Sumário

1 Introdução **2**
- 1.1 Escrevendo *externals* . . . 2
- 1.2 Organização do código-fonte e do objeto compilado . . . 3
- 1.3 Compilação . . . 3
  - 1.3.1 Misturando código C e C++ . . . 4
- 1.4 Arquivos de ajuda . . . 4
- 1.5 Utilizando *externals* . . . 4

2 O básico de um *external* **6**
- 2.1 Um *external* simples . . . 6
- 2.2 Uma biblioteca simples . . . 7
- 2.3 Variáveis globais . . . 9

3 Os tipos de dados do PD **11**
- 3.1 Símbolos . . . 11
- 3.2 Mensagens . . . 11
  - 3.2.1 Átomos . . . 11
  - 3.2.2 Seletores . . . 12

4 Construtor e destrutor **14**
- 4.1 Construtor . . . 14
- 4.2 Destrutor . . . 15

5 Inlets e outlets **17**
- 5.1 Inlets passivos . . . 17
- 5.2 Inlets ativos . . . 18
- 5.3 Tratamento de mensagens no primeiro inlet . . . 19
- 5.4 Oulets . . . 20

6 Processamento de Sinais Digitais **23**
- 6.1 Primeiro inlet para DSP . . . 23
- 6.2 Vários inlets DSP . . . 25
- 6.3 Primeiro outlet DSP . . . 26
- 6.4 Inlets e outlets DSP . . . 27
- 6.5 Inlets e outlets DSP criados dinamicamente . . . 27

7 Multithreading **31**
- 7.1 Criando threads . . . 31
- 7.2 Gerenciamento de threads . . . 32
- 7.3 Controle de concorrência . . . 32
- 7.4 Controle via Pure Data . . . 33

8 Externals com GUI - Usando o Tcl/Tk **34**
- 8.1 Iniciando no Tcl/Tk . . . 34
- 8.2 Escrevendo externals com GUI . . . 34
- 8.3 Adicionando componentes gráficos . . . 39
- 8.4 Adicionando comandos . . . 42

9 Orientação a Objetos **44**

10 Miscelâneas **45**
10.1 Gerenciamento de memória 45
10.2 Atoms 45
10.3 Binbufs 46
10.4 Clocks 47
10.5 Pure data 47
10.6 Pointers 48
10.7 Inlets and outlets 48
10.8 Canvases 49
10.9 Classes 50
10.10Printing 51
10.11System interface routines 52
10.12Threading 53
10.13Signals 53
10.14Utility functions for signals 55
10.15Data 55
10.16GUI interface - functions to send strings to TK 56

Capítulo 1

Introdução

Pure Data, ou simplesmente Pd, é um ambiente visual de programação musical que permite a criação de aplicações musicais complexas a partir da combinação de componentes visuais mais simples chamados **objetos**. As distribuições oficiais do Pure Data contêm diversos objetos prontos para o uso, mas também permitem a extensão de suas funcionalidades através da criação de novos objetos utilizando C/C++. Desta forma, novas linhas de código escritas pelo usuário são compilados como bibliotecas dinâmicas e podem ser carregadas pelo programa em tempo de execução. Objetos desta forma levam o nome de ***externals***.

Este é um tutorial prático para o desenvolvimento de *externals* em C para o Pure Data. A iniciativa de escrever este documento surgiu no primeiro semestre de 2011, durante a disciplina de Computação Musical ministrada pelo professor Marcelo Gomes de Queiroz no Instituto de Matemática e Estatística da Universidade de São Paulo. A intenção deste tutorial é auxiliar programadores a desenvolver *externals* de maneira bastante simples através de exemplos práticos.

Mais do que ampliar a gama de objetos do Pure Data e criar novos objetos, o objetivo deste trabalho é também fornecer ao pesquisador de computação musical uma ferramenta para implementar e testar algoritmos de processamento de áudio para caráter de estudo. Isto significa que podemos reimplementar várias coisas que já existem no Pure Data simplesmente porque é didático programar e colocar algoritmos para funcionar.

Não é objetivo deste tutorial ensinar processamento de som, ensinar algoritmos, ensinar programar em C/C++ ou ensinar a utilizar o Pure Data. Também não é objetivo questionar o modo como o Pd e seus externals foram feitos.

Seria indevido não dizer que nada no mundo se aprende sozinho. Foi graças aos vários *externals* escritos para o Pd, com seu código aberto e documentado que conseguimos reunir o conhecimento que aqui presente. Seria impossível citar todos os autores de *externals* que nos ajudaram sem saber. No entanto, não deixamos de agradecer ao que chamamos de comunidade de software livre, ao autor do Pd (seria Public Domain?), Miller Puckette e ao autor do outro tutorial, IOHannes Zmölnig. Muito obrigado.

Este tutorial está acompanhado de vários exemplos cujos códigos ilustram os nossos passos. A estes tutoriais foi adicionado exemplos de Tk e também um makefile que permite compilá-los em vários sistemas operacionais.

Este tutorial ainda não está pronto e por isto você encontrará caixinhas como esta com notas do que mais temos de fazer.

1.1 Escrevendo *externals*

O código fonte do Pure Data é organizado de acordo com convenções de programação orientada a objetos. Para o desenvolvimento de *externals*, é necessário seguir estas convenções e fornecer ao ambiente uma nova classe com alguns métodos específicos, como veremos mais adiante. Para desenvolver para o Pure Data, é necessário importar o arquivo de cabeçalho `m_pd.h`[1], que contém definições de constantes, tipos e funções.

Uma boa fonte de informação é o tutorial de *externals*[2] escrito pelo IOHannes[3], um dos programadores do Pure Data. Apesar de ter utilizado este documento como ponto de partida, boa parte do que está

[1] http://pure-data.git.sourceforge.net/git/gitweb.cgi?p=pure-data/pure-data;a=blob_plain;f=src/m_pd.h;hb=HEAD
[2] http://iem.at/pd/externals-HOWTO/pd-externals-HOWTO.pdf
[3] http://puredata.info/author/zmoelnig

incluso no presente tutorial foi aprendido a partir da leitura do código-fonte de *externals* contidos no repositório oficial do Pure Data[4].

Navegando pelos códigos-fonte deste repositório você poderá notar que os programadores que escreveram os externals que hoje estão disponível para o Pd seguiram estas convenções e por isto a leitura destes códigos-fonte pode ser didática e simples.

Por esta razão, o primeiro conselho que damos para quem irá escrever *externals* é seguir estas convenções, mesmo que as mesmas não sejam a maneira como você está acostumado a programar deste jeito pois assim seu código também será didático e simples de entender.

Este tutorial não pretende cobrir os algoritmos de processamento de sinais mas explicar como implementar estes algoritmos como objetos do Pd. Para processamento de sinais há uma vasta bibliografia disponível que possui os algoritmos e o ferramental matemático necessário para sua implementação.

Será que podemos citar aqui algum livro ou material para DSP?

1.2 Organização do código-fonte e do objeto compilado

Um novo *external* corresponde a uma nova classe na arquitetura orientada a objetos do Pure Data. Para que o carregamento da biblioteca dinâmica em tempo de execução funcione corretamente, é necessário que o arquivo binário produzido possua o mesmo nome que a classe correspondente ao *external*.

Para criar, por exemplo, um *external* chamado "passa-baixas", podemos escrever seu código-fonte em um arquivo chamado `passa-baixas.c`, e em seguida compilar um objeto de biblioteca compartilhada chamado `passa-baixas.pd_linux`, no caso do sistema GNU/Linux. Outras arquiteturas de sistema utilizam outras extensões para o nome do objeto com a biblioteca compartilhada do *external*, como por exemplo `.dll` (M$ Windows), `.pd_irix5` (SGI Irix) ou `.pd_darwin` (Mac OS X).

Importante: O nome do arquivo com o código-fonte não possui formato obrigatório, mas o nome do objeto compilado com a biblioteca dinâmica deve sempre corresponder ao nome da classe, assim como sua extensão deve sempre corresponder à arquitetura do sistema utilizado.

O mesmo cuidado é recomendado para os métodos que serão definidos internamente no objeto. Os nomes de métodos que serão apresentados neste material seguem o padrão encontrado no repositório do Pd. É fortemente recomendado que o mesmo padrão seja utilizado em seu texto.

1.3 Compilação

Para criar um objeto binário que pode ser carregado no Pure Data em tempo de execução, primeiro compilamos o código fonte, criando assim um ou mais objetos intermediários, e em seguida utilizamos um ligador (*linker*) para criar um objeto de biblioteca compartilhada.

No GNU/Linux, uma forma de realizar o processo `example01.c` → `example01.o` → `example01.pd_linux` é a seguinte:

```
1 EXTNAME=example01
2 cc -DPD -fPIC -Wall -o ${EXTNAME}.o -c ${EXTNAME}.c
3 ld -shared -lc -lm -o ${EXTNAME}.pd_linux ${EXTNAME}.o
4 rm ${EXTNAME}.o
```

A opção de compilação `-fPIC` resulta na criação de código binário que roda independente de sua posição na memória, adequado para geração de bibliotecas compartilhadas. A opção `-shared` passada para o ligador determina a criação de uma biblioteca compartilhada.

Para facilitar a compilação, é interessante utilizar um `makefile`. Os exemplos deste tutorial estão acompanhadas de um `makefile` produzido pelo professor Marcelo Queiroz e adaptado para este tutorial.

Tem uma dezena de jeito de compilar pro Windows, usando o Mingw ou o C++ Builder. Aqui[5] temos exemplos e muitas discussões de como compilar externals no Windows.

Será que compensa testarmos isto tudo?

[4] http://pure-data.svn.sourceforge.net/viewvc/pure-data/trunk/externals/

[5] http://puredata.hurleur.com/sujet-1029-problem-compiling-external-windows

1.3.1 Misturando código C e C++

Existem algumas diferenças entre compiladores C e C++ que tornam a sintaxe das linguagens incompatíveis, gerando resultados diferentes para um mesmo trecho de código. Um exemplo disso que influencia o funcionamento de *externals* no Pure Data é a geração da tabela de símbolos dos objetos binários.

Compiladores C++ realizam um processo chamado *name mangling* (ou "dilaceramento de nomes"), que consiste em alterar o nome de funções, estruturas, classes, etc, incluindo informações sobre o espaço de nomes do objeto em questão. Isto resulta em nome diferentes gravados nas tabelas de símbolos dos objetos binários, o que pode confundir o Pure Data no momento do carregamento de um *external*.

Para garantir que um compilador C++ gere nomes compatíveis com objetos binários C, utilize a expressão `extern "C"` na frente dos nomes das funções que serão chamadas pelo Pure Data:

```
1 extern "C" example01_setup(void);
2 extern "C" example01_new(void);
```

1.4 Arquivos de ajuda

É importante distribuir, junto com novos *externals*, um arquivo de ajuda do Pure Data com instruções e exemplos de utilização. Como convenção, o arquivo de ajuda deve ter o mesmo nome que o *external*, acrescido do sufixo `-help.pd`. Por exemplo, para o código fonte `example01.c`, que gera o objeto `example01.pd_linux`, escrevemos também o arquivo `example01-help.pd`.

No próximo capítulo veremos uma forma de associar o arquivo de ajuda com a opção de ajuda que aparece no menu contextual com um clique do botão direito no objeto do external dentro do Pure Data.

1.5 Utilizando *externals*

Para carregar um *external* em um `patch` do Pure Data em tempo de execução, basta criar um objeto (com `CTRL+1` ou acessando o menu `Put` → `Object`) com o caminho (relativo ou absoluto) para o objeto compilado com a biblioteca compartilhada, omitindo a extensão.

É possível adicionar o diretório que contém o arquivo binário do *external* ao caminho de busca do Pure Data, de forma que para acessá-lo de dentro de um *patch* não seja necessário digitar o caminho inteiro até o objeto. Isto pode ser feito através da passagem de um parâmetro na linha de comando do Pure Data com a opção `-path <caminho>`, ou de forma gráfica acessando a opção `File` → `Path...` no menu do Pure Data, como pode ser visto na figura 1.1.

Para carregar uma biblioteca de *externals* (mais de um *external* no mesmo arquivo-fonte), é possível indicar o nome da bibliotecai na linha de comando do Pure Data utilizando a opção `-lib <biblioteca>`, ou também graficamente através do menu `File` → `Startup...`, como pode ser visto na figura 1.2.

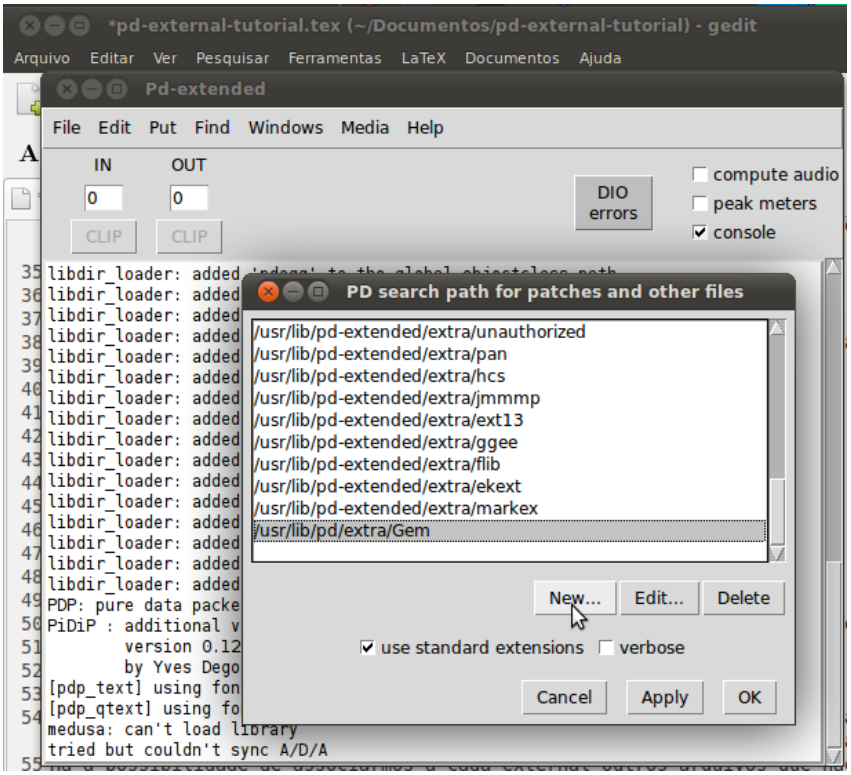

Figura 1.1: Adicionando o diretório de um *external* ao caminho de busca do Pure Data.

