Methode für die “bang”-Message muss natürlich der komplexeren Zählerstruktur genüge tun.

```
void counter_bang(t_counter *x)
{
    t_float f=x->i_count;
    t_int step = x->step;
    x->i_count+=step;
    if (x->i_down-x->i_up) {
        if ((step>0) && (x->i_count > x->i_up)) {
            x->i_count = x->i_down;
            outlet_bang(x->b_out);
        } else if (x->i_count < x->i_down) {
            x->i_count = x->i_up;
            outlet_bang(x->b_out);
        }
    }
    outlet_float(x->f_out, f);
}
```

Die einzelnen Outlets werden von den `outlet_...`-Funktionen über die Zeiger auf diese Ausgänge identifiziert.

Die übrigen Methoden müssen noch implementiert werden:

```
void counter_reset(t_counter *x)
{
    x->i_count = x->i_down;
}

void counter_set(t_counter *x, t_floatarg f)
```

```

{
  x->i_count = f;
}

void counter_bound(t_counter *x, t_floatarg f1, t_floatarg f2)
{
  x->i_down = (f1<f2)?f1:f2;
  x->i_up   = (f1>f2)?f1:f2;
}

```

## 4.5 der Code: counter

```

#include "m_pd.h"

static t_class *counter_class;

typedef struct _counter {
  t_object  x_obj;
  t_int    i_count;
  t_float  step;
  t_int    i_down, i_up;
  t_outlet *f_out, *b_out;
} t_counter;

void counter_bang(t_counter *x)
{
  t_float f=x->i_count;
  t_int step = x->step;
  x->i_count+=step;

  if (x->i_down-x->i_up) {
    if ((step>0) && (x->i_count > x->i_up)) {
      x->i_count = x->i_down;
      outlet_bang(x->b_out);
    } else if (x->i_count < x->i_down) {
      x->i_count = x->i_up;
      outlet_bang(x->b_out);
    }
  }
}

outlet_float(x->f_out, f);

```

```

}

void counter_reset(t_counter *x)
{
    x->i_count = x->i_down;
}

void counter_set(t_counter *x, t_floatarg f)
{
    x->i_count = f;
}

void counter_bound(t_counter *x, t_floatarg f1, t_floatarg f2)
{
    x->i_down = (f1<f2)?f1:f2;
    x->i_up   = (f1>f2)?f1:f2;
}

void *counter_new(t_symbol *s, int argc, t_atom *argv)
{
    t_counter *x = (t_counter *)pd_new(counter_class);
    t_float f1=0, f2=0;

    x->step=1;
    switch(argc){
    default:
    case 3:
        x->step=atom_getfloat(argv+2);
    case 2:
        f2=atom_getfloat(argv+1);
    case 1:
        f1=atom_getfloat(argv);
        break;
    case 0:
    }
    if (argc<2)f2=f1;

    x->i_down = (f1<f2)?f1:f2;
    x->i_up   = (f1>f2)?f1:f2;

    x->i_count=x->i_down;
}

```

```

inlet_new(&x->x_obj, &x->x_obj.ob_pd,
          gensym("list"), gensym("bound"));
floatinlet_new(&x->x_obj, &x->step);

x->f_out = outlet_new(&x->x_obj, &s_float);
x->b_out = outlet_new(&x->x_obj, &s_bang);

return (void *)x;
}

void counter_setup(void) {
    counter_class = class_new(gensym("counter"),
                              (t_newmethod)counter_new,
                              0, sizeof(t_counter),
                              CLASS_DEFAULT,
                              A_GIMME, 0);

    class_addbang (counter_class, counter_bang);
    class_addmethod(counter_class,
                    (t_method)counter_reset, gensym("reset"), 0);
    class_addmethod(counter_class,
                    (t_method)counter_set, gensym("set"),
                    A_DEFFLOAT, 0);
    class_addmethod(counter_class,
                    (t_method)counter_bound, gensym("bound"),
                    A_DEFFLOAT, A_DEFFLOAT, 0);

    class_sethelpsymbol(counter_class, gensym("help-counter"));
}

```

## 5 ein Signal-External: pan~

Signalklassen sind normale Klassen, die zusätzlich Methoden für Signale bereitstellen.

Alle Methoden und Konzepte die mit normalen Objektklassen realisierbar sind, sind also auch mit Signalklassen zuverwirklichen.

Per Konvention enden die symbolischen Namen mit einer Tilde ~.

Anhand einer Klasse “pan~“ soll demonstriert werden wie Signalklassen geschrieben werden können.

Ein Signal am linken Inlet wird mit einem Signal am zweiten Inlet gemischt. Der Mischungsgrad wird als `t_float`-Message an einen dritten Eingang festgelegt.

