A Pd's message-system

Non-audio-data are distributed via a message-system. Each message consists of a "selector" and a list of atoms.

A.1 atoms

There are three kinds of atoms:

- *A_FLOAT*: a numerical value (floating point)
- *A_SYMBOL*: a symbolic value (string)
- *A_POINTER*: a pointer

Numerical values are always floating point-values (`t_float`), even if they could be displayed as integer values.

Each symbol is stored in a lookup-table for reasons of performance. The command `gensym` looks up a string in the lookup-table and returns the address of the symbol. If the string is not yet to be found in the table, a new symbol is added.

Atoms of type *A_POINTER* are not very important (for simple externals).

The type of an atom `a` is stored in the structure-element `a.a_type`.

A.2 selectors

The selector is a symbol that defines the type of a message. There are five predefined selectors:

- "`bang`" labels a trigger event. A "bang"-message consists only of the selector and contains no lists of atoms.
- "`float`" labels a numerical value. The list of a "float"-Message contains one single atom of type *A_FLOAT*
- "`symbol`" labels a symbolic value. The list of a "symbol"-Message contains one single atom of type *A_SYMBOL*
- "`pointer`" labels a pointer value. The list of a "pointer"-Message contains one single atom of type *A_POINTER*
- "`list`" labels a list of one or more atoms of arbitrary type.

Since the symbols for these selectors are used quite often, their address in the lookup-table can be queried directly, without having to use `gensym`:

selector	lookup-routine	lookup-address
`bang`	`gensym("bang")`	`&s_bang`
`float`	`gensym("float")`	`&s_float`
`symbol`	`gensym("symbol")`	`&s_symbol`
`pointer`	`gensym("pointer")`	`&s_pointer`
`list`	`gensym("list")`	`&s_list`
— (signal)	`gensym("signal")`	`&s_symbol`

Other selectors can be used as well. The receiving class has to provide a method for a specifique selector or for "anything", which is any arbitrary selector.

Messages that have no explicit selector and start with a numerical value, are recognised automatically either as "float"-message (only one atom) or as "list"-message (several atoms).

For example, messages "`12.429`" and "`float 12.429`" are identical. Likewise, the messages "`list 1 for you`" is identical to "`1 for you`".

B Pd-types

Since Pd is used on several platforms, many ordinary types of variables, like `int`, are re-defined. To write portable code, it is reasonable to use types provided by Pd.

Apart from this there are many predefined types, that should make the life of the programmer simpler.

Generally, Pd-types start with `t_`.

Pd-type	description
`t_atom`	atom
`t_float`	floating point value
`t_symbol`	symbol
`t_gpointer`	pointer (to graphical objects)
`t_int`	integer value
`t_signal`	structure of a signal
`t_sample`	audio signal-value (floating point)
`t_outlet`	outlet of an object
`t_inlet`	inlet of an object
`t_object`	object-interna
`t_class`	a Pd-class
`t_method`	class-method
`t_newmethod`	pointer to a constructor (new-routine)

C important functions in "m_pd.h"

C.1 functions: atoms

C.1.1 SETFLOAT

```
SETFLOAT(atom, f)
```

This macro sets the type of `atom` to `A_FLOAT` and stores the numerical value `f` in this atom.

C.1.2 SETSYMBOL

```
SETSYMBOL(atom, s)
```

This macro sets the type of `atom` to `A_SYMBOL` and stores the symbolic pointer `s` in this atom.

C.1.3 SETPOINTER

```
SETPOINTER(atom, pt)
```

This macro sets the type of `atom` to `A_POINTER` and stores the pointer `pt` in this atom.

C.1.4 atom_getfloat

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

If the type of the atom `a` is `A_FLOAT`, the numerical value of this atom else "0.0" is returned.

C.1.5 atom_getfloatarg

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

If the type of the atom – that is found at in the atom-list `argv` with the length `argc` at the place `which` – is `A_FLOAT`, the numerical value of this atom else "0.0" is returned.

C.1.6 atom_getint

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

If the type of the atom `a` is `A_FLOAT`, its numerical value is returned as integer else "0" is returned.

C.1.7 atom_getsymbol

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

If the type of the atom `a` is `A_SYMBOL`, a pointer to this symbol is returned, else a null-pointer "0" is returned.

C.1.8 atom_gensym

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

If the type of the atom `a` is `A_SYMBOL`, a pointer to this symbol is returned.

Atoms of a different type, are "reasonably" converted into a string. This string is – on demand – inserted into the symbol-table. A pointer to this symbol is returned.

C.1.9 atom_string

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

Converts an atom `a` into a `C`-string `buf`. The memory to this char-Buffer has to be reserved manually and its length has to be declared in `bufsize`.

C.1.10 gensym

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

Checks, whether the C-string `*s` has already been inserted into the symbol-table. If no entry exists, it is created. A pointer to the symbol is returned.