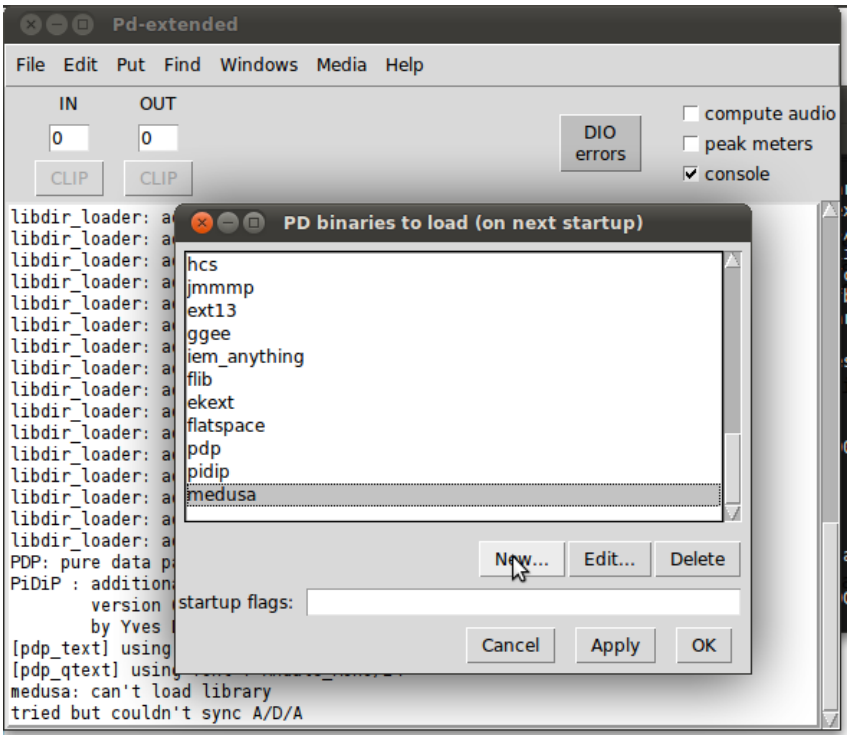

Figura 1.2: Adicionando uma biblioteca ao Pure Data.

Capítulo 2

O básico de um *external*

Escrever um *external* significa seguir as recomendações da API. Peço ao leitor bastante paciência pois este tutorial pretende andar um pouco devagar para mostrar os passos da escrita de um *external*.

2.1 Um *external* simples

Como dissemos anteriormente, a arquitetura do Pure Data é organizada de acordo com o paradigma de orientação a objetos: cada objeto gráfico do Pure Data corresponde a uma instância de uma classe. Neste sentido, um *external* está associado a um conjunto de estruturas de dados que representam classes em C. Para cada classe é necessário haver métodos de instanciação, destruição, processamento de sinais, tratamento de mensagens, etc.

A infraestrutura mínima para o funcionamento de um external (de nome, digamos, `<external>`) consiste em uma estrutura de dados para a representação de uma classe, que deve ter nome `t_<external>`, e dois métodos obrigatórios, chamados `<external>_setup()` e `<external>_new()`. Note que a convenção de nomes utilizada no Pure Data é de que toda função deve ser nomeada da forma `<contexto>_<funcao>()`.

A estutura de dados que representa uma classe do Pure Data deve obrigatoriamente possuir o primeiro atributo do tipo `t_object`, no qual é armazenado o objeto criado no momento da instanciação. Outros atributos podem ser adicionados a esta estrutura de maneira que cada instância da mesma classe possua os atributos necessários para seu funcionamento. Uma classe que acessa um arquivo, por exemplo, pode possuir como atributos uma string para guardar o caminho e um inteiro para guardar o descritor do arquivo.

Um exemplo de estrutura de dados para representação de uma classe chamada `example1` consiste no seguinte:

```
1 static t_class *example1_class;
2
3 typedef struct _example1 {
4   t_object x_obj;
5 } t_example1;
```

Sempre que um *external* é carregado pelo Pure Data, o método de nome `<external>_setup()` é executado. No exemplo dado acima, o Pure Data irá procurar, no arquivo binário `example1.pd_linux` que contém a biblioteca compartilhada, o método de nome `example1_setup(void)`. Este método é utilizado para realizar a inicialização da classe, informando ao Pure Data da existência de uma nova classe no sistema e associando a ela os métodos de instanciação e destruição, além de outras informações:

```
1 void example1_setup(void) {
2   example1_class = class_new(
3     gensym("example1"),          // Nome simbolico
4     (t_newmethod) example1_new, // Construtor
5     0,                           // Destrutor
6     sizeof (t_example1),         // Tamanho dos atributos
```

```
7       CLASS_NOINLET,                  // Flags da classe
8       0                               // Tipos dos argumentos
9     );
10  }
```

Dentro do método `<external>_setup()` não há limite para o número de classes a definir, de forma que é possível definir apenas uma classe (como no exemplo 1) ou uma biblioteca inteira com várias classes (como no exemplo 3). A introdução de uma nova classe no sistema é realizada pela função `class_new()`. São parâmetros da função `class_new()`:

- Nome simbólico da classe.
- Método construtor de um objeto.
- Método destrutor de um objeto.
- Tamanho do espaço de dados dos atributos de um objeto.
- Flags que definem a representação gráfica do objeto.
- Tipos dos parâmetros a serem passados para o construtor quando da instanciação de um objeto (veja o próximo capítulo).

É necessário terminar a lista de tipos de parâmetros com um número inteiro 0, para indicar ao Pure Data que a lista de tipos terminou. Consulte a documentação da função `class_new()` para mais detalhes[1].

O método `<external>_new()`, que foi associado como método de instanciação de objetos na chamada de `class_new()`, realiza a instanciação de objetos propriamente dita. Neste método, além da instanciação de um novo objeto através da função `pd_new()`, é possível definir os valores dos atributos da estrutura de dados da classe e também inicializar quaisquer outros contextos que sejam necessários, como por exemplo abrir arquivos, preencher vetores, alocar memória, etc.

```
1  // Construtor da classe
2  void * example1_new(void) {
3      t_example1 *x = (t_example1 *) pd_new(example1_class);
4      return (void *) x;
5  }
```

Após a criação da estrutura de dados dos métodos da forma mencionada acima, a compilação realizada da forma descrita na seção 1.3, e a criação do objeto no Pure Data como descrito na seção 1.5, o resultado pode ser visto na figura 2.1.

2.2 Uma biblioteca simples

Um mesmo método `<external>_setup()` pode definir várias classes diferentes. A isto damos o nome de biblioteca. Neste cenário, o método `<external>_setup()` possui o mesmo nome do arquivo com a biblioteca, mas cada classe podem ter um nome diferente (veja o exemplo 3).

```
1  void example3_setup(void) {
2    post("Initializing my library");
3
4    myobj1_class = class_new(
5      gensym("myobj1"),
6      (t_newmethod) myobj1_new, // Constructor
7      0,
```

[1] http://pdstatic.iem.at/externals-HOWTO/node9.html#SECTION00092100000000000000

Figura 2.1: Nosso primeiro *external* do PD. Ainda inútil. :-)

```
8       sizeof (t_myobj1),
9       CLASS_NOINLET,
10      0);
11    class_sethelpsymbol(myobj1_class, gensym("myobj1-help"));
12
13    myobj2_class = class_new(
14      gensym("myobj2"),
15      (t_newmethod) myobj2_new, // Constructor
16      0,
17      sizeof (t_myobj2),
18      CLASS_NOINLET,
19      0);
20    class_sethelpsymbol(myobj2_class, gensym("myobj2-help"));
21  }
```

Se o arquivo foi preenchido corretamente, compilado corretamente e adicionado ao caminho do PureData, teremos o resultado visto na figura 2.2.

Dentro do Pure Data, um clique com o botão direito em um objeto abre um menu no qual uma das opções é `Ajuda`. Quando esta opção é selecionada, o Pure Data abre um patch associado ao objeto, que deve conter instruções e exemplos de uso. Por padrão, o Pure Data procura um arquivo com o mesmo nome que o external (acrescido da extensão `-help.pd`) no diretório padrão de documentação (`doc/5.reference`). Para associar um arquivo diferente do padrão, basta utilizar a função `class_sethelpsymbol`:

```
1  class_sethelpsymbol(myclass_class, gensym("my_class-help"));
```

Um objeto pode ainda ter outros nomes (*aliases*). Para definir isto podemos utilizar a função `class_addcreator()`. Veja o exemplo:

```
1  class_addcreator((t_newmethod)medusa_new, gensym("med"), 0);
```

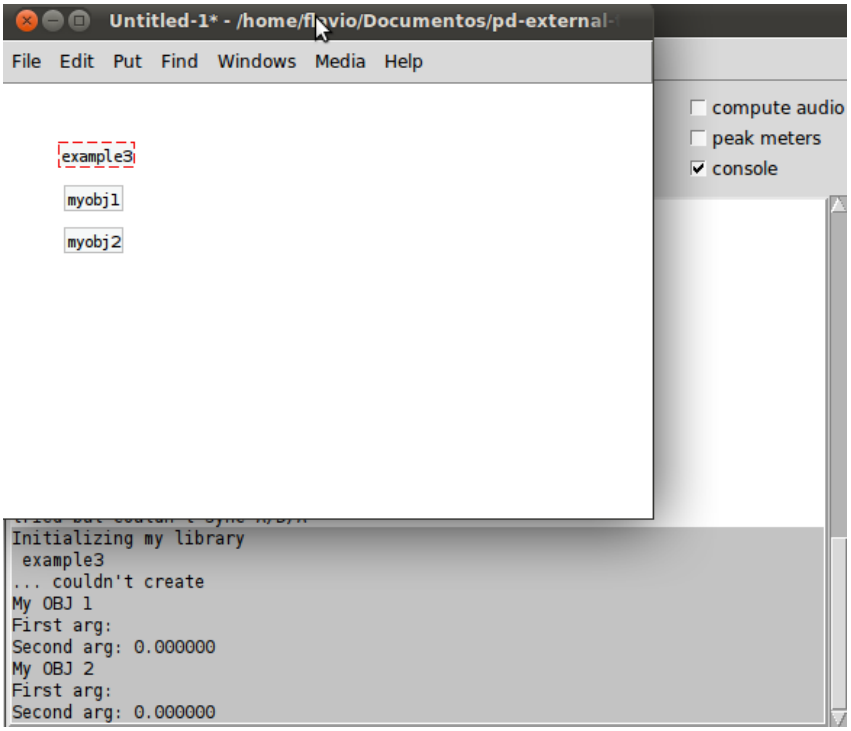

Figura 2.2: Nosso segundo *external* do PD. Ainda inútil. :-)

2.3 Variáveis globais

É possível utilizar variáveis globais para armazenar dados de um *external*. Estas variáveis são visíveis para todas as intâncias de objetos do *external* e todas podem alterar seus valores. Isto pode ser útil ou um desastre (veja o exemplo16). Por exemplo, cada instância do *external* `example16` definido a partir do código a seguir incrementa em uma unidade o valor do contador, como pode ser visto na figura 2.3:

```
1 int count = 0;
2
3 void * example16_new(void) {
4     t_example16 *x = (t_example16 *) pd_new(example16_class);
5     post("Counter value: %d",count);
6     count++;
7     return (void *) x;
8 }
```

Caso isto não seja desejável, o ideal é incluir as variáveis dentro da estrutura do objeto. Assim, neste exemplo cada instância terá seu próprio contador:

```
1 static t_class *example_class;
2
3 typedef struct _example {
4     t_object x_obj;
5     t_int counter;
6 } t_example;
```

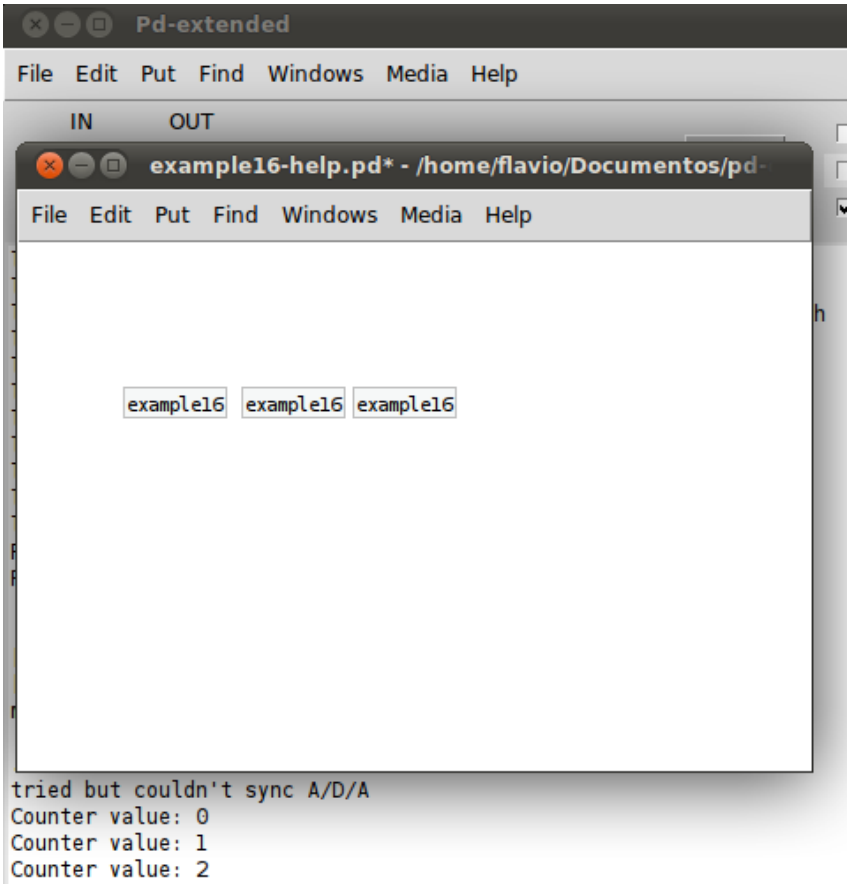

Figura 2.3: Repare na saída da janela principal.

```
7
8  void * example_new(void) {
9      t_example *x = (t_example *) pd_new(example_class);
10     post("Counter value: %d",x->counter);
11     x->counter++;
12     return (void *) x;
13 }
```

Capítulo 3

Os tipos de dados do PD

Uma vez que o Pure Data é utilizado em diversas plataformas, muitos tipos comuns de variáveis, como `int`, são redefinidos. Para escrever um *external* que seja portável para qualquer plataforma, é razoável que você utilize os tipos providos pelo Pure Data. Como dissemos na seção 1.2, para escrever um *external*, é necessário incluir o arquivo `m_pd.h` que possui definições de constantes (versão do Pure Data, sistema operacional, compilador, etc), estruturas, assinaturas de funções e tipos de dados.

Existem muitos tipos predefinidos que devem fazer a vida do programador mais simples. Em geral, os tipos do pd têm nome iniciado por `t_`.

tipo do pd	descrição
`t_atom`	átomo
`t_float`	valor de ponto flutuante
`t_symbol`	símbolo
`t_gpointer`	ponteiro (para objetos gráficos)
`t_int`	valor inteiro
`t_signal`	estrutura de um sinal
`t_sample`	valor de um sinal de áudio (ponto flutuante)
`t_outlet`	*outlet* de um objeto
`t_inlet`	*inlet* de um objeto
`t_object`	objeto gráfico
`t_class`	uma classe do pd
`t_method`	um método de uma classe
`t_newmethod`	ponteiro para um construtor (uma função `_new`)

3.1 Símbolos

Um símbolo corresponde a um valor constante de uma *string*, ou seja, uma sequência de letras que formam uma palavra única.

Cada símbolo é armazenado em uma tabela de busca por razões de performance. A função `gensym(char *)` procura por uma string em uma tabela de busca e retorna o endereço daquele símbolo. Se a string não foi encontrada na tabela, um novo símbolo é adicionado.

Estes símbolos serão usados para várias coisas como para criar e associar mensagens entre objetos, definir ações esperadas para mensagens recebidas por inlets, criar comunicação entre a GUI e o Pd, entre outras.

3.2 Mensagens

Dados que não correspondem a áudio são distribuídos via um sistema de mensagens. Cada mensagem é composta de um "seletor"e uma lista de átomos.