## 5.1 Variablen einer Signalklasse

Da eine Signalklasse nur eine erweiterte normale Klasse ist, gibt es keine prinzipielle Unterschiede zwischen den Datenräumen.

```
typedef struct _pan_tilde {
    t_object x_obj;

    t_sample f_pan;
    t_float f;
} t_pan_tilde;
```

Es wird nur eine Variable für den *Mischfaktor* der Panningfunktion benötigt.

Die Variable `f` wird gebraucht, falls kein Signal am Signalinlet liegt. Wird dann an diesen Signalinlet ein numerischer Wert als Message geschickt, so ersetzt dieser das Signal und wird in der Variable `f` gespeichert.

## 5.2 Signalklassen

```
void pan_tilde_setup(void) {
    pan_tilde_class = class_new(gensym("pan~"),
        (t_newmethod)pan_tilde_new,
        0, sizeof(t_pan_tilde),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

    class_addmethod(pan_tilde_class,
        (t_method)pan_tilde_dsp, gensym("dsp"), 0);
    CLASS_MAINIGNALIN(pan_tilde_class, t_pan_tilde, f);
}
```

Jeder Signalklasse muss eine Methode für die Signalverarbeitung zugeordnet werden. Wenn die Audioengine von pd gestartet wird, wird allen Objekten eine Message mit dem Selector “`dsp`” geschickt. Alle Klassen, die eine Methode für die “`dsp`”-Message haben, sind Signalklassen.

Signalklassen, die Signal-Inlets zur Verfügung stellen wollen, müssen dies mit dem `CLASS_MAINIGNALIN`-Makro anmelden. Dadurch ist der erste Inlet

als Signalinlet deklariert. `t_float`-Messages können nicht mehr an einen solchen Eingang gesendet werden.

Das erste Argument des Makros ist ein Zeiger auf die Signalklasse. Das zweite Argument ist der Typ des Datenraums der Klasse. Das dritte Argument ist eine Dummy-Variable aus dem Datenraum, die gebraucht wird, um bei nicht vorhandenen Signalen am Signalinlet diese durch `t_float`-Messages einfach ersetzen zu können.

### 5.3 Konstruktion von Signal-In- und Outlets

```
void *pan_tilde_new(t_floatarg f)
{
    t_pan_tilde *x = (t_pan_tilde *)pd_new(pan_tilde_class);

    x->f_pan = f;

    inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
    floatinlet_new (&x->x_obj, &x->f_pan);

    outlet_new(&x->x_obj, &s_signal);

    return (void *)x;
}
```

Zusätzliche Signal-Eingänge werden normal mit der Routine `inlet_new` hinzugefügt. Die letzten beiden Argumente sind dann jeweils ein Verweis auf den symbolischen Selector “signal” in der lookup-Tabelle.

Signal-Outlets werden ebenfalls wie Message-Outlets generiert, deren Outlet mit dem Selector “signal” versehen ist.

### 5.4 DSP-Methode

Wenn die Audio-Engine von `pd` eingeschalten wird, so teilen ihr alle Signal-Objekte mit, welche Methode von ihrer Klasse zur digitalen Signalverarbeitung herangezogen werden soll.

Die “DSP”-Methode hat als Argumente einen Zeiger auf den Klassendatenraum und einen Zeiger auf ein Array von Signalen.

Die Signale im Array sind so angeordnet, dass sie am graphischen Objekt im Uhrzeigersinn gelesen werden.<sup>4</sup>

---

<sup>4</sup> Sofern linke und rechte Ein- und Ausgangssignale vorhanden sind, gilt also: Zuerst

```

void pan_tilde_dsp(t_pan_tilde *x, t_signal **sp)
{
    dsp_add(pan_tilde_perform, 5, x,
            sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec, sp[0]->s_n);
}

```

`dsp_add` fügt eine “Perform”-Routine (erstes Argument) zum DSP-Baum hinzu. Das zweite Argument ist die Anzahl der nachfolgenden Zeiger auf diverse Variablen. Welche Zeiger auf welche Variablen übergeben werden, unterliegt keiner Beschränkung.

`sp[0]` bezeichnet hier das erste Eingangssignal, `sp[1]` das zweite Eingangssignal, `sp[3]` das Ausgangssignal.

Die Struktur `t_signal` enthält einen Zeiger auf den zugehörigen Signalvektor `.s_vec` (ein Array von Samples `t_sample`), sowie die Länge dieses Signalvektors `.s_n`. Da innerhalb eines Patches alle Signalvektoren die gleiche Länge haben, genügt es, die Länge eines dieser Vektoren abzufragen.

## 5.5 perform-Routine

Die `perform`-Routine ist das eigentliche DSP-Herzstück einer Signalklasse.

Ihr wird ein Zeiger auf ein Integer-Array übergeben. In diesem Array sind die Zeiger gespeichert, die mit `dsp_add` übergeben wurden. Sie müssen auf ihren ursprünglichen Typ zurückgecastet werden.