C.2 functions: classes

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, ...);
```

Generates a class with the symbolic name `name`. `newmethod` is the constructor that creates an instance of the class and returns a pointer to this instance.

If memory is reserved dynamically, this memory has to be freed by the destructor-method `freemethod` (without any return argument), when the object is destroyed.

`size` is the static size of the class-data space, that is returned by `sizeof(t_mydata)`.

`flags` define the presentation of the graphical object. A (more or less arbitrary) combination of following objects is possible:

flag	description
`CLASS_DEFAULT`	a normal object with one inlet
`CLASS_PD`	*object (without graphical presentation)*
`CLASS_GOBJ`	*pure graphical object (like arrays, graphs,...)*
`CLASS_PATCHABLE`	*a normal object (with one inlet)*
`CLASS_NOINLET`	the default inlet is suppressed

Flags the description of which is printed in *italic* are of small importance for writing externals.

The remaining arguments `arg1,...` define the types of object-arguments passed at the creation of a class-object. A maximum of six type checked arguments can be passed to an object. The list of argument-types are terminated by "0".

Possible types of arguments are:

`A_DEFFLOAT`	a numerical value
`A_DEFSYMBOL`	a symbolical value
`A_GIMME`	a list of atoms of arbitrary length and types

If more than six arguments are to be passed, `A_GIMME` has to be used and a manual type-check has to be made.

C.2.2 class_addmethod

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

Adds a method `fn` for a selector `sel` to a class `c`.

The remaining arguments `arg1,...` define the types of the list of atoms that follow the selector. A maximum of six type-checked arguments can be passed. If more than six arguments are to be passed, `A_GIMME` has to be used and a manual type-check has to be made.

The list of arguments is terminated by "0".

Possible types of arguments are:

`A_DEFFLOAT`	a numerical value
`A_DEFSYMBOL`	a symbolical value
`A_POINTER`	a pointer
`A_GIMME`	a list of atoms of arbitrary length and types

C.2.3 class_addbang

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

Adds a method `fn` for "bang"-messages to the class `c`.

The argument of the "bang"-method is a pointer to the class-data space:

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

C.2.4 class_addfloat

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

Adds a method `fn` for "float"-messages to the class `c`.

The arguments of the "float"-method is a pointer to the class-data space and a floating point-argument:

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

C.2.5 class_addsymbol

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

Adds a method `fn` for "symbol"-messages to the class `c`.

The arguments of the "symbol"-method is a pointer to the class-data space and a pointer to the passed 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);
```

Adds a method `fn` for "pointer"-messages to the class `c`.

The arguments of the "pointer"-method is a pointer to the class-data space and a pointer to a 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);
```

Adds a method `fn` for "list"-messages to the class `c`.

The arguments of the "list"-method are – apart from a pointer to the class-data space – a pointer to the selector-symbol (always `&s_list`), the number of atoms and a pointer to the list of atoms:

```
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);
```

Adds a method `fn` for an arbitrary message to the class `c`.

The arguments of the anything-method are – apart from a pointer to the class-data space – a pointer to the selector-symbol, the number of atoms and a pointer to the list of atoms:

```
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, ...);
```

Adds a creator-symbol `s`, alternative to the symbolic class name, to the constructor `newmethod`. Thus, objects can be created either by their "real" class name or an alias-name (p.e. an abbreviation, like the internal "float" resp. "f").

The "0"-terminated list of types corresponds to that of `class_new`.

C.2.10 class_sethelpsymbol

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

If a Pd-object is right-clicked, a help-patch for the corresponding object class can be opened. By default this is a patch with the symbolic class name in the directory "*doc/5.reference/*".

The name of the help-patch for the class that is pointed to by `c` is changed to the symbol `s`.

Therefore, several similar classes can share a single help-patch.

Path-information is relative to the default help path *doc/5.reference/*.

C.2.11 pd_new

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

Generates a new instance of the class `cls` and returns a pointer to this instance.

C.3 functions: inlets and outlets

All routines for inlets and outlets need a reference to the object-interna of the class-instance. When instantiating a new object, the necessary data space-variable of the `t_object`-type is initialised. This variable has to be passed as the `owner`-object to the various inlet- and outlet-routines.

C.3.1 inlet_new

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

Generates an additional "active" inlet for the object that is pointed at by `owner`. Generally, `dest` points at "`owner.ob_pd`".

The selector `s1` at the new inlet is substituted by the selector `s2`.

If a message with selector `s1` appears at the new inlet, the class-method for the selector `s2` is called.

This means

- The substituting selector has to be declared by `class_addmethod` in the setup-routine.
- It is possible to simulate a certain right inlet, by sending a message with this inlet's selector to the leftmost inlet.

  Using an empty symbol (`gensym("")`) as selector makes it impossible to address a right inlet via the leftmost one.
- It is not possible to add methods for more than one selector to a right inlet. Particularly it is not possible to add a universal method for arbitrary selectors to a right inlet.

C.3.2 floatinlet_new

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

Generates a new "passive" inlet for the object that is pointed at by `owner`. This inlet enables numerical values to be written directly into the memory `fp`, without calling a dedicated method.

C.3.3 symbolinlet_new

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

Generates a new "passive" inlet for the object that is pointed at by `owner`. This inlet enables symbolic values to be written directly into the memory `*sp`, without calling a dedicated method.

C.3.4 pointerinlet_new

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

Generates a new "passive" inlet for the object that is pointed at by `owner`. This inlet enables pointer to be written directly into the memory `gp`, without calling a dedicated method.

C.3.5 outlet_new

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

Generates a new outlet for the object that is pointed at by `owner`. The Symbol `s` indicates the type of the outlet.

symbol	symbol-address	outlet-type
"bang"	`&s_bang`	message (bang)
"float"	`&s_float`	message (float)
"symbol"	`&s_symbol`	message (symbol)
"pointer"	`&s_gpointer`	message (pointer)
"list"	`&s_list`	message (list)
—	0	message
"signal"	`&s_signal`	signal

There are no real differences between outlets of the various message-types. At any rate, it makes code more easily readable, if the use of outlet is shown at creation-time. For outlets for any messages a null-pointer is used. Signal-outlet must be declared with `&s_signal`.

Variables if the type `t_object` provide pointer to one outlet. Whenever a new outlet is generated, its address is stored in the object variable `(*owner).ob_outlet`.

If more than one message-outlet is needed, the outlet-pointers that are returned by `outlet_new` have to be stored manually in the data space to address the given outlets.

C.3.6 outlet_bang

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

Outputs a "bang"-message at the outlet specified by `x`.

C.3.7 outlet_float

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

Outputs a "float"-message with the numeric value `f` at the outlet specified by `x`.

C.3.8 outlet_symbol

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

Outputs a "symbol"-message with the symbolic value `s` at the outlet specified by `x`.

C.3.9 outlet_pointer

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

Outputs a "pointer"-message with the pointer `gp` at the outlet specified by `x`.

C.3.10 outlet_list

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

Outputs a "list"-message at the outlet specified by `x`. The list contains `argc` atoms. `argv` points to the first element of the atom-list.

Independent of the symbol `s`, the selector "list" will precede the list.

To make the code more readable, `s` should point to the symbol list (either via `gensym("list")` or via `&s_list`)

C.3.11 outlet_anything

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

Outputs a message at the outlet specified by `x`.

The message-selector is specified with `s`. It is followed by `argc` atoms. `argv` points to the first element of the atom-list.

C.4 functions: DSP

If a class should provide methods for digital signal-processing, a method for the selector "dsp" (followed by no atoms) has to be added to this class

Whenever Pd's audio engine is started, all objects that provide a "dsp"-method are identified as instances of signal classes.

DSP-method

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

In the "dsp"-method a class method for signal-processing is added to the DSP-tree by the function `dsp_add`.

Apart from the data space `x` of the object, an array of signals is passed. The signals in the array are arranged in such a way, that they can be read in the graphical representation of the object clockwisely.

In case there are both two in- and out-signals, this means:

pointer	to signal
sp[0]	left in-signal
sp[1]	right in-signal
sp[2]	right out-signal
sp[3]	left out-signal

The signal structure contains apart from other things:

structure-element	description
`s_n`	length of the signal vector
`s_vec`	pointer to the signal vector

The signal vector is an array of samples of type `t_sample`.

perform-routine

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

A pointer `w` to an array (of integer) is passed to the perform-routine that is inserted into the DSP-tree by `class_add`.

In this array the pointers that are passed via `dsp_add` are stored. These pointers have to be casted back to their original type.

The first pointer is stored at `w[1]` !!!

The perform-routine has to return a pointer to integer, that points directly behind the memory, where the object's pointers are stored. This means, that the return-argument equals the routine's argument `w` plus the number of used pointers (as defined in the second argument of `dsp_add`) plus one.

C.4.1 CLASS_MAINSIGNALIN

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

The macro `CLASS_MAINSIGNALIN` declares, that the class will use signal-inlets.

The first macro-argument is a pointer to the signal-class. The second argument is the type of the class-data space. The third argument is a (dummy-)floating point-variable of the data space, that is needed to automatically convert "float"-messages into signals if no signal is present at the signal-inlet.

No "float"-methods can be used for signal-inlets, that are created this way.

C.4.2 dsp_add

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

Adds the perform-routine `f` to the DSP-tree. The perform-routine is called at each DSP-cycle.

The second argumentn defines the number of following pointer-arguments

Which pointers to which data are passes is not limited. Generally, pointers to the data space of the object and to the signal-vectors are reasonable. The length of the signal-vectors should also be passed to manipulate signals effectively.

C.4.3 sys_getsr

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

Returns the sampler ate of the system.

C.5 functions: memory

C.5.1 getbytes

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

Reserves `nbytes` bytes and returns a pointer to the allocated memory.

C.5.2 copybytes

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

Copies `nbytes` bytes from `*src` into a newly allocated memory. The address of this memory is returned.

C.5.3 freebytes

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

Frees `nbytes` bytes at address `*x`.

C.6 functions: output

C.6.1 post

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

Writes a `C`-string to the standard error (shell).

C.6.2 error

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

Writes a `C`-string as an error-message to the standard error (shell).

The object that has output the error-message is marked and can be identified via the Pd-menu *Find->Find last error*.