3.2.1 Átomos

Os tipos de átomo mais utilizados são:

- `A_FLOAT`: um valor numérico (de ponto flutuante).
- `A_SYMBOL`: um valor simbólico (string).
- `A_POINTER`: um ponteiro.

Valores numéricos são sempre considerados valores de ponto flutuante (`t_float`), mesmo que possam ser exibidos como valores inteiros.

Átomos do tipo `A_POINTER` não são muito importantes (para *externals* simples).

O tipo de um átomo `a` é armazenado no elemento da estrutura `a.a_type`.

Outros tipos de átomo definidos no arquivo `m_pd.h` são:

- A_NULL,
- A_FLOAT,
- A_SYMBOL,
- A_POINTER,
- A_SEMI,
- A_COMMA,
- A_DEFFLOAT,
- A_DEFSYM,
- A_DOLLAR,
- A_DOLLSYM,
- A_GIMME,
- A_CANT

Verificar para que estes átomos servem e montar exemplos com eles.

3.2.2 Seletores

Um seletor é um símbolo que define o tipo de uma mensagem. Existe cinco seletores pré-definidos:

- `bang`: rotula um gatilho de evento. Uma mensagem de `bang` consiste somente do seletor e não contém uma lista de átomos.
- `float` rotula um valor numérico. A lista de uma mensagem `float` contém um único átomo de tipo `A_FLOAT`.
- `symbol` rotula um valor simbólico. A lista de uma mensagem `symbol` consiste em um único átimo do tipo `A_SYMBOL`.
- `pointer` rotula um valor de ponteiro. A lista de uma mensagem do tipo `pointer` contém um único átimo do tipo `A_POINTER`.
- `list` rotula uma lista de um ou mais átomos de tipos arbitrários.

Uma vez que os símbolos para estes seletores são utilizados com frequência, seu endereço na tabela de símbolos pode ser utilizado diretamente, sem a necessidade da utilização de `gensym`:

seletor	rotina de busca	endereço de busca
`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_signal`

Outros seletores também podem ser utilizados. A classe receptora tem que prover um médodo para um seletor específico ou para `anything`, que corresponde a qualquer seletor arbitrário.

Mensagens que não possuem seletor explícito e começam com um valor numérico são reconhecidas automaticamente como mensagens `float` (se consistirem de apenas um átomo) ou como mensagens `list` (se forem compostas de diversos átomos).

Por exemplo, as mensagens `12.429` e `float 12.429` são idênticas. Da mesma forma, as mensagens `list 1 para voce` é idêntica a `1 para voce`.

Capítulo 4

Construtor e destrutor

O Construtor de um objeto pode receber parâmetros. Estes parâmetros são ilustrados abaixo.

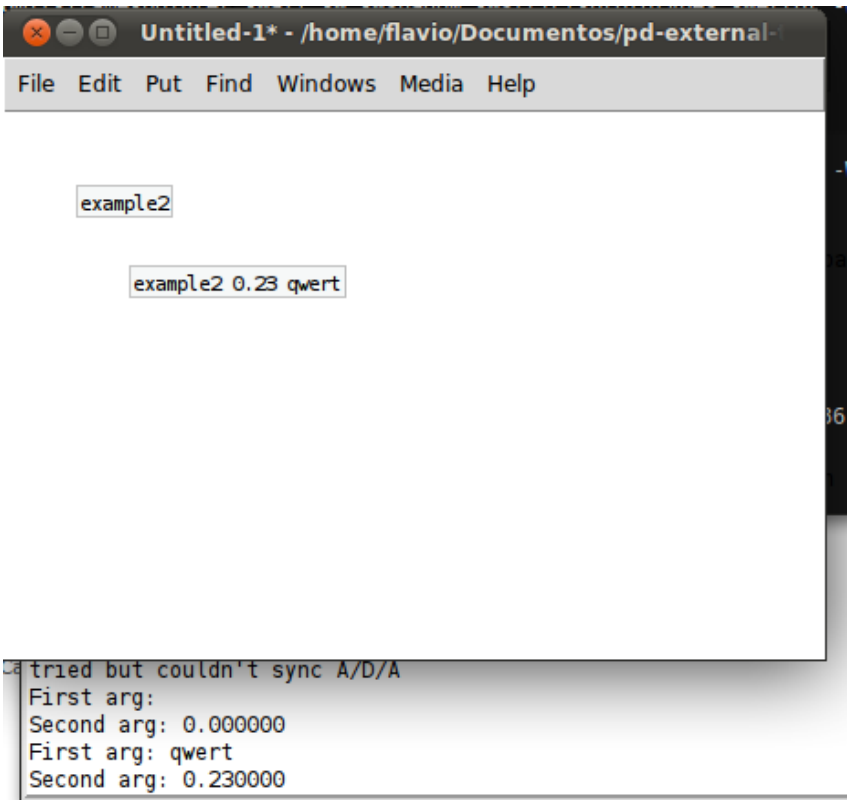

Figura 4.1: External recebendo parâmetros. Note a tela de saída no fundo da imagem.

4.1 Construtor

Parâmetros de inicialização no construtor podem permitir que inicializemos o external com determinados valores. Isto é feito definindo os parâmetros no métodos class_new() quanto na definição da função construtora. (Veja o exemplo02).

```
1 // Constructos of the class
2 void * example2_new(t_symbol * arg1, t_floatarg arg2) {
3     t_example2 *x = (t_example2 *) pd_new(example2_class);
4     post("First arg: %s", arg1->s_name);
5     post("Second arg: %f", arg2);
6     return (void *) x;
7 }
```

```
8
9  void example2_setup(void) {
10     example2_class = class_new(gensym("example2"),
11             (t_newmethod) example2_new, // Constructor
12             0,
13             sizeof (t_example2),
14       CLASS_NOINLET,
15             A_DEFFLOAT, // First Constructor parameter
16             A_DEFSYMBOL, // Second Constructor parameter
17             0);
18 }
```

Notem que os parâmetros são definidos com um tipo e são recebidos com outro. Como explicado na seção 3.2, todos os dados que não correspondem a sinais de áudio são transmitidos como mensagens, compostas de átomos. Para ver os tipos de átomo que podem ser utilizados na passagem de parâmetros, veja a seção 3.2.1.

Estranho que no método class_new, o float foi definido primeiro.

Para aceitar qualquer tipo de átomo na passagem de um parâmetro específico, utilize o tipo de átomo `A_GIMME` (veja o exemplo09).

```
1  // Constructor of the class
2  void * example9_new(t_symbol *s, int argc, t_atom * argv) {
3     t_example9 *x = (t_example9 *) pd_new(example9_class);
4     post("%d parameters received",argc);
5     return (void *) x;
6  }
7
8  void example9_setup(void) {
9     example9_class = class_new(gensym("example9"),
10      (t_newmethod) example9_new, // Constructor
11      (t_method) example9_destroy, // Destructor
12      sizeof (t_example9),
13      CLASS_NOINLET,
14      A_GIMME, // Allows various parameters
15      0); // LAST argument is ALWAYS zero
16 }
```

Quando utilizamos o tipo de átomo `A_GIMME` o método construtor funciona como uma função `main()` em C: ela recebe os parâmetros `argc`, que indica o número de átomos na lista, e `*argv`, que aponta para a lista de átomos de fato. Veja o exemplo na figura 4.2.

Note que o Pure Data não obriga que o usuário passe parâmetros para o objeto. É como se todo construtor, independentemente de como ele está definido, aceitasse sua instanciação vazia. Cabe ao programador verificar se os parâmetros recebidos são em quantidade, tipo e valor esperado e, caso não seja, abortar a construção do objeto e não retornar sua instância.

4.2 Destrutor

O destrutor de uma classe permite liberar alguma memória eventualmente alocada pelo construtor ou por outras funções do *external* (veja o exemplo 07).

```
1  // Destroy the object
2  void example9_destroy(t_example9 *x) {
3    post("You say good bye and I say hello");
4  }
5
6  void example9_setup(void) {
```

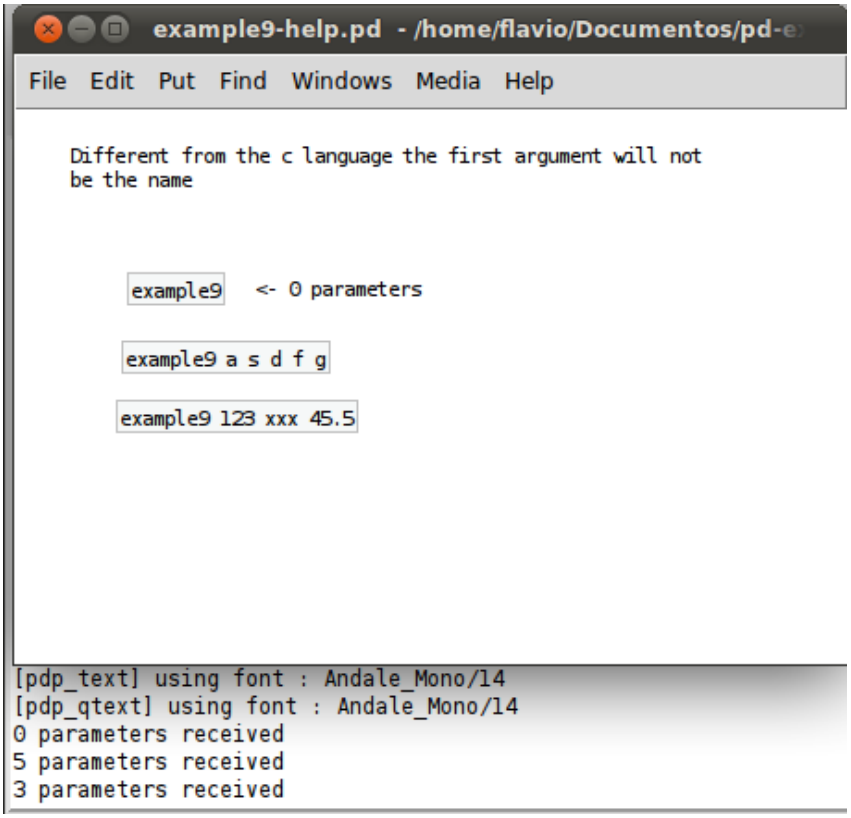

Figura 4.2: Diferente da linguagem C, o primeiro parâmetro não é o nome do external.

```
7     example9_class = class_new(gensym("example9"),
8       (t_newmethod) example9_new, // Constructor
9       (t_method) example9_destroy, // Destructor
10      sizeof (t_example9),
11      CLASS_NOINLET,
12      A_GIMME, // Allows various parameters
13      0); // LAST argument is ALWAYS zero
14  }
```

A liberação da memória pode ser feita utilizando a função `freebytes()` definida na API do Pure Data.

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

Capítulo 5

Inlets e outlets

Os objetos que criamos até agora são inúteis. Não servem para nada, pois não se comunicam com outros objetos nem modificam sinais de áudio. Para dar utilidade a um *external*, é necessário que ele comunique com outros objetos do Pure Data. Isto é feito por meio de *inlets* e *outlets*, portas de entrada e saída (respectivamente) de sinais de áudio e/ou mensagens.

Neste capítulo vamos tratar exclusivamente de inlets e outlets de mensagens. Entre os inlets de mensagem há os tipos passivos e ativos. Inlets **passivos** são inlets cujo valor recebido é associada diretamente a um atributo do objeto. São chamados de passivos pois a alteração do seu valor não resulta na chamada de um método e a atribuição do valor recebido ao atributo do objeto é feita automaticamente. Inlets **ativos**, por outro lado, são associados a funções e permitem a execução de uma função arbitrária quando um valor é recebido no inlet.

5.1 Inlets passivos

Abaixo vemos um exemplo de objeto com um inlet passivo (veja a figura 5.1):

```
1  static t_class *example4_class;
2
3  typedef struct _example4 {
4    t_object x_obj;
5    t_float my_float;
6  } t_example4;
7
8  // Constructor of the class
9  void * example4_new(t_symbol * arg1, t_floatarg arg2) {
10   t_example4 *x = (t_example4 *) pd_new(example4_class);
11   post("First arg: %s", arg1->s_name);
12   post("Second arg: %f", arg2);
13   floatinlet_new(&x->x_obj, &x->my_float);
14   return (void *) x;
15 }
```

Neste exemplo, o atributo my_float do objeto é associado a um inlet do tipo float. Isto significa que, caso tenhamos uma mensagem float ligada a este objeto, o valor desta mensagem ficará armazenada no atributo my_float.

Um inlet passivo é associado a um tipo do Pure Data, e requer que o atributo associado seja do mesmo tipo do valor recebido através do inlet. Para cada tipo do Pure Data, utiliza-se uma função diferente para criar inlets que recebam aquele tipo (veja o exemplo 04).

As funções para criar os inlets passivos dos tipos mais comuns são:

- `floatinlet_new(t_object *owner, t_float *fp)`
- `symbolinlet_new(t_object *owner, t_symbol **sp)`

- `pointerinlet_new(t_object *owner, t_gpointer *gp)`

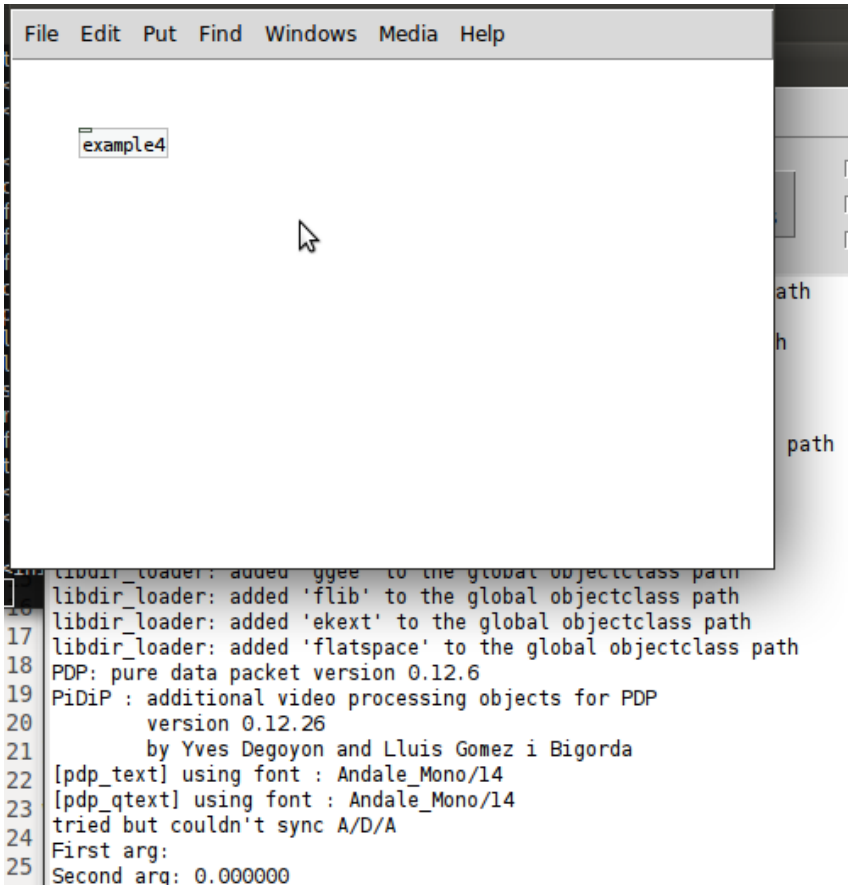

Figura 5.1: Inlets passivos

5.2 Inlets ativos

Um inlet ativo associa ao inlet uma função. Assim como o inlet passivo, a criação de um inlet ativo define o tipo do átomo que o inlet receberá (veja o exemplo 05 e o resultado na figura 5.2).

```
1  // all inlet-methods receive the object as their first argument.
2  void example5_bang(t_example5 *x) {
3    post("BANGED!");
4    post("My_float value: %f",x->my_float);
5  }
6
7  void example5_anything(t_example5 *x, t_symbol *s, int argc,
       t_atom *argv){
8    post("ANYTHING!");
9  }
10
11 void example5_setup(void) {
12   example5_class = class_new(gensym("example5"),
13     (t_newmethod) example5_new, // Constructor
14     0,
15     sizeof (t_example5),
16     CLASS_DEFAULT,
17     0); // LAST argument is ALWAYS zero
18   class_addbang(example5_class, example5_bang);
```

```
19    class_addanything(example5_class, example5_anything);
20 }
```