Die `perform`-Routine muß einen Zeiger auf Integer zurückgeben, der hinter den Speicherplatz zeigt, in dem die eigenen Zeiger gespeichert sind. Dies bedeutet, dass das Rückgabeargument gleich dem Übergabeargument plus der Anzahl der eigenen Zeigervariablen (wie sie als zweites Argument in `dsp_add` angegeben wurde) plus eins.

```

t_int *pan_tilde_perform(t_int *w)
{
    t_pan_tilde *x = (t_pan_tilde *) (w[1]);
    t_sample *in1 = (t_sample *) (w[2]);
    t_sample *in2 = (t_sample *) (w[3]);
    t_sample *out = (t_sample *) (w[4]);
    int n = (int) (w[5]);

    t_sample f_pan = (x->f_pan<0)?0.0:(x->f_pan>1)?1.0:x->f_pan;

```

---

kommt das linke Eingangssignal, danach die rechten Eingangssignale; nach den rechten Ausgangssignalen kommt das linke Ausgangssignal.

```

while (n--) *out++ = (*in1++)*(1-f_pan)+(*in2++)*f_pan;

return (w+6);
}

```

In der while-Schleife wird jedes Sample der Signalvektoren einzeln abgearbeitet.

Eine Optimierungsroutine bei der Erstellung des DSP-Baumes wird darauf geachtet, keine unnötigen Kopieroperationen durchzuführen. Es kann daher geschehen, dass ein Eingangs- und ein Ausgangssignal an der gleichen Stelle im Speicher stehen. Es ist daher in solchem Falle darauf zu achten, dass nicht in das Ausgangssignal geschrieben wird, bevor dort das Eingangssignal ausgelesen wurde.

## 5.6 der Code: pan~

```

#include "m_pd.h"

static t_class *pan_tilde_class;

typedef struct _pan_tilde {
    t_object x_obj;
    t_sample f_pan;
    t_sample f;
} t_pan_tilde;

t_int *pan_tilde_perform(t_int *w)
{
    t_pan_tilde *x = (t_pan_tilde *) (w[1]);
    t_sample *in1 = (t_sample *) (w[2]);
    t_sample *in2 = (t_sample *) (w[3]);
    t_sample *out = (t_sample *) (w[4]);
    int n = (int) (w[5]);
    t_sample f_pan = (x->f_pan < 0) ? 0.0 : (x->f_pan > 1) ? 1.0 : x->f_pan;

    while (n--) *out++ = (*in1++)*(1-f_pan)+(*in2++)*f_pan;

    return (w+6);
}

void pan_tilde_dsp(t_pan_tilde *x, t_signal **sp)

```

```

{
    dsp_add(pan_tilde_perform, 5, x,
            sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec, sp[0]->s_n);
}

void *pan_tilde_new(t_floatarg f)
{
    t_pan_tilde *x = (t_pan_tilde *)pd_new(pan_tilde_class);

    x->f_pan = f;

    inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
    floatinlet_new (&x->x_obj, &x->f_pan);
    outlet_new(&x->x_obj, &s_signal);

    return (void *)x;
}

void pan_tilde_setup(void) {
    pan_tilde_class = class_new(gensym("pan~"),
        (t_newmethod)pan_tilde_new,
        0, sizeof(t_pan_tilde),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

    class_addmethod(pan_tilde_class,
        (t_method)pan_tilde_dsp, gensym("dsp"), 0);
    CLASS_MAINSIGNALIN(pan_tilde_class, t_pan_tilde, f);
}

```

## A das Message-System von *pd*

Nicht-Audio-Daten werden über ein Message-System verteilt. Jede Message besteht aus einem “Selector” und einer Liste von Atomen.

### A.1 Atome

Es gibt drei Arten von Atomen:

- *A\_FLOAT*: ein numerischer Wert (Gleitkommazahl)
- *A\_SYMBOL*: ein symbolischer Wert (String)
- *A\_POINTER*: ein Zeiger

Numerische Werte werden immer als Floating-Point-Werte (`double`) dargestellt, auch wenn es sich um Ganzzahlwerte handelt.

Jedes Symbol wird aus Performancegründen in einer lookup-Tabelle abgelegt. Der Befehl `gensym` speichert, wenn nötig, einen String in dieser Symboltabelle und gibt seine Adresse in der Tabelle zurück.

Atome vom Typ *A\_POINTER* haben in der Praxis (für einfache Externals) eher untergeordnete Bedeutung.

Der Typ eines Atoms `a` wird im Strukturelement `a.a_type` gespeichert.

### A.2 Selectoren

Der Selector ist ein Symbol und bestimmt, welchen Typ eine Message hat. Es gibt fünf vordefinierte Selectoren:

- “`bang`” bezeichnet ein Triggerevent. Die Message besteht nur aus dem Selector und enthält keine Liste von Atomen.
- “`float`” bezeichnet einen numerischen Wert. Die Liste enthält nur ein Atom.
- “`symbol`” bezeichnet einen symbolischen Wert. Die Liste enthält nur ein Atom.
- “`pointer`” bezeichnet einen Zeiger. Die Liste enthält nur ein Atom.
- “`list`” bezeichnet eine Liste von mehreren Atomen.

Da die Symbole für diese Selectoren relativ häufig verwendet werden, kann man deren Symboltabellen-Adresse auch direkt, ohne den Umweg über `gensym` abfragen:

Selector	lookup-Routine	lookup-Adresse
<code>bang</code>	<code>gensym("bang")</code>	<code>&amp;s_bang</code>
<code>float</code>	<code>gensym("float")</code>	<code>&amp;s_float</code>
<code>symbol</code>	<code>gensym("symbol")</code>	<code>&amp;s_symbol</code>
<code>pointer</code>	<code>gensym("pointer")</code>	<code>&amp;s_pointer</code>
<code>list</code>	<code>gensym("list")</code>	<code>&amp;s_list</code>
<code>— (Signal)</code>	<code>gensym("signal")</code>	<code>&amp;s_symbol</code>

Es können auch andere Selectoren verwendet werden, doch muss dann die Empfängerklasse entweder selbst eine Methode für diesen Selector zur Verfügung stellen, oder eine Methode für “anything”, also jeden beliebigen Selector, anbieten.

Messages die ohne Selector sofort mit einem Zahlenwert beginnen, werden automatisch entweder als numerischer Wert (nur ein Atom) oder als Liste (mehrere Atome) erkannt.

Zum Beispiel sind also die Messages “12.429” und “float 12.429” ident. Ebenfalls ident sind auch die Listen-Messages “list 1 kleines Haus” und “1 kleines Haus”.

## B pd-Typen

Da pd auf mehreren Plattformen benutzt wird, werden viele gewöhnliche Variablentypen, wie `int`, neu definiert. Um portablen Code zu schreiben ist es daher angebracht, die von pd bereitgestellten Typen zu verwenden.

Weiters gibt es viele vordefinierte Typen, die das Leben des Programmierers vereinfachen sollten. pd-Typen beginnen im Allgemeinen mit `t_`.

pd-Type	Beschreibung
<code>t_atom</code>	Atom
<code>t_float</code>	Gleitkomma-Zahl
<code>t_symbol</code>	Symbol
<code>t_gpointer</code>	Zeiger (auf graphische Objekte)
<code>t_int</code>	Ganzzahl
<code>t_signal</code>	Struktur auf ein Signal
<code>t_sample</code>	Audio-Signalwert (Gleitkomma)
<code>t_outlet</code>	Outlet eines Objekts
<code>t_inlet</code>	Inlet eines Objekts
<code>t_object</code>	Objekt-Interna
<code>t_class</code>	eine pd-Klasse
<code>t_method</code>	Zeiger auf Klassenmethode
<code>t_newmethod</code>	Zeiger auf Klasseninstanzierungsmethode (new-Routine)

## C Wichtige Funktionen aus “m\_pd.h”

### C.1 Funktionen: Atome

#### C.1.1 SETFLOAT

`SETFLOAT(atom, f)`

Dieses Makro setzt den Typ von `atom` auf `A_FLOAT` und setzt den numerischen Wert dieses Atoms auf `f`.

#### C.1.2 SETSYMBOL

`SETSYMBOL(atom, s)`

Dieses Makro setzt den Typ von `atom` auf `A_SYMBOL` und setzt den symbolischen Wert dieses Atoms auf `s`.

#### C.1.3 SETPOINTER

`SETPOINTER(atom, pt)`

Dieses Makro setzt den Typ von `atom` auf `A_POINTER` und setzt den Zeiger-Wert dieses Atoms auf `pt`.

#### C.1.4 atom\_getfloat

```
t_float atom_getfloat(t_atom *a);
```

Wenn der Typ des Atoms a A\_FLOAT ist, wird dessen numerischer Wert, ansonsten "0.0" zurückgegeben.

#### C.1.5 atom\_getfloatarg

```
t_float atom_getfloatarg(int which, int argc, t_atom *argv)
```

Wenn das Atom, das in der Atomliste argv mit der Länge argc an der Stelle which zu finden ist, vom Typ A\_FLOAT ist, wird dessen numerischer Wert, ansonsten "0.0" zurückgegeben.

#### C.1.6 atom\_getint

```
t_int atom_getint(t_atom *a);
```

Wenn der Typ des Atoms a A\_FLOAT ist, wird dessen numerischer Wert als Ganzzahlwert, ansonsten "0" zurückgegeben.

#### C.1.7 atom\_getsymbol

```
t_symbol atom_getsymbol(t_atom *a);
```

Wenn der Typ des Atoms a A\_SYMBOL ist, wird ein Zeiger auf dessen Symbol ansonsten auf das Symbol "float" zurückgegeben.