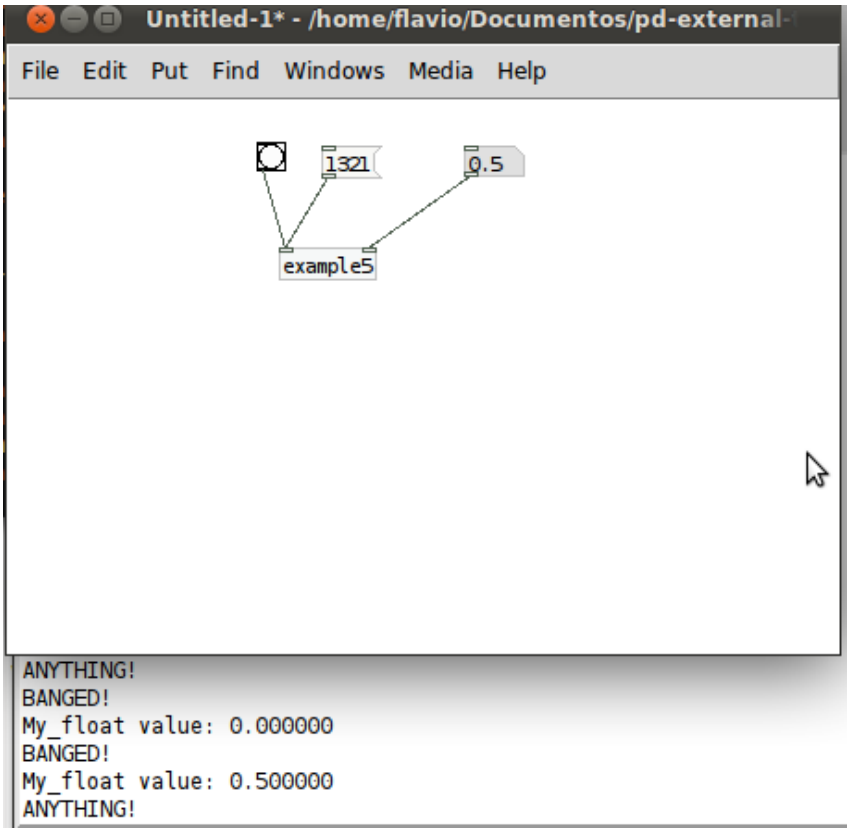

Figura 5.2: Inlets ativos.

Da mesma forma que ocorre com inlets passivos, cada inlet ativo é associado a um tipo de átomo, de forma que sua definição também deve ser feita através de funções específicas. Abaixo, a tabela com os métodos que criam inlets ativos, e assinaturas possíveis para funções associadas a cada tipo:

método do pd para criar inlet ativo	assinatura para o método associado ao inlet
`class_addbang(t_class *c, t_method fn);`	`my_b(t_myt *x);`
`class_addfloat(t_class *c, t_method fn);`	`my_f(t_myt *x, t_floatarg f);`
`class_addsymbol(t_class *c, t_method fn);`	`my_s(t_myt *x,t_symbol *s);`
`class_addpointer(t_class *c, t_method fn);`	`my_p(t_myt *x, t_gpointer *pt);`
`class_addlist(t_class *c, t_method fn);`	`my_l(t_myt *x, t_symbol *s, int argc, t_atom *argv);`
`class_addanything(t_class *c, t_method fn);`	`my_a(t_mydata *x, t_symbol *s, int argc, t_atom *argv);`

5.3 Tratamento de mensagens no primeiro inlet

É possível associar átomos a métodos e executar métodos diferentes dependendo do conteúdo das mensagens recebidas pelo primeiro inlet. Isto é feito através da função `add_method()` (veja o exemplo 08).

```
1  // Constructor of the class
2  void * example8_new(void) {
3    t_example8 *x = (t_example8 *) pd_new(example8_class);
4    // create an inlet that calls the method "example8_alfa"
5    inlet_new(&x->x_obj, &x->x_obj.ob_pd, gensym("float"), gensym("
       alfa"));
6    return (void *) x;
```

```
7  }
8
9  void example8_start(t_example8 *x){
10   post("START / BANG");
11 }
12
13 void example8_open(t_example8 *x, t_symbol *s){
14   post("open %s",s->s_name);
15 }
16
17
18 void example8_alfa(t_example8 *x, t_floatarg f){
19   post("ALFA VALUE %f",f);
20 }
21
22 void example8_setup(void) {
23   example8_class = class_new(gensym("example8"),
24     (t_newmethod) example8_new, // Constructor
25     (t_method) example8_destroy, // Destructor
26     sizeof (t_example8),
27     CLASS_DEFAULT,
28     0); // LAST argument is ALWAYS zero
29   // All these messages will be received by the first left inlet
30   class_addmethod(example8_class, (t_method) example8_start,
31     gensym("start"), 0); // two messages, the same function
32   class_addmethod(example8_class, (t_method) example8_start,
33     gensym("bang"),  0); // may be "start" or "bang" messages
34   class_addmethod(example8_class, (t_method) example8_open,
35     gensym("open"),  A_DEFSYMBOL,0);
36   // These messages will be associated with inlet 2
37   class_addmethod(example8_class, (t_method) example8_alfa,
38     gensym("alfa"), A_DEFFLOAT,0);
39 }
```

Desta forma não precisamos tratar a mensagem que o inlet recebe mas definí-las de antemão e criar funções que mapeiem a mensagem recebida. Veja a figura 5.3.

O método `inlet_new()` pode ser utilizado para criar inlets genéricos que terão com métodos associados através de símbolos de mensagens:

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

5.4 Oulets

Depois de termos tratado as formas de entrada de dados através de inlets do Pure Data, chegou a hora de falarmos das saídas. A saída de dados dos objetos do Pure Data é feita por meio de outlets (veja o exemplo 06).

```
1  typedef struct _example6 {
2    t_object x_obj;
3    t_outlet *my_outlet; // Defines an outlet
4  } t_example6;
5
6  // The BANG method, first inlet
7  void example6_bang(t_example6 *x) {
```

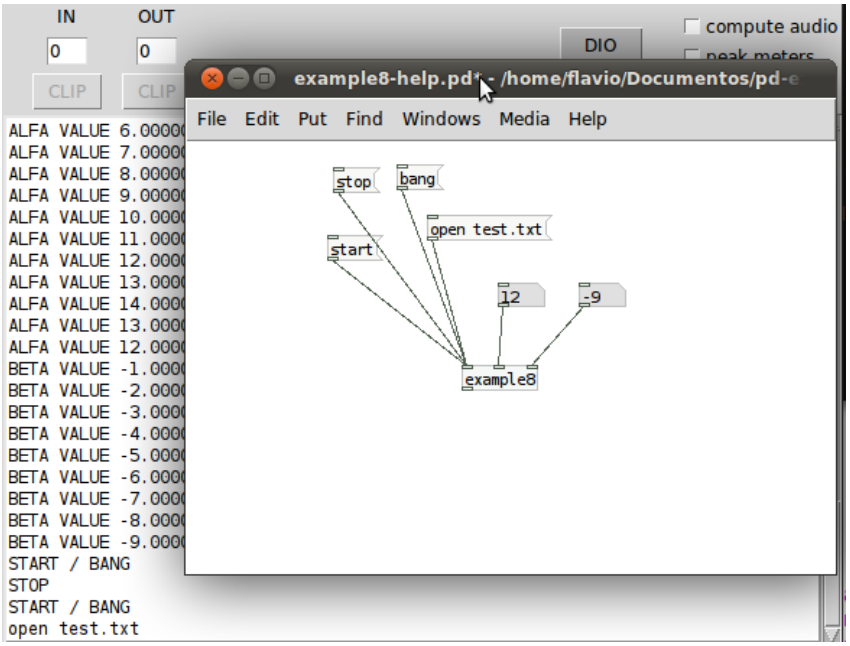

Figura 5.3: Mais inlets.

```
8   post("BANGED!");
9   outlet_bang(x->my_outlet); // Bang my outlet
10  }
11
12  // Constructor of the class
13  void * example6_new(t_symbol * arg1, t_floatarg arg2) {
14    t_example6 *x = (t_example6 *) pd_new(example6_class);
15    x->my_outlet = outlet_new(&x->x_obj, gensym("bang"));
16    return (void *) x;
17  }
18
19  void example6_setup(void) {
20      example6_class = class_new(gensym("example6"),
21        (t_newmethod) example6_new, // Constructor
22        0,
23        sizeof (t_example6),
24        CLASS_DEFAULT,
25        A_DEFFLOAT, // First Constructor parameter
26        A_DEFSYMBOL, // Second Consctructo parameter
27        0); // LAST argument is ALWAYS zero
28      class_addbang(example6_class, example6_bang);
29  }
```

Um outlet deve ser definido na estrutura do objeto e instanciado pela função `outlet_new()`, definindo também o tipo do átomo associado. No caso deste exemplo, o outlet é do tipo `bang` e dispara um bang toda vez que recebe um bang (veja a figura 5.4).

Há funções definidas para enviar vários tipos diferentes para um outlet. São elas:

- void outlet_bang(t_outlet *x);
- void outlet_pointer(t_outlet *x, t_gpointer *gp);
- void outlet_float(t_outlet *x, t_float f);
- void outlet_symbol(t_outlet *x, t_symbol *s);

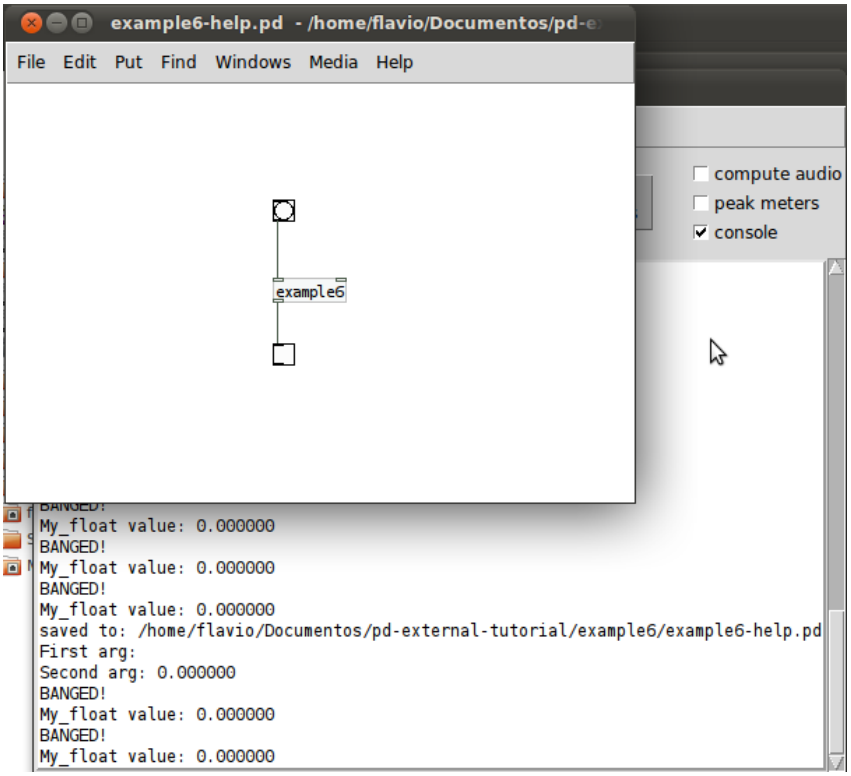

Figura 5.4: Um external bem útil que recebe um bang e envia um bang.

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

Tem um tipo de inlet/outlet que utiliza um sistema de proxy para que os mesmos possam ser definidos on the fly. Ele baseia-se em definir um método em outro objeto (o proxy) e associar este novo inlet ao proxy. Assim podemos instanciar vários proxys e tratar vários inlets passivos, por exemplo.

Capítulo 6

Processamento de Sinais Digitais

Precisa mudar o nome dos exemplos para exemplo~. Neste caso, a função setup deve ser renomeada para "tilde_setup"

Enfim chegamos no processamento de áudio propriamente dito: *Digital Signal Processing* ou processamento de sinal digital. O Pure Data possui inlets e outlets específicos para o processamento de sinal. É fácil reconhecer: eles são pintados de cinza escuro.

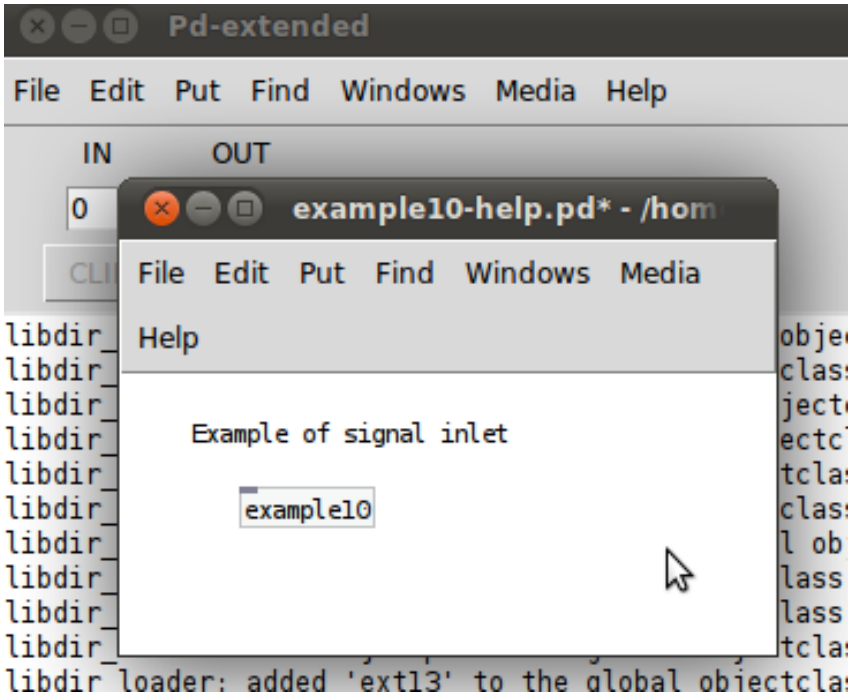

Figura 6.1: Primeiro Inlet DSP

6.1 Primeiro inlet para DSP

Para realizar DSP no Pure Data é necessário alguns cuidados (veja o exemplo 10). Primeiramente, é necessário possuir na estrutura de dados um atributo do tipo `t_float` para armazenar o valor de entrada do inlet.

```
1 typedef struct _example10 {
2     t_object x_obj;
3     t_float x_f; /* inlet value when set by message */
4 } t_example10;
```

Exemplo 10

Se o *external* necessita de apenas um inlet DSP, a macro `CLASS_MAINSIGNALIN()` define um inlet DSP no primeiro inlet da esquerda. Para esta macro funcionar, é necessário que a classe seja do tipo `CLASS_DEFAULT`, e que um método seja associado à mensagem "dsp", de forma que será executado quando o DSP for iniciado. A forma de declaração de outros inlets DSP será vista logo adiante.

```
5  void example10_setup(void) {
6    example10_class = class_new(gensym("example10"),
7      (t_newmethod) example10_new, // Constructor
8      (t_method) example10_destroy, // Destructor
9      sizeof (t_example10), CLASS_DEFAULT, A_GIMME, 0);
10   // this declares the leftmost, "main" inlet
11   // as taking signals.
12   CLASS_MAINSIGNALIN(example10_class, t_example10, x_f);
13   class_addmethod(example10_class, (t_method) example10_dsp,
       gensym("dsp"), 0);
14 }
```

Exemplo 10

O próximo passo é definir o método DSP que associamos na função `_setup()`:

```
16 static void example10_dsp(t_example10 *x, t_signal **sp){
17   // adds a method for dsp
18   dsp_add(example10_perform, 3, sp[0]->s_vec, sp[0]->s_n, x);
19 }
```

Exemplo 10

Todo método de processamento de sinais associado através da função `dsp_add()` será executado em todo ciclo DSP enquanto o processamento de sinais estiver ligado para o Pure Data ou para aquela janela específica, através do objeto `switch` . Por isto, cuidado com alocações de memória, inicialização de variáveis, etc.

Neste exemplo, o método `example10_perform()` é associado ao processamento de áudio do Pure Data e será chamada em cada execução do ciclo DSP com 3 argumentos: o endereço para o sinal de entrada (`sp[0]->s_vec`), a quantidade de amostras no bloco (`sp[0]->s_n`), e o endereço da estrutura que contém sua instância (`x`).

Podemos passar para o método perform quaisquer parâmetros em qualquer ordem. Só é importante e óbvio que devemos lembrar quais parâmetros foram passados e em qual ordem. O próximo passo é criar o método perform propriamente dito.

```
20 static t_int * example10_perform(t_int *w){
21   t_float     *in = (t_float *) (w[1]);
22   int          n  = (int) (w[2]);
23   t_example10 *x  = (t_example10 *) (w[3]);
24   // do something ...
25   return (w + 4); // next block's address
26 }
```

Exemplo 10

O método perform receberá como parâmetro um vetor com os valores definidos no método `dsp_add()`. A posição 0 deste vetor sempre conterá o endereço para a própria função `_perform()`. Nas próximas posições devemos rever o que definimos na função DSP acima. Na posição 1, o sinal de entrada; na posição 2 o tamanho do vetor de entrada e na posição 3 a estrutura de dados correspondente ao objeto. Note que podemos enviar estes dados em outra ordem ou ainda enviar outros dados para a função `perform()`. Este método deve retornar a próxima posição do vetor, ou seja, o endereço dado na chamada somado com a quantidade de atributos do método mais um.

Pergunta: O que raios há na posição w[0]? Resposta: o endereço da própria função perform.

6.2 Vários inlets DSP

É possível definir vários inlets de DSP para um *external* (veja o exemplo 11). A criação de inlets adicionais não é feita no método `_setup()` mas sim no construtor do objeto. Quanto ao primeiro inlet, só é necessário definí-lo explicitamnete no construtor se a classe não for do tipo `CLASS_DEFAULT`.

```
1  // Constructor of the class
2  void * example11_new(t_symbol *s, int argc, t_atom * argv) {
3      t_example11 *x = (t_example11 *) pd_new(example11_class);
4      inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
           // second signal inlet
5      inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
           // third signal inlet
6      inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
           // fourth signal inlet
7      return (void *) x;
8  }
```

Exemplo 11

Neste caso, a utilização do método `class_addmethod` é exatamente igual à anterior, a menos de uma mudança na quantidade de parâmetros por causa da quantidade de inlets:

```
9  static void example11_dsp(t_example11 *x, t_signal **sp){
10   dsp_add(example11_perform, 6, sp[0]->s_vec, sp[0]->s_n, x);
11 }
```

Exemplo 11

Note que precisamos agora alterar a quantidade de parâmetros passadas ao método `_perform()`, que ficará assim:

```
12 static t_int * example11_perform(t_int *w){
13   t_float *in1 = (t_float *)(w[1]);
14   t_float *in2 = (t_float *)(w[2]);
15   t_float *in3 = (t_float *)(w[3]);
16   t_float *in4 = (t_float *)(w[4]);
17   int n = (int)(w[5]);
18   t_example11 *x = (t_example11 *)(w[6]);
19   return (w + 7); // proximo bloco
20 }
```

Exemplo 11

Neste ponto, este external deve ter uma aparência como na figura 6.2.

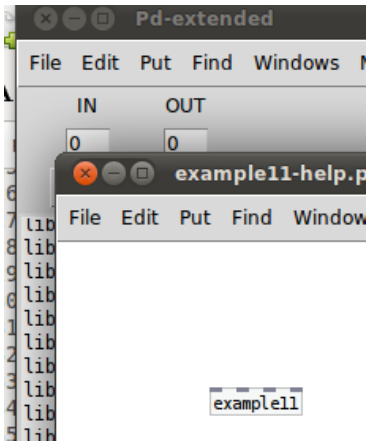

Figura 6.2: Vários inlets DSP.

6.3 Primeiro outlet DSP

A criação dos outlets é feita no construtor do external (veja o exemplo 12) e não é necessário adicionar os outlets à estrutura da classe.

```
1  // Constructor of the class
2  void * example12_new(t_symbol *s, int argc, t_atom * argv) {
3    t_example12 *x = (t_example12 *) pd_new(example12_class);
4    outlet_new(&x->x_obj, &s_signal); // first signal outlet
5    outlet_new(&x->x_obj, &s_signal); // second signal outlet
6    outlet_new(&x->x_obj, &s_signal); // third signal outlet
7    outlet_new(&x->x_obj, &s_signal); // fourth signal outlet
8    return (void *) x;
9  }
```

Exemplo 12

A definição do método `_perform()` será idêntica ao do exemplo anterior, quando criamos quatro inlets:

```
10 static void example12_dsp(t_example12 *x, t_signal **sp){
11   dsp_add(example12_perform, 6, sp[0]->s_vec, sp[0]->s_n, x);
12 }
```

Exemplo 12

O método perform também será quase idêntico ao do exemplo anterior, porém recebendo quatro outlets:

```
13 static t_int * example12_perform(t_int *w){
14   t_float *out1 = (t_float *)(w[1]);
15   t_float *out2 = (t_float *)(w[2]);
16   t_float *out3 = (t_float *)(w[3]);
17   t_float *out4 = (t_float *)(w[4]);
18   int n = (int)(w[5]);
19   t_example12 *x = (t_example12 *)(w[6]);
20   return (w + 7); // proximo bloco
21 }
```

Exemplo 12

O resultado pode ser visto na figura 6.3.

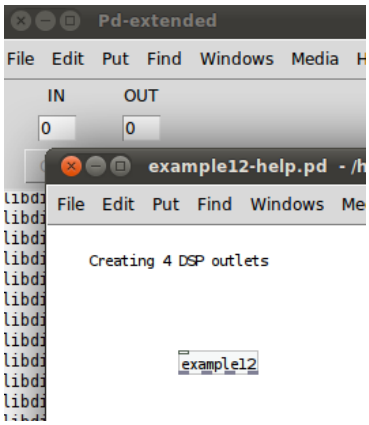

Figura 6.3: Primeiro Outlet DSP.

6.4 Inlets e outlets DSP

Nosso próximo exemplo (veja o exemplo 13) mistura no mesmo objeto inlets e outlets DSP, o que é bastante comum. Neste ponto, deve estar mais ou menos claro como é feita a construção de um objeto assim. Não é necessário guardar os endereços dos inlets e outlets na estrutura de dados que representa o objeto. É necessário apenas criar os inlets e outlets no construtor (lembre-se que o primeiro inlet já foi criado no método setup. Ele é mágico!).

```
1  // Constructor of the class
2  void * example13_new(t_symbol *s, int argc, t_atom * argv) {
3      t_example13 *x = (t_example13 *) pd_new(example13_class);
4
5      inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
           // second signal inlet
6      inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
           // third signal inlet
7      inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
           // fourth signal inlet
8
9      outlet_new(&x->x_obj, &s_signal); // first signal outlet
10     outlet_new(&x->x_obj, &s_signal); // second signal outlet
11     outlet_new(&x->x_obj, &s_signal); // third signal outlet
12     outlet_new(&x->x_obj, &s_signal); // fourth signal outlet
13     return (void *) x;
14 }
```

Exemplo 13

No método seguinte associamos o método `_perform()` à cadeia DSP do Pure Data:

```
15 static void example13_dsp(t_example13 *x, t_signal **sp){
16   dsp_add(example13_perform, 10, sp[0]->s_vec, sp[0]->s_n, x);
17 }
```

Exemplo 13

No método `_perform()` recebemos como argumento um bloco de memória que contém primeiro os buffers de entrada e em seguida os buffers de saída:

```
18 static t_int * example13_perform(t_int *w){
19   t_float *in1 = (t_float *)(w[1]);
20   t_float *in2 = (t_float *)(w[2]);
21   t_float *in3 = (t_float *)(w[3]);
22   t_float *in4 = (t_float *)(w[4]);
23   t_float *out1 = (t_float *)(w[5]);
24   t_float *out2 = (t_float *)(w[6]);
25   t_float *out3 = (t_float *)(w[7]);
26   t_float *out4 = (t_float *)(w[8]);
27   int n = (int)(w[9]);
28   t_example13 *x = (t_example13 *)(w[10]);
29   return (w + 11); // proximo bloco
30 }
```

Exemplo 13

O resultado pode ser visto na figura 6.4.

6.5 Inlets e outlets DSP criados dinamicamente

É possível definir através de parâmetros para o construtor a quantidade de inlets e/ou outlets DSP que um external deve possuir. Isso significa que o número de inlets e outlets é definido dinamicamente, em tempo de execução, através de um argumento. Para isto, existe mais de uma possibilidade de implementação.

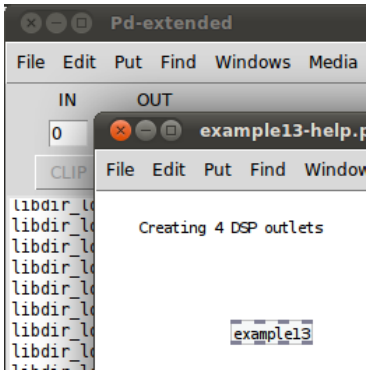

Figura 6.4: Vários inlets e outlets DSP.

A primeira possibilidade consiste em passarmos para o construtor a informação de quantos inlets e outlets teremos na função dsp (veja o exemplo 17) e armazenarmos na estrutura de dados que contém o objeto do *external*.

```
1  typedef struct _example17 {
2     t_object x_obj;
3     t_int outlet_counter;
4     t_int inlet_counter;
5  } t_example17;
6
7  // Constructor of the class
8  void * example17_new(t_symbol *s, t_floatarg inlet_counter,
       t_floatarg outlet_counter) {
9    t_example17 *x = (t_example17 *) pd_new(example17_class);
10   int i;
11   post("inlet counter: %f", inlet_counter);
12   post("outlet counter: %f", outlet_counter);
13   x->inlet_counter = inlet_counter;
14   x->outlet_counter = outlet_counter;
15   for(i = 0 ; i < inlet_counter ; i++)
16     inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
17   for(i = 0 ; i < outlet_counter ; i++)
18     outlet_new(&x->x_obj, &s_signal);
19   return (void *) x;
20 }
```

Exemplo 17

Na passagem de parâmetro para o DSP usamos estas variáveis para contar quantos parâmetros serão usados.

```
21 static t_int * example17_perform(t_int *w){
22   t_example17 *x = (t_example17 *)(w[1]);
23   int n = (int)(w[2]), i;
24   for(i = 0 ; i < x->inlet_counter ; i++) {
25     // DO something with the input
26   }
27   for(i = 0 ; i < x->outlet_counter ; i++) {
28     // DO something with the output
29   }
30   return (w + (2 + x->inlet_counter + x->outlet_counter +1));
31 }
32
33 static void example17_dsp(t_example17 *x, t_signal **sp){
34   int qtd = x->inlet_counter + x->outlet_counter + 2;
```

```
35    dsp_add(example17_perform, qtd , x, sp[0]->s_n, sp[0]->s_vec);
36 }
```

Exemplo 17

A segunda opção é usar outro método que baseia-se no modelo de alocação de memória do pd. Criamos um vetor e apontamos este vetor para o dado da entrada / saída do external. Assim podemos utilizar a estrutura amarrada ao external para produzir / consumir o dado. Veja o exemplo 19.

```
1  void example19_setup(void) {
2    example19_class = class_new(gensym("example19"),
3      (t_newmethod) example19_new, // Constructor
4      (t_method) example19_destroy, // Destructor
5      sizeof (t_example19),
6      CLASS_NOINLET,
7      A_DEFFLOAT, // # of inlets
8      0); // LAST argument is ALWAYS zero
9    class_addmethod(example19_class, (t_method) example19_dsp,
       gensym("dsp"), 0);
10 }
```

Exemplo 19

Neste ponto, esta solução é bastante parecida com a anterior e poderia ser usada também com uma quantidade variável de outlets. A alteração está na estrutura da classe.

```
11 typedef struct _example19 {
12     t_object x_obj;
13     t_sample * outvec[64]; // 64 is a magic number!
14     t_int inlet_counter;
15 } t_example19;
```

Exemplo 19

Ok, 64 não é um número mágico de verdade. Ele é o tamanho de bloco padrão do Pure Data. Para que esta solução seja genérica o suficiente é melhor usar a função `sys_getblksize()` para obter o tamanho do bloco e alocar a quantidade correta de espaço. Da forma em que está, corremos o risco de obter erros de leitura ou escrita em lugares sem permissão.

```
16 void * example19_new(t_symbol *s, t_floatarg inlet_counter,
     t_floatarg outlet_counter) {
17   t_example19 *x = (t_example19 *) pd_new(example19_class);
18   post("inlet counter %f", inlet_counter);
19   x->inlet_counter = inlet_counter;
20   int i;
21   for(i = 0 ; i < inlet_counter ; i++)
22     inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
23   return (void *) x;
24 }
```

Exemplo 19

Como no exemplo anterior, o construtor cria a quantidade de outlets passada como argumento na criação do objeto. Aqui, poderíamos utilizar `malloc()` para alocar o vetor com os dados de maneira inteligente ao invés de usar o 64 mágico.

```
25 static void example19_dsp(t_example19 *x, t_signal **sp){
26   short i;
27   for (i = 0; i < x->inlet_counter; i++)
28     x->outvec[i] = sp[i]->s_vec; // get outlets addresses
29   dsp_add(example19_perform, 1 , x);
30 }
```

Exemplo 19

O método DSP não fará mais que passar o próprio objeto ao método `_perform()`. Na verdade, poderia não passar nem o objeto.

```
31 static t_int * example19_perform(t_int *w){
32   t_example19 *x = (t_example19 *)(w[1]);
33   // Call DSP function or start DSP thread
34   return (w + 2);
35 }
```

Exemplo 19

A função perform não precisa fazer nada mais que despertar o processo consumidor e liberar o processamento no Pure Data. E este é um ponto que considero importante para levantar outro questionamento.