#### C.1.8 atom\_gensym

```
t_symbol *atom_gensym(t_atom *a);
```

Wenn der Typ des Atoms a A\_SYMBOL ist, wird ein Zeiger auf dessen Symbol zurückgegeben.

Atome anderen Typs werden zuerst "sinnvoll" in Strings umgewandelt. Diese Strings werden, falls nötig, in die Symbol-Tabelle eingetragen. Die Zeiger auf das Symbol wird zurückgegeben.

#### C.1.9 atom\_string

```
void atom_string(t_atom *a, char *buf, unsigned int bufsize);
```

Konvertiert ein Atom a in einen C-String buf. Der char-Buffer muss selbst reserviert und seine Länge in bufsize angegeben werden.

### C.1.10 gensym

```
t_symbol *gensym(char *s);
```

Prüft, ob für den C-String `*s` bereits ein Eintrag in der Symbol-lookup-Tabelle vorhanden ist. Ist noch kein Eintrag vorhanden, so wird einer angelegt. Ein Zeiger auf das Symbol in der Tabelle wird zurückgegeben.

## C.2 Funktionen: Klassen

### C.2.1 class\_new

```
t_class *class_new(t_symbol *name,  
                  t_newmethod newmethod, t_method freemethod,  
                  size_t size, int flags,  
                  t_atomtype arg1, ...);
```

Generiert eine neue Klasse mit dem symbolischen Namen `name`.

`newmethod` ist eine Konstruktorfunktion, die eine Instanz der Klasse konstruiert und einen Zeiger auf diese Instanz zurückgibt.

Wird manuell dynamischer Speicher reserviert, so muss dieser bei Zerstörung eines Objektes mit der Destruktormethode `freemethod` (kein Rückgabeargument) wieder freigegeben werden.

`size` ist statische die Größe des Klassendatenraumes, die mit der Funktion `sizeof(t_mydata)` berechnet werden kann.

`flags` bestimmen das Aussehen des graphischen Objektes. Eine beliebige Kombination folgender Flags ist möglich:

Flag	Bedeutung
CLASS_DEFAULT	Ein normales Objekt mit einem Inlet
CLASS_PD	<i>Objekte ohne Graphikdarstellung</i>
CLASS_GOBJ	<i>reine Graphikobjekte (wie Arrays, Graphen,...)</i>
CLASS_PATCHABLE	<i>normales Objekt (mit einem Inlet)</i>
CLASS_NOINLET	Der standardmäßige Inlet wird unterdrückt

Flags, deren Bedeutung *kursiv* gedruckt ist, haben geringe Bedeutung beim Schreiben von Externals.

Die restlichen Argumente `arg1, ...` definieren die Typen die Übergabeargumente bei der Objektkreation. Höchstens sechs typgeprüfte Argumente können einem Objekt übergeben werden. Die Argumententypeliste wird "0" terminiert.

Mögliche Argumententypen sind:

A_DEFFLOAT	ein numerischer Wert
A_DEFSYMBOL	ein symbolischer Wert
A_GIMME	eine Atomliste beliebiger Länge und Typen

Sollten mehr als sechs Argumente übergeben werden, muss man `A_GIMME` verwenden und eine händische Typprüfung durchführen.

### C.2.2 `class_addmethod`

```
void class_addmethod(t_class *c, t_method fn, t_symbol *sel,  
    t_atomtype arg1, ...);
```

Fügt der Klasse, auf die `c` zeigt, die Methode `fn` für eine Message mit dem Selector `sel` hinzu.

Die restlichen Argumente `arg1,...` definieren die Typen der Atomliste die dem Selector folgt. Höchstens sechs typgeprüfte Argumente angegeben werden. Sollten mehr als sechs Argumente übergeben werden, muss man `A_GIMME` verwenden und eine händische Typprüfung durchführen.

Die Argumententypeliste wird "0" terminiert.

Mögliche Argumententypen sind:

<code>A_DEFFLOAT</code>		ein numerischer Wert
<code>A_DEFSYMBOL</code>		ein symbolischer Wert
<code>A_POINTER</code>		eine Zeiger
<code>A_GIMME</code>		eine Atomliste beliebiger Länge und Typen

### C.2.3 `class_addbang`

```
void class_addbang(t_class *c, t_method fn);
```

Fügt der Klasse, auf die `c` zeigt, die Methode `fn` für eine "bang"-Message hinzu. Die "bang"-Methode hat als Übergabeargument einen Zeiger auf den Klassendatenraum:

```
void my_bang_method(t_mydata *x);
```

### C.2.4 `class_addfloat`

```
void class_addfloat(t_class *c, t_method fn);
```

Fügt der Klasse, auf die `c` zeigt, die Methode `fn` für eine "float"-Message hinzu. Die "float"-Methode hat als Übergabeargument einen Zeiger auf den Klassendatenraum und ein Gleitkommaargument:

```
void my_float_method(t_mydata *x, t_floatarg f);
```