O Pd precisa que o processamento dos blocos termine em determinado tempo. Qual seria este tempo? Para um bloco de 64 amostras e taxa de amostragem 44.100 amostras por segundo, é necessário que todo o processamento para um bloco termine em um pouco mais de 1 milisegundo. Parece rápido mas seu computador é mais rápido que isto. Criar um processo consumidor que libere o bloco de processamento pode ser uma abordagem melhor para processos baseados em fluxo, como escrita em arquivos. Para isto precisamos utilizar outra função em outra thread e este é nosso próximo assunto.

Vale a pena mostrar como fazer o loop entre as amostras ou é óbvio?

Capítulo 7

Multithreading

Como vimos no capítulo anterior, o bloco de processamento do Pd possui um limite de tempo para a execução. É possível utilizar threads para separar processos que consumam mais tempo do que o o período do bloco DSP, como por exemplo no vaso de processos do tipo produtor/consumidor.

A programação multithread não é exatamente comum no Pure Data mas pode ser útil para várias coisas como escrita em arquivo, envio de dados para a rede ou atualização da interface gráfica (como veremos no próximo capítulo).

Apesar de existirem várias bibliotecas para programação paralela, como por exemplo a simples utilização do comando fork do GNU/Linux, é desejável que os externals do Pure Data sejam compatíveis com diversos sistemas operacionais. Nos repositórios do Miller Puckette, autor do Pure Data, encontramos patches nos quais ele utiliza threads POSIX implementadas pela biblioteca `pthread` [1].

Note que esta solução, que em muito se aproxima da última forma de criar inlets e outlets DSP implica em não trabalharmos mais em tempo real. Implementações deste tipo não podem ser pensadas para processamentos aonde a entrada de áudio será processada e devolvida na saída de áudio no mesmo bloco de processamento do Pd.

7.1 Criando threads

Para utilizar a biblioteca de threads do POSIX é necessário incluir o arquivo de cabeçalho correspondente. Em seguida, para armazenar as threads que criamos, utilizamos uma variáveis que armazenam threads (veja o exemplo 20).

```
1 #include <pthread.h>
2 ...
3 typedef struct _example20 {
4     t_object x_obj;
5     pthread_t example20_thread;
6 } t_example20;
```

O próximo passo é criar uma função associada a esta thread e a enfim lançar a thread. O lançamento da thread pode ser feito na função DSP. Isto implica criar e iniciar uma thread nova em cada ciclo DSP do Pd.

```
1 void * example20_thread_function(void * arg) {
2   t_example20 *x = (t_example20 *) arg;
3   while(1){
4     // DO SOMETHING
5     printf("Threading running!\n");
6     sleep(1);
7   }
8   return 0;
9 }
```

[1] Para maiores informações, visite: https://computing.llnl.gov/tutorials/pthreads/

```
10
11 static void example20_dsp(t_example20 *x, t_signal **sp){
12   pthread_create(&x->example20_thread, NULL,
         example20_thread_function, x);
13   dsp_add(example20_perform, 1 , x);
14 }
```

A função de criação da thread recebe o endereço da variável aonde a thread será armazenada, os atributos da thread sendo criada[2], a função de inicialização associada a esta thread e os argumentos passados para esta função.

Caso seja passado mais de um argumento, é recomendado que se crie uma estrutura de dados e que esta seja passada como argumento para a thread.

7.2 Gerenciamento de threads

Há diversas funções para o gerenciamento de threads, definidas no arquivo de cabeçalho `pthread.h`. Entre elas:

- `pthread_detach(threadid)`: Indica para a implementação que o armazenamento da thread pode ser recuperado quando a mesma se encontra terminada
- `pthread_join(threadid,status)`: Indica para o trecho de código que chamou a thread que o mesmo deve esperar que a mesma tenha terminado sua execução.
- `pthread_exit(void *value_ptr)`: Encerra a execução de uma thread e libera sua alocação de memória.

Em princípio, threads POSIX não possuem funções para interromper e continuar a execução. Apesar disto, é possível implementar estes comandos por meio de mutexes, como veremos a seguir.

7.3 Controle de concorrência

Uma das dificuldades de utilização de threads é o controle da concorrência por recursos entre threads. Em situações de race condition, é necessário que controlemos o acesso de threads concorrentes a trechos de código que acessam dados comuns. Isto é feito por meio de mutex (mutual exclusion), sistemas de controle atômicos que garantem que apenas uma thread será executada sobre um trecho de código por vez.

Um mutex é definido da seguinte forma:

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
3 int play = 0;
```

O controle ao trecho de código pode ser feito da seguinte forma:

```
1 for(;;) { /* Playback loop */
2   pthread_mutex_lock(&lock);
3   while(!play) { /* We're paused */
4     pthread_cond_wait(&cond, &lock); /* Wait for play signal */
5   }
6   pthread_mutex_unlock(&lock);
7   /* Continue playback */
8 }
```

[2]No caso, passando `NULL` serão utilizados os atributos padrão. Para uma lista completa dos atributos, visite: http://sourceware.org/pthreads-win32/manual/pthread_attr_init.html

Creio que seria mais útil um exemplo funcional...

7.4 Controle via Pure Data

Além de podermos trabalhar threads para nosso external, na biblioteca do Pd há funções para que possamos criar mutex no Pd. São elas:

- `void sys_lock(void);`
- `void sys_unlock(void);`
- `int sys_trylock(void);`

Capítulo 8

Externals com GUI - Usando o Tcl/Tk

O Pd permite que externals possuam interfaces gráficas mais rebuscadas que as simples caixinhas que o mesmo desenha. Isto pode ser feito adicionando código Tcl/tk ao external. Isto porque a GUI do próprio Pd é feita nesta linguagem.

8.1 Iniciando no Tcl/Tk

O Tcl (Tool Command Language) é uma linguagem de programação dinâmica bastante poderosa e simples de ser utilizada. Tk é um conjunto de ferramentas para construção de GUI de aplicações desktop e é a GUI padrão não apenas do TCL mas de várias outras linguagens e pode ser executada nativamente em vários sistemas operacionais modernos como Windows, Mac OS X, Linux, entre outros [1].

O Tk possui vários objetos prontos para GUI como botões, labels, janelas, checkbox, entre outros. Os objetos criados devem ser armazenados em uma variável. Toda variável em Tk possui um nome que inicia com ponto (.). Após criar o objeto e definir seus atributos, basta que o mesmo seja empacotado (pack).

```
1 label .hello -text "Hello World"
2 pack .hello
```

Uma vez que um programa tk esteja pronto, basta salvá-lo com a extensão .tcl e utilizar um interpretador para executá-lo. Um exemplo deste interpretador no Linux é o wish. Assim, salvando o exemplo anterior com o nome de helloWorld.tcl e executando

```
1 wish helloWorld.tcl
```

Teremos o resultado:

No diretório tk deste tutorial temos exemplos mais interessantes de GUI com Tk mas a prática desta linguagem vai além do escopo deste tutorial. Um tutorial mais completo de Tcl/Tk pode ser encontrado em [2] e uma lista dos objetos e parâmetros pode ser encontrada em [3].

8.2 Escrevendo externals com GUI

A primeira necessidade para implementar um external com GUI é a inclusão da biblioteca g_canvas.h. Esta biblioteca encontra-se disponível em [4] e nela estarão as funcionalidades necessárias para que o Pd

[1] Visite: http://www.tcl.tk/ para maiores informações
[2] http://www.bin-co.com/tcl/tutorial/
[3] http://www.tkdocs.com/widgets/index.html
[4] http://www.koders.com/c/fid97160440CD236854358462C336536646E0933C46.aspx

Figura 8.1: Hello World no Tcl/Tk

desenhe nossas GUI. Veja que não queremos alterar a GUI do Pd mas apenas fazer uma GUI para o external. Isto significa que teremos o cuidade de pedir ao Pd que desenhe nossa GUI.

O segundo passo é definirmos uma variável para armazenar o comportamento do nosso objeto (Veja o exemplo 14).

```
1 t_widgetbehavior widgetbehavior; // This represents the external
    GUI
```

Nesta variável teremos as funções definidas para o que acontece com nosso objeto gráfico. Isto deve ser feito no método setup do objeto.

```
1  void example14_setup(void) {
2      example14_class = class_new(gensym("example14"),
3              (t_newmethod) example14_new, // Constructor
4              (t_method) example14_destroy, // Destructor
5              sizeof (t_example14),
6        CLASS_DEFAULT,
7        A_GIMME, // Allows various parameters
8              0); // LAST argument is ALWAYS zero
9
10     // The external GUI rectangle definition
11     widgetbehavior.w_getrectfn = my_getrect;
12     //How to make ir visible / invisible
13     widgetbehavior.w_visfn= my_vis;
14     //what to do whe moved
15     widgetbehavior.w_displacefn= my_displace;
16     // What to do when selected
17     widgetbehavior.w_selectfn= my_select;
18     // What to do when active
19     widgetbehavior.w_activatefn = my_activate;
20     // What to do when deleted
21     widgetbehavior.w_deletefn = my_delete;
22     // What to do when clicked
23     widgetbehavior.w_clickfn = my_click;
24
25     // What about object properties?
26     class_setpropertiesfn(example14_class, my_properties);
27     // How to save its properties with the patch?
28     class_setsavefn(example14_class, my_save);
29
30     //Associate the widgetbehavior with the class
31     class_setwidget(example14_class, &widgetbehavior);
32
33 }
```

Além de definir as funções callback da GUI é necessário associar o widgetbehavior a classe. Uma vez que isto for feito, o Pd não mais irá renderizar a famosa caixinha. A partir disto a responsabilidade de renderizar a GUI passa a ser do programador.

Vamos ver as funções criadas para desenhar uma GUI.

```
1  // THE BOUNDING RECTANGLE
2  static void my_getrect(t_gobj *z, t_glist *glist, int *xp1, int *
       yp1, int *xp2, int *yp2){
3    // This function is always called.
4    // Better do not put a post here...
5    // post("GETRECT");
6    t_example14 *x = (t_example14 *)z;
7    *xp1 = x->x_obj.te_xpix;
8    *yp1 = x->x_obj.te_ypix;
9    *xp2 = x->x_obj.te_xpix + 30;
10   *yp2 = x->x_obj.te_ypix + 50;
11 }
```

Esta função recebe ponteiros para inteiros que deverão ser apontados para os valores que queremos definir como o retângulo do nosso objeto. Veja que isto não é necessariamente o tamanho do retângulo do objeto gráfico mas aonde o mesmo poderá ser clicado na tela. Um exemplo disto é o comentário do Pd. Apesar do mesmo as vezes ficar maior, sua área clicável é sempre um quadradinho na esquerda. Isto significa que nem sempre a representação gráfica da área do objeto é a mesma da sua área de desenho.

```
1  // MAKE VISIBLE OR INVISIBLE
2  static void my_vis(t_gobj *z, t_glist *glist, int vis){
3    t_example14 *x = (t_example14 *)z;
4
5    // takes the Canvas to draw a GUI
6    t_canvas * canvas = glist_getcanvas(glist);
7
8    if(vis){ // VISIBLE
9    post("VISIBLE");
10
11         sys_vgui(".x%lx.c create rectangle %d %d %d %d -tags %xrr
               -fill #FF0000\n",
12     glist_getcanvas(glist),
13     x->x_obj.te_xpix,
14     x->x_obj.te_ypix,
15     x->x_obj.te_xpix + 70,
16     x->x_obj.te_ypix + 50,
17     x
18     );
19        sys_vgui(".x%x.c create text %d %d -text {example14} -
              anchor w  -tags %xlb\n",
20                 canvas,
21     x->x_obj.te_xpix + 2,
22     x->x_obj.te_ypix + 12,
23     x);
24   }else{ // INVISIBLE
25     post("INVISIBLE");
26     sys_vgui(".x%x.c delete %xrr\n",canvas, x);
27     sys_vgui(".x%x.c delete %xlb\n",canvas, x);
28   }
29 //  canvas_fixlinesfor(glist, (t_text *)x);
```

```
30 }
```

Deixar o objeto visível ou invisível significa pedir a GUI do Pd que desenhe ou apague um desenho. Neste caso nosso desenho é um retângulo vermelho. Usamos o nome da própria instância como nome do objeto Tk para evitar que sobrescrevamos o nome de algum outro componente gráfico do Pd. Também desenhamos um texto com o nome do objeto pois o Pd não fará isto. Caso nosso objeto se torne invisível, temos de remover ambos do canvas.

```
1  // WHAT TO DO IF SELECTED?
2  static void my_select(t_gobj *z, t_glist *glist, int state){
3    t_example14 *x = (t_example14 *)z;
4    if (state) {
5    post("SELECTED");
6          sys_vgui(".x%x.c create rectangle %d %d %d %d -tags %xSEL
               -outline blue\n",
7      glist_getcanvas(glist),
8      x->x_obj.te_xpix,
9      x->x_obj.te_ypix,
10     x->x_obj.te_xpix + 70,
11     x->x_obj.te_ypix + 50,
12     x
13     );
14   }else {
15   post("DESELECTED");
16   sys_vgui(".x%x.c delete %xSEL\n",glist_getcanvas(glist), x);
17   }
18 }
```

O que acontece quando um objeto do Pd é selecionado? Ele ganha um contorno azul. O que acontece quando nosso objeto é selecionado? Nada. Por isto desenhamos aqui um contorno azul. Quando ele é deselecionado, removemos o contorno azul.

```
1  // DISPLACE IT
2  void my_displace(t_gobj *z, t_glist *glist,int dx, int dy){
3    post("MOVED");
4    t_canvas * canvas = glist_getcanvas(glist);
5    t_example14 *x = (t_example14 *)z;
6    x->x_obj.te_xpix += dx; // x movement
7    x->x_obj.te_ypix += dy; // y movement
8
9          sys_vgui(".x%lx.c coords %xSEL %d %d %d %d \n", //MOVE O
               SELECIONADO
10     canvas,
11     x,
12     x->x_obj.te_xpix,
13     x->x_obj.te_ypix,
14     x->x_obj.te_xpix + 70,
15     x->x_obj.te_ypix + 50
16     );
17         sys_vgui(".x%x.c coords %xrr %d %d %d %d\n",canvas,x,x->
               x_obj.te_xpix,x->x_obj.te_ypix,x->x_obj.te_xpix + 70,x
               ->x_obj.te_ypix + 50);
18         sys_vgui(".x%x.c coords %xlb %d %d \n",canvas,x,x->x_obj.
               te_xpix + 2, x->x_obj.te_ypix + 12);
19
20   canvas_fixlinesfor(glist, (t_text *)x);
21 }
```

O que fazer quando movemos um objeto. Aqui será necessário mover todos os objetos pois não temos um container que possui todos os objetos. Inclusive é necessário mover o contorno azul que desenhamos quando o mesmo se tornou selecionado.

```
1 // What to do if activated?
2 static void my_activate(t_gobj *x, struct _glist *glist, int
      state){
3   post("Activated");
4 }
```

O método será executado quando um objeto se tornar ativo.

```
1 static int my_click(t_gobj *z, struct _glist *glist, int xpix,
      int ypix, int shift, int alt, int dbl, int doit){
2   post("Clicked xpix:%d ypix:%d shift:%d alt:%d dbl:%d doit:%d",
       xpix,ypix, shift, alt, dbl, doit);
3   return 0;
4 }
```

O método click é retornado com qualquer evento do mouse. Caso ele seja clicado, o mesmo receberá o valor 1 no parâmetro doit. Vale lembrar que este método só será executado quando o Pd não estiver no modo de edição.

```
1 static void my_delete(t_gobj *z, t_glist *glist){
2   t_text *x = (t_text *)z;
3   canvas_deletelinesfor(glist_getcanvas(glist), x);
4   post("Object deleted!");
5 }
```

Este método será o destrutor da GUI.

```
1 void my_save(t_gobj *c, t_binbuf *b){
2   post("SAVE");
3 }
```

Quando salvamos o *patch* pode ser necessário armazenar parâmetros para recuperar nosso external como GUI na mesma situação. Este método recebe um buffer aonde poderá armazenar dados que serão devolvidos quando o mesmo for recriado. Note que isto não está associado com a GUI mas com a classe.

Nunca implementei Salvar. Parece legal...

```
1 void my_properties(t_gobj *c, t_glist *list){
2   post("PROPERTIES");
3 }
```

Caso o usuário pressione o botão contrário ele pode alterar as propriedades do objeto. Este método permite definir a lista de propriedades que o objeto aceita. Note que isto não está associado com a GUI mas com a classe.

Ainda não implementei propriedades.

Caso nosso objeto tenha sido criado corretamente, ele ficará como a seguir

Note que este objeto foi definido em sua criação como um CLASS_DEFAULT, o que implica que o mesmo possui um inlet. Este inlet existe e está funcionando e aceitando conexões de outros objetos,

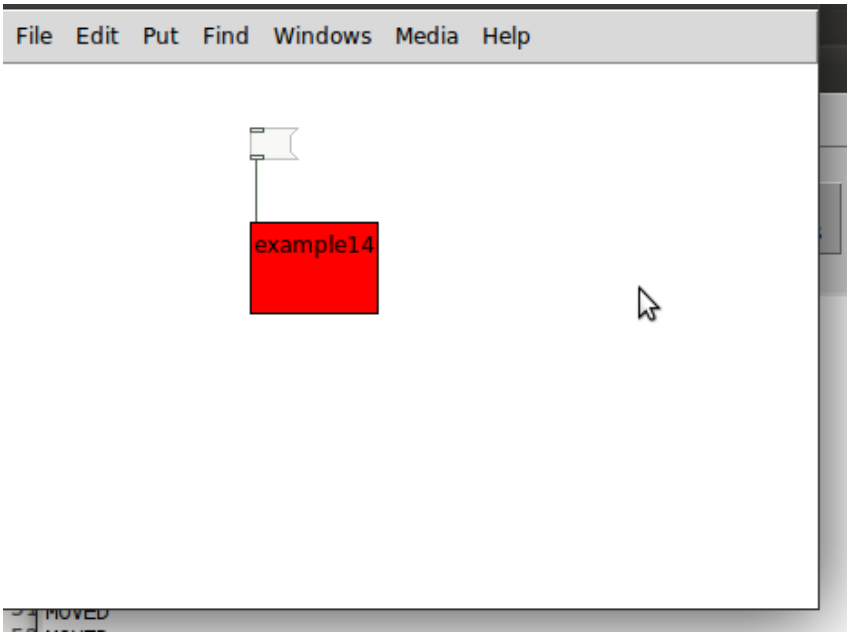

Figura 8.2: Adicionando GUI tk.

mensagens e números. Só não aceita conexões DSP pois as mesmas não foram definidas para este objeto. Mesmo assim o Pure Data não irá desenhá-lo pois precisamos definir tudo na GUI. Caso queira um retângulo em cima para mostrar ao usuário que temos um inlet, temos que desenhá-lo, movê-lo quando necessário e também apagá-lo quando o objeto se tornar invisível.

8.3 Adicionando componentes gráficos

O Objeto Tk que o Pd nos disponibiliza é um canvas. Um canvas é, em princípio, uma tela de pintura. Em um canvas podemos adicionar linhas, ovals, pontos, textos, retângulos e janelas. É por se tratar de um canvas e não de uma janela que não temos o conceito de um container para os objetos. Desta maneira, para adicionarmos um componente como um botão ou um slider, é necessário adicionarmos uma janela ao canvas para que a mesma abrigue este componente gráfico. Há vários componentes gráficos que podem ser adicionados em um external. Vejamos um exemplo (exemplo 15).

```
1  // MAKE VISIBLE OR INVISIBLE
2  static void my_vis(t_gobj *z, t_glist *glist, int vis){
3    t_example15 *x = (t_example15 *)z;
4    t_canvas * canvas = glist_getcanvas(glist);
5
6    if(vis){ // VISIBLE
7    // Define the tk/tcl commands / functions
8    sys_vgui("proc do_something {} {\n set name [.x%x.c.s%xtx get]\
        n puts \"OIA: $name\" \n}\n",canvas,x);
9    sys_vgui("proc do_otherthing {val} {\n set name [.x%x.c.s%xtx
        get]\n puts \"OIA: $name\" \n}\n",canvas,x);
10       // The text field
11   sys_vgui("entry .x%x.c.s%xtx -width 12 -bg yellow \n", canvas,x
        );
12   // The button
13   sys_vgui("button .x%x.c.s%xbb -text {click} -command
        do_something\n", canvas,x);
14   // The radio button
15   sys_vgui("radiobutton .x%x.c.s%xrb -value 1 -command
        do_something\n", canvas,x);
16   // The h slider
```

```
17    sys_vgui("scale .x%x.c.s%xsb -orient horizontal -command
          do_otherthing \n", canvas,x);
18    // A checkbutton
19    sys_vgui("checkbutton .x%x.c.s%xcb -foreground blue -background
          yellow -command do_something\n", canvas,x);
20    // The red rectangle
21          sys_vgui(".x%lx.c create rectangle %d %d %d %d -tags %xrr
                -fill #FF0000\n",
22      glist_getcanvas(glist),
23      x->x_obj.te_xpix,
24      x->x_obj.te_ypix,
25      x->x_obj.te_xpix + 170,
26      x->x_obj.te_ypix + 150,
27      x
28      );
29    // A window to the button (bb)
30    sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
          s%xbb -tags %xbb\n",
31      canvas,
32      x->x_obj.te_xpix + 70,
33      x->x_obj.te_ypix + 120,
34      canvas,
35      x,
36      x);
37    // A window to the radiobutton
38    sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
          s%xrb -tags %xrb\n",
39      canvas,
40      x->x_obj.te_xpix + 70,
41      x->x_obj.te_ypix,
42      canvas,
43      x,
44      x);
45    // A window to the combo box
46    sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
          s%xcb -tags %xcb\n",
47      canvas,
48      x->x_obj.te_xpix  + 100,
49      x->x_obj.te_ypix,
50      canvas,
51      x,
52      x);
53    // A window to the slider button
54    sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
          s%xsb -tags %xsb\n",
55      canvas,
56      x->x_obj.te_xpix,
57      x->x_obj.te_ypix + 80,
58      canvas,
59      x,
60      x);
61    // A window to the text
62    sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
          s%xtx -tags %xtx\n",
63      canvas,
64      x->x_obj.te_xpix,
65      x->x_obj.te_ypix + 60,
66      canvas,
```

```
67      x,
68      x);
69    // a label field
70          sys_vgui(".x%x.c create text %d %d -text {example15~} -
               anchor w  -tags %xlb\n",
71                canvas,
72      x->x_obj.te_xpix,
73      x->x_obj.te_ypix + 40,
74      x);
75
76    }else{ // INVISIBLE
77      sys_vgui(".x%x.c delete %xbb\n",canvas, x);
78      sys_vgui(".x%x.c delete %xcb\n",canvas, x);
79      sys_vgui(".x%x.c delete %xrb\n",canvas, x);
80      sys_vgui(".x%x.c delete %xsb\n",canvas, x);
81      sys_vgui(".x%x.c delete %xrr\n",canvas, x);
82      sys_vgui(".x%x.c delete %xlb\n",canvas, x);
83      sys_vgui(".x%x.c delete %xtx\n",canvas, x);
84    }
85  }
```

Criamos os componentes gráficos e pedimos ao canvas para criar uma janela para que a mesma abrigue o componente. Assim, na hora de remover podemos remover apenas a janela. O mesmo ocorre na hora de mover. Note que além de adicionarmos componentes gráficos associamos eles a um comando. Este será nosso próximo tópico.

```
1   void my_displace(t_gobj *z, t_glist *glist,int dx, int dy){
2     t_canvas * canvas = glist_getcanvas(glist);
3     t_example15 *x = (t_example15 *)z;
4     x->x_obj.te_xpix += dx;
5     x->x_obj.te_ypix += dy;
6
7           sys_vgui(".x%lx.c coords %xSEL %d %d %d %d \n", //MOVE O
               SELECIONADO
8       glist_getcanvas(glist),
9       x,
10      x->x_obj.te_xpix,
11      x->x_obj.te_ypix,
12      x->x_obj.te_xpix + 170,
13      x->x_obj.te_ypix + 150
14      );
15          sys_vgui(".x%x.c coords %xrr %d %d %d %d\n",canvas,x,x->
               x_obj.te_xpix,x->x_obj.te_ypix,x->x_obj.te_xpix + 170,
               x->x_obj.te_ypix + 150);
16          sys_vgui(".x%x.c coords %xbb %d %d \n",canvas,x,x->x_obj.
               te_xpix,x->x_obj.te_ypix);
17          sys_vgui(".x%x.c coords %xcb %d %d \n",canvas,x,x->x_obj.
               te_xpix + 30,x->x_obj.te_ypix);
18          sys_vgui(".x%x.c coords %xsb %d %d \n",canvas,x,x->x_obj.
               te_xpix,x->x_obj.te_ypix + 80);
19          sys_vgui(".x%x.c coords %xtx %d %d \n",canvas,x,x->x_obj.
               te_xpix,x->x_obj.te_ypix + 120);
20          sys_vgui(".x%x.c coords %xrb %d %d \n",canvas,x,x->x_obj.
               te_xpix + 60,x->x_obj.te_ypix);
21          sys_vgui(".x%x.c coords %xlb %d %d \n",canvas,x,x->x_obj.
               te_xpix,x->x_obj.te_ypix + 40);
22
23    canvas_fixlinesfor(glist, (t_text *)x);
```

```
24 }
```

8.4 Adicionando comandos

Os comandos dos nossos componentes gráficos podem ser recebidos e interagirem com o external assim como o external consegue alterar os valores da GUI dependendo do que recebe em seus inlets. Veja o exemplo 21.

Para este exemplo, definimos na criação da classe no método setup() os métodos de retorno de nossa GUI.

```
1 // depois associaremos estes "tipos" de mensagem aos inlets 2 e 3
2     class_addmethod(example21_class, (t_method)example21_alfa,
          gensym("alfa"), A_DEFFLOAT,0);
3     class_addmethod(example21_class, (t_method)example21_beta,
          gensym("beta"), A_DEFFLOAT,0);
4 // metodo do botao ok
5     class_addmethod(example21_class, (t_method)example21_btok,
          gensym("btok"), A_DEFSYMBOL,0);
```

No método vis da GUI, criamos comandos.

```
1       sys_vgui("proc slide_alfa {val} {\n pd [concat example21%x
            alfa $val \\;]\n}\n",x);
2       sys_vgui("proc slide_beta {val} {\n pd [concat example21%x
            beta $val \\;]\n}\n",x);
3       sys_vgui("proc botao_ok {} {\n set name [.x%x.c.s%xtx get]\
            n pd [concat example21%x btok $name \\;]\n}\n",x->canvas
            ,x,x);
4       sys_vgui("proc botao_file_chooser {} {\n\
5                set filename [tk_getOpenFile]\n\
6               .x%x.c.s%xtx delete 0 end \n\
7               .x%x.c.s%xtx insert end $filename \n}\n",x->canvas,
                   x,x->canvas,x);
8
9   (...)
10  sys_vgui("entry .x%x.c.s%xtx -width 25 -bg white -textvariable
        \"teste\" \n", x->canvas,x);
11  sys_vgui("button .x%x.c.s%xbfc -text {...} -command
        botao_file_chooser\n", x->canvas,x);
12  sys_vgui("scale .x%x.c.s%xsb1 -length 250 -resolution 0.01 -
        from 0.5 -to 2 -orient horizontal -command slide_alfa \n", x
        ->canvas,x);
13  sys_vgui("scale .x%x.c.s%xsb2 -length 250 -resolution 0.01 -
        from 0.5 -to 2 -orient horizontal -command slide_beta \n", x
        ->canvas,x);
14  sys_vgui("button .x%x.c.s%xbt -text {start} -command botao_ok\n
        ", x->canvas,x);
15  (...)
```

Os comandos alfa, beta e btok serão executados pelo objeto pd. Desta forma pedimos ao pd que, ao chamarmos este método o mesmo chame a função associada a este symbolo anteriormente. Os objetos criados usarão estes comandos para suas ações. Note que no caso do filechooser, o comando irá associar o caminho do arquivo escolhido com nosso campo text e tudo isto será feito diretamente no Tk.

Então basta criarmos as funções associadas:

```
1  static void example21_btok(t_example21* x, t_symbol * file_name){
2      x->file_name = file_name->s_name;
3      example21_bang(x);
4  }
5
6  // Metodo para definir o nome do arquivo com retorno para a GUI
7  void example21_set_file_name(t_example21 *x, char * file_name){
8      sys_vgui(".x%x.c.s%xtx delete 0 end \n", x->canvas, x);
9      sys_vgui(".x%x.c.s%xtx insert end %s\n", x->canvas, x ,
           file_name);
10     x->file_name = file_name;
11     // DO SOMETHING
12 }
13
14 void example21_alfa(t_example21 *x, t_floatarg f){
15     if( f >= 2) f = 2;
16     if( f <= 0.5) f = 0.5;
17     sys_vgui(".x%x.c.s%xsb1 set %f\n",x->canvas,x,f);
18     x->alfa = f;
19 }
20
21 void example21_beta(t_example21 *x, t_floatarg f){
22     if( f >= 2) f = 2;
23     if( f <= 0.5) f = 0.5;
24     sys_vgui(".x%x.c.s%xsb2 set %f\n",x->canvas,x,f);
25     x->beta = f;
26 }
```

O resultado é visto a seguir.

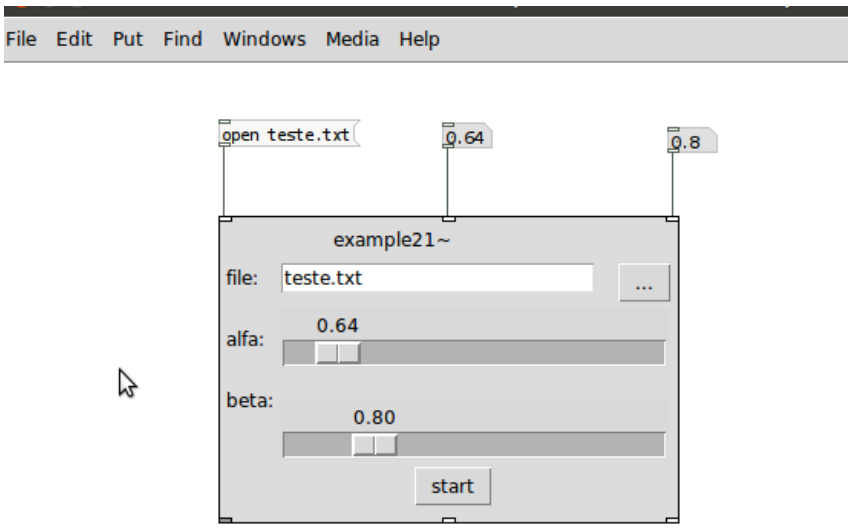

Figura 8.3: External com GUI e comandos

Capítulo 9

Orientação a Objetos

Parte do texto foi retirada de http://www.katjaas.nl/pitchshift/soundtouch .html.

Este cara é um exemplo de external que utiliza uma biblioteca OO: http://pure-data.svn.sourceforge.net/viewvc/pure-data/trunk/externals/fftw/ No caso, a bibliteca utilizada é a FFTW3. Outro exemplo é o PDCUDA do Drebs!

Outro exemplo que utilizad OO em external é o pixopencv.

http://pure-data.svn.sourceforge.net/viewvc/pure-data/trunk/externals/pix_opencv/

É possível criar externals utilizando C++ e orientação a objetos. A parte crítica de tal implementação é garantir que o PD consiga enxergar as funções dentro do objeto compilado do C++. É necessário pensar nisto não apenas como uma alternativa para criar externals em C++ mas também para utilizar bibliotecas C++ para a criação de externals. Utilizar estas bibliotecas torna necessário que a mesma seja compilada na forma como símbolos em C. Para garantir isto é necessário utilizar a definição extern "C".

TERMINAR ESTE EXEMPLO. COMO FAZER PARA O MAKEFILE FUNCIONAR TAMBÉM PARA ARQUIVOS .H e .CPP?