### C.2.5 class\_addsymbol

```
void class_addsymbol(t_class *c, t_method fn);
```

Fügt der Klasse, auf die `c` zeigt, die Methode `fn` für eine “symbol”-Message hinzu. Die “symbol”-Methode hat als Übergabeargument einen Zeiger auf den Klassendatenraum und einen Zeiger auf das übergebene Symbol:

```
void my_symbol_method(t_mydata *x, t_symbol *s);
```

### C.2.6 class\_addpointer

```
void class_addpointer(t_class *c, t_method fn);
```

Fügt der Klasse, auf die `c` zeigt, die Methode `fn` für eine “pointer”-Message hinzu. Die “pointer”-Methode hat als Übergabeargument einen Zeiger auf den Klassendatenraum und einen Zeiger auf einen Pointer:

```
void my_pointer_method(t_mydata *x, t_gpointer *pt);
```

### C.2.7 class\_addlist

```
void class_addlist(t_class *c, t_method fn);
```

Fügt der Klasse, auf die `c` zeigt, die Methode `fn` für eine “list”-Message hinzu. Die “list”-Methode hat als Übergabeargument neben einem Zeiger auf den Klassendatenraum einen Zeiger auf das Selectorsymbol (immer `&s_list`), die Anzahl der Atome in der Liste sowie einen Zeiger auf die Atomliste:

```
void my_list_method(t_mydata *x,  
t_symbol *s, int argc, t_atom *argv);
```

### C.2.8 class\_addanything

```
void class_addanything(t_class *c, t_method fn);
```

Fügt der Klasse, auf die `c` zeigt, die Methode `fn` für eine beliebige Message hinzu. Die anything-Methode hat als Übergabeargument neben einem Zeiger auf den Klassendatenraum einen Zeiger auf das Selectorsymbol, die Anzahl der Atome in der Liste sowie einen Zeiger auf die Atomliste:

```
void my_any_method(t_mydata *x,  
t_symbol *s, int argc, t_atom *argv);
```

### C.2.9 class\_addcreator

```
void class_addcreator(t_newmethod newmethod, t_symbol *s,  
    t_atomtype type1, ...);
```

Fügt zu einem Konstruktor `newmethod` ein zum Klassennamen alternatives Kreatorsymbol `s` hinzu. Dadurch können Objekte mit dem richtigen Klassennamen und einem Aliasnamen (zum Beispiel eine Abkürzung, wie das Internal “float” bzw. “f”) kreiert werden.

Die “0”-terminierte Typenliste entspricht der von `class_new`.

### C.2.10 class\_sethelpsymbol

```
void class_sethelpsymbol(t_class *c, t_symbol *s);
```

Clickt man mit der rechten Maustaste auf ein pd-Objekt, so kann man sich einen Hilfe-Patch für die zugehörige Objektklasse anzeigen lassen. Standardmäßig wird ist dies ein Patch mit dem symbolischen Klassennamen im Verzeichnis “*doc/5.reference/*” gesucht.

Für die Klasse, auf die `c` zeigt, wird der Name des Hilfepatches auf den symbolischen Wert `s` geändert.

Dadurch können sich mehrere verwandte Klassen einen Hilfepatch teilen.

Pfadangaben erfolgen relativ zum Standardhilfepfad *doc/5.reference/*.

### C.2.11 pd\_new

```
t_pd *pd_new(t_class *cls);
```

Generiert eine neue Instanz der Klasse `cls` und gibt einen Zeiger auf diese Instanz zurück.

## C.3 Funktionen: In- und Outlets

Alle Inlet- und Outletroutinen benötigen eine Referenz auf die Objektinterna der Klasseninstanz. Die notwendige Variable vom Typ `t_object` im Datenraum wird bei der Objektinstanziierung initialisiert. Diese Variable muß als `owner`-Objekt den Inlet- und Outletroutinen übergeben werden.

### C.3.1 inlet\_new

```
t_inlet *inlet_new(t_object *owner, t_pd *dest,  
    t_symbol *s1, t_symbol *s2);
```

Generiert einen zusätzlichen “aktiven” Inlet des Objektes, auf das `owner` zeigt. `dest` zeigt im Allgemeinen auf “`owner.ob_pd`”.

Der Selector `s1` am neuen Inlet, wird durch den Selector `s2` substituiert.

Tritt also eine Message mit dem Selector `s1` am neuen Inlet auf, wird die Klassenmethode für den Selector `s2` ausgeführt.

Dies bedeutet

- Der substituierende Selector muss mit `class_addmethod` angegeben werden.
- Man kann einen bestimmten rechten Eingang simulieren, indem man dem ersten Eingang eine Message mit dem Selector dieses Eingangs schickt.

Verwendet man ein leeres Symbol (`gensym("")`) als Selector, so erreicht man, dass der rechte Eingang nicht über den ersten angesprochen werden kann.