```
1 extern "C" void example18_setup(void) {
2     example18_class = class_new(gensym("example18"),
3             (t_newmethod) example18_new, // Constructor
4             0,
5             sizeof (t_example18),
6       CLASS_NOINLET,
7             0);
8 };
```

Usando o setup "externalizado"o mesmo passa a ser exportado e compilado em um objeto com esta função "visível". Deve ser o suficiente para compilar o external com sua API utilizando C++. Para compilar, usados um compilador C++ como o g++ do Linux.

```
1  LINUXCFLAGS = -msse -DPD -DUNIX -DICECAST -O3 -funroll-loops -
       fomit-frame-pointer -fcheck-new \
2      -Wall -W -Wshadow \
3      -Wno-unused -Wno-parentheses -Wno-switch -fvisibility=hidden
4
5  LINUXINCLUDE =  -I ./include
6
7    g++ $(LINUXCFLAGS) $(LINUXINCLUDE) -c *.cpp
8    g++ --export-dynamic -shared -o $*.pd_linux *.o -lc -lm -lstdc
       ++
9    strip --strip-unneeded $(NAME).pd_linux
10   rm -f *.o ../$(NAME).pd_linux
```

Capítulo 10

Miscelâneas

Aqui temos uma lista de funções definidas pela biblioteca que permite criarmos externals (m_pd.h). Não se trata de todas as funções mas de algumas que podem ser útil termos descrito como material de referência.

Ideal mesmo era rechear esta documentação com exemplos.

10.1 Gerenciamento de memória

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

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

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

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

Desaloca um ponteiro da memória.

```
void *resizebytes(void *x, size_t oldsize, size_t newsize);
```

10.2 Atoms

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

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

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

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

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

```
t_int atom_getintarg(int which, int argc, t_atom *argv);
```

```
t_symbol *atom_getsymbolarg(int which, int argc, t_atom *argv);
```

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

10.3 Binbufs

```
t_binbuf *binbuf_new(void);
```

```
void binbuf_free(t_binbuf *x);
```

```
t_binbuf *binbuf_duplicate(t_binbuf *y);
```

```
void binbuf_text(t_binbuf *x, char *text, size_t size);
```

```
void binbuf_gettext(t_binbuf *x, char **bufp, int *lengthp);
```

```
void binbuf_clear(t_binbuf *x);
```

```
void binbuf_add(t_binbuf *x, int argc, t_atom *argv);
```

```
void binbuf_addv(t_binbuf *x, char *fmt, ...);
```

```
void binbuf_addbinbuf(t_binbuf *x, t_binbuf *y);
```

```
void binbuf_addsemi(t_binbuf *x);
```

```
void binbuf_restore(t_binbuf *x, int argc, t_atom *argv);
```

```
void binbuf_print(t_binbuf *x);
```

```
int binbuf_getnatom(t_binbuf *x);
```

```
t_atom *binbuf_getvec(t_binbuf *x);
```

```
void binbuf_eval(t_binbuf *x, t_pd *target, int argc, t_atom *argv);
```

```
int binbuf_read(t_binbuf *b, char *filename, char *dirname, int crflag);
```

```
int binbuf_read_via_canvas(t_binbuf *b, char *filename, t_canvas *canvas, int crflag);
```

```
int binbuf_read_via_path(t_binbuf *b, char *filename, char *dirname, int crflag);
```

```
int binbuf_write(t_binbuf *x, char *filename, char *dir, int crflag);
```

```
void binbuf_evalfile(t_symbol *name, t_symbol *dir);
```

```
t_symbol *binbuf_realizedollsym(t_symbol *s, int ac, t_atom *av, int tonew);
```

10.4 Clocks

```
t_clock *clock_new(void *owner, t_method fn);
```

```
void clock_set(t_clock *x, double systime);
```

```
void clock_delay(t_clock *x, double delaytime);
```

```
void clock_unset(t_clock *x);
```

```
double clock_getlogicaltime(void);
```

```
double clock_getsystime(void);
```
OBSOLETE; use clock_getlogicaltime()

```
double clock_gettimesince(double prevsystime);
```

```
double clock_getsystimeafter(double delaytime);
```

```
void clock_free(t_clock *x);
```

10.5 Pure data

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

```
void pd_free(t_pd *x);
```

```
void pd_bind(t_pd *x, t_symbol *s);
```

```
void pd_unbind(t_pd *x, t_symbol *s);
```

```
t_pd *pd_findbyclass(t_symbol *s, t_class *c);
```

Verifica se ha um objeto desta classe instanciado no patch atual.

```
void pd_pushsym(t_pd *x);
```

```
void pd_popsym(t_pd *x);
```

```
t_symbol *pd_getfilename(void);
```

```
t_symbol *pd_getdirname(void);
```

```
void pd_bang(t_pd *x);
```

```
void pd_pointer(t_pd *x, t_gpointer *gp);
```

```
void pd_float(t_pd *x, t_float f);
```

```
void pd_symbol(t_pd *x, t_symbol *s);
```

```
void pd_list(t_pd *x, t_symbol *s, int argc, t_atom *argv);
```

```
void pd_anything(t_pd *x, t_symbol *s, int argc, t_atom *argv);
```

10.6 Pointers

```
void gpointer_init(t_gpointer *gp);
```

```
void gpointer_copy(const t_gpointer *gpfrom, t_gpointer *gpto);
```

```
void gpointer_unset(t_gpointer *gp);
```

```
int gpointer_check(const t_gpointer *gp, int headok);
```

10.7 Inlets and outlets

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

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

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

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

```
t_inlet *signalinlet_new(t_object *owner, t_float f);
```

```
void inlet_free(t_inlet *x);
```

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

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

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

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

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

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

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

```
t_symbol *outlet_getsymbol(t_outlet *x);
```

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

```
t_object *pd_checkobject(t_pd *x);
```

10.8 Canvases

```
void glob_setfilename(void *dummy, t_symbol *name, t_symbol *dir);
```

```
void canvas_setargs(int argc, t_atom *argv);
```

```
void canvas_getargs(int *argcp, t_atom **argvp);
```

```
t_symbol *canvas_getcurrentdir(void);
```

```
t_glist *canvas_getcurrent(void);
```

```
void canvas_makefilename(t_glist *c, char *file, char *result,int resultsize);
```

```
t_symbol *canvas_getdir(t_glist *x);
```

```
char sys_font[];
    /* default typeface set in s_main.c */
```

```
char sys_fontweight[];
    /* default font weight set in s_main.c */
```

```
int sys_fontwidth(int fontsize);
```

```
int sys_fontheight(int fontsize);
```

```
void canvas_dataproperties(t_glist *x, t_scalar *sc, t_binbuf *b);
```

```
int canvas_open(t_canvas *x, const char *name, const char *ext, char *dirresult, char **nameresult,
unsigned int size, int bin);
```

10.9 Classes

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

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

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

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

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

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

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

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

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

```
void class_sethelpsymbol(t_class *c, t_symbol *s);
```
Define o arquivo de help para a classe.

```
void class_setwidget(t_class *c, t_widgetbehavior *w);
```
Define qual será o comportamento da GUI deste objeto. Quando esta função é chamada o Pd não se responsabiliza mais quanto ao desenho do external.

```
void class_setparentwidget(t_class *c, t_parentwidgetbehavior *w);
```

```
t_parentwidgetbehavior *class_parentwidget(t_class *c);
```

```
char *class_getname(t_class *c);
```

```
char *class_gethelpname(t_class *c);
```

```
void class_setdrawcommand(t_class *c);
```

```
int class_isdrawcommand(t_class *c);
```

```
void class_domainsignalin(t_class *c, int onset);
```

```
void class_set_extern_dir(t_symbol *s);
```

```
void class_setsavefn(t_class *c, t_savefn f);
```

```
t_savefn class_getsavefn(t_class *c);
```

```
void class_setpropertiesfn(t_class *c, t_propertiesfn f);
```

```
t_propertiesfn class_getpropertiesfn(t_class *c);
```

10.10 Printing

```
void post(const char *fmt, ...);
```
Envia um mensagem para a saída padrão do PD. Funciona de maneira similar ao printf e aceita argumentos como %d ou %f. Ao contrário do printf, quebra linha no final.

```
void startpost(const char *fmt, ...);
```

```
void poststring(const char *s);
```

```
void postfloat(t_floatarg f);
```

```
void postatom(int argc, t_atom *argv);
```

```
void endpost(void);
```

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

```
void verbose(int level, const char *fmt, ...);
```

```
void bug(const char *fmt, ...);
```

```
void pd_error(void *object, const char *fmt, ...);
```

```
void sys_logerror(const char *object, const char *s);
```

```
void sys_unixerror(const char *object);
```

```
void sys_ouch(void);
```

10.11 System interface routines

```
int sys_isreadablefile(const char *name);
```

```
int sys_isabsolutepath(const char *dir);
```

```
void sys_bashfilename(const char *from, char *to);
```

```
void sys_unbashfilename(const char *from, char *to);
```

```
int open_via_path(const char *name, const char *ext, const char *dir, char *dirresult, char
**nameresult, unsigned int size, int bin);
```

```
int sched_geteventno(void);
```

```
double sys_getrealtime(void);
```

```
int (*sys_idlehook)(void);
```
Hook to add idle time computation

10.12 Threading

```
void sys_lock(void);
```

```
void sys_unlock(void);
```

```
int sys_trylock(void);
```

10.13 Signals

```
t_int *plus_perform(t_int *args);
```

```
t_int *zero_perform(t_int *args);
```

```
t_int *copy_perform(t_int *args);
```

```
void dsp_add_plus(t_sample *in1, t_sample *in2, t_sample *out, int n);
```

```
void dsp_add_copy(t_sample *in, t_sample *out, int n);
```

```
void dsp_add_scalarcopy(t_float *in, t_sample *out, int n);
```

```
void dsp_add_zero(t_sample *out, int n);
```

```
int sys_getblksize(void);
```
Retorna o tamanho do bloco de processamento do Pure Data.

```
t_float sys_getsr(void);
```
Retorna qual a amostragem (Sample Rate) atual do Pure Data.

```
int sys_get_inchannels(void);
```
Retorna a quantidade de canais de entrada do Pure Data.

```
int sys_get_outchannels(void);
```
Retorna a quantidade de canais de saída do Pure Data.

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

```
void dsp_addv(t_perfroutine f, int n, t_int *vec);
```

```
void pd_fft(t_float *buf, int npoints, int inverse);
```

```
int ilog2(int n);
```

```
void mayer_fht(t_sample *fz, int n);
```

```
void mayer_fft(int n, t_sample *real, t_sample *imag);
```

```
void mayer_ifft(int n, t_sample *real, t_sample *imag);
```

```
void mayer_realfft(int n, t_sample *real);
```

```
void mayer_realifft(int n, t_sample *real);
```

```
float *cos_table;
```

```
int canvas_suspend_dsp(void);
```

```
void canvas_resume_dsp(int oldstate);
```

```
void canvas_update_dsp(void);
```

```
int canvas_dspstate;
```

```
typedef struct _resample {
int method; // up/downsampling method ID
t_int downsample; // downsampling factor
t_int upsample; // upsampling factor
t_sample *s_vec; // here we hold the resampled data
int s_n;
t_sample *coeffs; // coefficients for filtering...
int coefsize;
t_sample *buffer; // buffer for filtering
int bufsize;
} t_resample;
```

```
void resample_init(t_resample *x);
```

```
void resample_free(t_resample *x);
```

```
void resample_dsp(t_resample *x, t_sample *in, int insize, t_sample *out, int outsize, int
method);
```

```
void resamplefrom_dsp(t_resample *x, t_sample *in, int insize, int outsize, int method);
```

```
void resampleto_dsp(t_resample *x, t_sample *out, int insize, int outsize, int method);
```

10.14 Utility functions for signals

```
t_float mtof(t_float);
```
Converte valores MIDI em frequencia.

```
t_float ftom(t_float);
```
Converte frequencias em valores MIDI.

```
t_float rmstodb(t_float);
```
Converte amplitudes RMS em decibéis.

```
t_float powtodb(t_float);
```
Converte potência em decibél.

```
t_float dbtorms(t_float);
```
Converte decibél para RMS.

```
t_float dbtopow(t_float);
```
Converte decibél para potência.

```
t_float q8_sqrt(t_float);
```

```
t_float q8_rsqrt(t_float);
```

10.15 Data

```
t_class *garray_class;
```

```
int garray_getfloatarray(t_garray *x, int *size, t_float **vec);
```

```
int garray_getfloatwords(t_garray *x, int *size, t_word **vec);
```

```
t_float garray_get(t_garray *x, t_symbol *s, t_int indx);
```

```
void garray_redraw(t_garray *x);
```

```
int garray_npoints(t_garray *x);
```

```
char *garray_vec(t_garray *x);
```

```
void garray_resize(t_garray *x, t_floatarg f);
```

```
void garray_usedindsp(t_garray *x);
```

```
void garray_setsaveit(t_garray *x, int saveit);
```

```
t_class *scalar_class;
```

```
t_float *value_get(t_symbol *s);
```

```
void value_release(t_symbol *s);
```

```
int value_getfloat(t_symbol *s, t_float *f);
```

```
int value_setfloat(t_symbol *s, t_float f);
```

10.16 GUI interface - functions to send strings to TK

```
void sys_vgui(char *fmt, ...);
```
Envia comandos Tk para o canvas com argumentos.

```
void sys_gui(char *s);
```
Envia comandos Tk para o canvas.

```
void sys_pretendguibytes(int n);
```

```
void sys_queuegui(void *client, t_glist *glist, t_guicallbackfn f);
```

```
void sys_unqueuegui(void *client);
```

```
void gfxstub_new(t_pd *owner, void *key, const char *cmd);
```

```
void gfxstub_deleteforkey(void *key);
```

```
t_class *glob_pdobject;
```
object to send ”pd”messages