- Es ist nicht möglich, einem rechten Eingang Methoden für mehr als einen Selector zuzuweisen. Insbesondere ist es nicht möglich, ihm eine allgemeine Methode für einen beliebigen Selector zuzuweisen.

### C.3.2 floatinlet\_new

```
t_inlet *floatinlet_new(t_object *owner, t_float *fp);
```

Schafft einen neuen “passiven” Eingang für das Objekt, auf das `owner` zeigt, der es erlaubt, einen numerischen Wert von außen direkt auf einen Speicherplatz `fp` zu schreiben, ohne eine eigene Methode aufzurufen.

### C.3.3 symbolinlet\_new

```
t_inlet *symbolinlet_new(t_object *owner, t_symbol **sp);
```

Schafft einen neuen “passiven” Eingang für das Objekt, auf das `owner` zeigt, der es erlaubt, einen symbolischen Wert von außen direkt auf einen Speicherplatz `sp` zu schreiben, ohne eine eigene Methode aufzurufen.

### C.3.4 pointerinlet\_new

```
t_inlet *pointerinlet_new(t_object *owner, t_gpointer *gp);
```

Schafft einen neuen “passiven” Eingang für das Objekt, auf das `owner` zeigt, der es erlaubt, einen Zeigerwert von außen direkt auf einen Speicherplatz `gp` zu schreiben, ohne eine eigene Methode aufzurufen.

### C.3.5 outlet\_new

```
t_outlet *outlet_new(t_object *owner, t_symbol *s);
```

Generiert einen neuen Ausgang für das Objekt, auf das `owner` zeigt. Das Symbol, auf das `s` zeigt, zeigt den Typ des Ausgangs an.

Symbolwert	Symboladresse	Outlet-Typus
“bang”	<code>&amp;s_bang</code>	Message (Bang)
“float”	<code>&amp;s_float</code>	Message (Float)
“symbol”	<code>&amp;s_symbol</code>	Message (Symbol)
“pointer”	<code>&amp;s_gpointer</code>	Message (List)
“list”	<code>&amp;s_list</code>	Message
—	0	Message
“signal”	<code>&amp;s_signal</code>	Signal

Zwischen den verschiedenen Message-Outlet-Typen gibt es keinen Unterschied. Allerdings macht es den Code leichter lesbar, wenn schon bei der Outlet-Generierung angezeigt wird, wozu der Ausgang verwendet wird. Für allgemeine Message-Outlets verwendet man einen “0”-Pointer.

Variablen vom Typ `t_object` stellen einen Zeiger auf einen Outlet zur Verfügung. Bei der Generierung eines neuen Outlets, wird seine Adresse in der Objektvariablen `(*owner).ob_outlet` gespeichert.

Werden mehrere Message-Ausgänge benötigt, müssen die Outletzeiger, die von `outlet_new` zurückgegeben werden, manuell im Datenraum gespeichert werden, um die jeweiligen Ausgänge ansprechen zu können.

### C.3.6 outlet\_bang

```
void outlet_bang(t_outlet *x);
```

Gibt am Outlet, auf den `x` zeigt, eine “bang”-Message aus.

### C.3.7 outlet\_float

```
void outlet_float(t_outlet *x, t_float f);
```

Gibt am Outlet, auf den `x` zeigt, eine “float”-Message mit dem numerischen Wert `f` aus.

### C.3.8 outlet\_symbol

```
void outlet_symbol(t_outlet *x, t_symbol *s);
```

Gibt am Outlet, auf den `x` zeigt, eine “symbol”-Message mit dem symbolischen Wert von `s` aus.

### C.3.9 outlet\_pointer

```
void outlet_pointer(t_outlet *x, t_gpointer *gp);
```

Gibt am Outlet, auf den `x` zeigt, eine “pointer”-Message mit dem Zeiger `gp` aus.

### C.3.10 outlet\_list

```
void outlet_list(t_outlet *x,  
                t_symbol *s, int argc, t_atom *argv);
```

Gibt am Outlet, auf den `x` zeigt, eine “list”-Message mit `argc` Atomen aus. `argv` zeigt auf das erste Atom der Liste.

Unabhängig davon, auf welches Symbol `s` zeigt, wird der Selector “list” der Liste vorangestellt.

Aus Lesbarkeitsgründen sollte man aber trotzdem einen Zeiger auf das Symbol “list” (`gensym("list")` oder `&s_list`) angeben.

### C.3.11 outlet\_anything

```
void outlet_anything(t_outlet *x,  
                    t_symbol *s, int argc, t_atom *argv);
```

Gibt am Outlet, auf den `x` zeigt, eine Message mit dem Selector, auf den `s` zeigt, aus. Dem Selector folgen `argc` Atome. `argv` zeigt auf das erste Atom dieser Liste.

## C.4 Funktionen: DSP

Soll eine Klasse Methoden zur digitalen Signalverarbeitung zur Verfügung stellen, so muss ihr eine Methode für den Selector “dsp” hinzugefügt werden.

Wird die Audio-Engine gestartet, so werden alle Objekte, die eine “dsp”-Methode zur Verfügung stellen, als Instanzen von Signalklassen identifiziert.

### DSP-Methode

```
void my_dsp_method(t_mydata *x, t_signal **sp)
```

In der “dsp”-Methode wird mit der Funktion `dsp_add` die Klassenroutine für Signalverarbeitung in den DSP-Baum eingebunden.

Neben dem eigenen Datenraum `x`, wird auch ein Array von Signalen übergeben. Die Signale im Array sind so angeordnet, dass sie am graphischen Objekt im Uhrzeigersinn gelesen werden.

Sofern je zwei Ein- und Ausgangssignale vorhanden sind, gilt also:

Zeiger	auf Signal
sp[0]	linkes Eingangssignal
sp[1]	rechtes Eingangssignal
sp[2]	rechtes Ausgangssignal
sp[3]	linkes Ausgangssignal

Die Signalstruktur enthält unter anderem:

Strukturelement	Bedeutung
s_n	Länge des Signalvektors
s_vec	Zeiger auf den Signalvektor

Der Signalvektor ist ein Array auf Samples vom Typ `t_sample`.

## Perform-Routine

```
t_int *my_perform_routine(t_int *w)
```

Der Perform-Routine die mit `class_add` in den DSP-Baum eingefügt wurde, wird ein Zeiger `w` auf ein (Integer-)Array übergeben. In diesem Array sind die Zeiger gespeichert, die mit `dsp_add` übergeben wurden. Sie müssen auf ihren ursprünglichen Typ zurückgecastet werden. Der erste Zeiger ist an der Stelle `w[1]` gespeichert !!!

Die perform-Routine muß einen Zeiger auf Integer zurückgeben, der hinter den Speicherplatz zeigt, in dem die eigenen Zeiger gespeichert sind. Dies bedeutet, dass das Rückgabeargument gleich dem Übergabeargument plus der Anzahl der eigenen Zeigervariablen (wie sie als zweites Argument in `dsp_add` angegeben wurde) plus eins.

### C.4.1 CLASS\_MAINSIGNALIN

```
CLASS_MAINSIGNALIN(<class_name>, <class_data>, <f>);
```

Das Makro `CLASS_MAINSIGNALIN` meldet an, dass die Klasse Signal-Inlets braucht.

Das erste Argument des Makros ist ein Zeiger auf die Signalklasse. Das zweite Argument ist der Typ des Datenraums der Klasse. Das dritte Argument ist eine (Dummy-)Gleitkomma-Variable aus dem Datenraum, die gebraucht wird, um bei nicht vorhandenen Signalen am Signalinlet, "float"-Messages wie Signale behandeln zu können.

An so kreierten Signaleingängen können daher keine zusätzlichen "float"-Messages geschickt werden.

### C.4.2 dsp\_add

```
void dsp_add(t_perfroutine f, int n, ...);
```

Fügt dem DSP-Baum eine Perform-Routine `f` hinzu, die jeden DSP-Zyklus neu aufgerufen wird.

Das zweite Argument `n` legt die Anzahl der nachfolgenden Zeigerargumente fest.

Welche Zeiger auf welche Variablen übergeben werden, unterliegt keiner Beschränkung. Sinnvoll sind im Allgemeinen Zeiger auf den Datenraum und auf die Signalvektoren. Auch die Länge der Signalvektoren sollte übergeben werden, um effektiv Signale manipulieren zu können.

### C.4.3 sys\_getsr

```
float sys_getsr(void);
```

Gibt die Abtastrate des Systems zurück.

## C.5 Funktion: Memory

### C.5.1 getbytes

```
void *getbytes(size_t nbytes);
```

Reserviert `nbytes` Bytes und gibt einen Zeiger auf den reservierten Speicher zurück.

### C.5.2 copybytes

```
void *copybytes(void *src, size_t nbytes);
```

Kopiert `nbytes` Bytes von `*src` in einen neu allozierten Speicher. Die Adresse dieses Speichers wird zurückgegeben.

### C.5.3 freebytes

```
void freebytes(void *x, size_t nbytes);
```

Gibt `nbytes` Bytes an der Adresse `*x` frei.

## C.6 Funktionen: Ausgabe

### C.6.1 post

```
void post(char *fmt, ...);
```

Schreibt einen C-String auf den Standarderror (Shell).

### C.6.2 error

```
void error(char *fmt, ...);
```

Schreibt einen C-String als Fehlermeldung auf den Standarderror (Shell). Das Objekt, das die Fehlermeldung ausgegeben hat, wird markiert und ist über das pd-Menü *Find->Find last error* identifizierbar.