Max/MSP Externals Tutorial

Version 2.5 (September 2003)* by Ichiro Fujinaga

This document is a tutorial for writing external objects for Max 3.5.x and Max 4 + MSP 2. It assumes that the reader is familiar with the Max environment and the C Programming Language. This document is intended to expand upon the material presented by David Zicarelli in his *Writing External Objects for Max* (1996) and is based on a tutorial started by Dale Stammen while at McGill University. Several examples are provided to demonstrate this process. Max externals (external objects) can be created using Think C, Symantec C++, and CodeWarrier.

This version (2.5) describes the development of Max/MSP externals (PowerPC only) using CodeWarrior on PowerPC. If you are using Think C or Symantec C++ compilers or on 68k machines please refer to the version 1 of this document.

Developing Max External Object with CodeWarrior□

This tutorial will explain how to create PowerPC native Max externals using MetroWerks Codewarrior (Pro 2 or later) Refer also to Zicareli (1996, 12–8). A very simple external object called **bang** will be created. Refer to Figure 1. for the source code of bang.c. The explanation of how **bang** works and writing Max external objects is provided later. First, some preparations are in order.

Creating a PPC Max External Object (shared library) with CodeWarrior Pro

1. If you are using IDE 4.1 or later and making object for Max 4 or later, follow the instruction in Zicarelli's *Writing External Objects for Max 4.0 and MSP 2.0* tutorial
2. Make sure both Max 3.5.x with Software Developers Kit and CodeWarrior are properly installed. In Max4/MSP2 download the Max4/MSP2 Software Development Kit from `http://www.cycling74.com/products/dlmaxmsp.html`.
3. Launch CodeWarrior IDE [2.1 or later].
4. From File menu select New… [IDE 4.0 or later] or New Project.
5. [IDE 4.1] In the New dialog box Enter the Project name: bang.mcp. Set the Location:. Click on MacOS C Stationary. Hit OK. In the New Project dialog box, select Standard Console:PPC:Std C Console PPC. Hit OK.

   [IDE 4.0] In the New dialog box Enter the Project name: bang.mcp. Set the Location:. Click on MacOS C/C++ Stationary. Hit OK. In the New Project dialog box, select Standard Console:Std C Console PPC. Hit OK.

   [IDE 3.3 or earlier] In the dialog box (Select project stationary), use the MacOS:C_C++: Standard Console:Std C Console PPC. Name the new project, e.g., bang.mcp.
6. A Project window named bang.mcp should open. Remove Sources folder in the project window by clicking on the name, then select Clear from the Edit menu. Do the same for ANSI Libraries. From Mac Libraries remove: MSL RuntimePPC.lib and MathLib.

* Thanks to Craig Sapp for editing and correcting errors.

7. From File menu select New Text File or New [IDE 3.3 or earlier]. Enter the C source code into this window, then save it in the bang folder as bang.c or copy a source file (bang.c) into the folder. Then drag-and-drop the file (bang.c) into the Project window.
8. From the Project menu, select Add Files to add MaxLib, which can be found in Max::Max/MSP SDK::Max includes, to the project. Close the Project Messages (Access Path, in IDE 2.1) box. If you are writing MSP externals, you will also need to add Max Audio Library.
9. From the Edit menu, select PPC Std C Console Settings. From Target Settings Panel Select PPC Target. Change the Project Type from Application to Shared Library. Change the File Name to name of the object, e.g. bang. Creator to max2, and File Type to???? [for Max 3.5 or earlier] or to iLaF [for Max 4 or later].
10. From the Target Settings Panels subwindows on the left, under Linker, select PPC Linker. Uncheck Generate SYM File. Under Entry Points in Main type main.
11. From the Target Settings Panels, under Linker, select PPC PEF. Check Expand Uninitialized Data.
12. [For IDE 3.0 and later] From the Target Settings Panels, select C/C++ Language. Uncheck ANSI Strict. In Prefix File: enter MacHeaders.h.
13. Close the PPC Std C Console Settings window and save it.
14. To compile the code, select Make from the Project menu. If successful you should see the familiar Max external object file created in the project folder (bang).
15. Run Max on a PPC Mac or double-click the newly created object. Create a new patcher and create a new box. Type the object name into the box and external object will be created. The location of the object should be specified in the File Preferences... in the Options menu of Max.

```
// bang.c -- A very simple Max external object.
//      Outputs a bang when a bang is received.

#include "ext.h"      // Required for all Max external objects

void *this_class;     // Required. Global pointing to this class

typedef struct _bang  // Data structure for this object
{
  t_object b_ob;      // Must always be the first field; used by Max
  void *b_out;        // Pointer to outlet, need one for each outlet
} t_bang;

// Prototypes for methods: need a method for each incoming message
void *bang_new(void);          // object creation method
void bang_bang(t_bang *bang);  // method for bang message

void main(void)
{
  // set up our class: create a class definition
  setup((t_messlist**) &this_class,(method)bang_new,0L,(short)sizeof(t_bang),0L, 0);

  addbang((method)bang_bang); // bind method "bang_bang" to the "bang" message}

void *bang_new(void)
{
  t_bang *bang;

  // create the new instance and return a pointer to it
  bang = (t_bang *)newobject(this_class);

  bang->b_out = bangout(bang); // create a bang outlet

  return(bang);               // must return a pointer to the new instance
}

void bang_bang(t_bang *bang)
{
  outlet_bang(bang->b_out);    // send a bang to the outlet bang->b_out
}
```

Figure 1. Source code for bang.c

Writing Max External Objects

To create an external Max object, you write a shared library. When you type the name of your object into an empty box in a Max patcher window, its shared library file is opened and its contents loaded into memory. The object is then created and able to receive messages from the Max environment. How your object will respond to the various messages is determined by the code you have written.

Your code for an external Max object will consist of a main function and functions (methods) that respond to specific Max messages sent to your object by the Max environment.

The sturcture of a minimal external object can be divided into four sections:

- intialization
- main()
- definition of the method to create a new object
- definition of methods that bind to other messages

The intializations consists of the necessary #include files, object structure definition, global variable declarations, and function prototypes. The main function, which is called only once when the user types the name of your object into a box in a Max patcher window for the first time, will define your objects class via `setup()` function and binds methods that will be used for incoming messges. The only requisite method for any class definition is the method that creates new objects. Within this method, memory for the new object is allocated and inlets and outlets are defined. Finally, methods that respond to other messages and other functions are defined. An explantion of each of these four sections is given below using a very simple object called **bang,** which simply outputs a bang upon a bang input. (See Figure 1 for the complete source code.)

The **bang** object: Initialization

The following lines are required for all objects:

```
#include "ext.h"   // Required for all Max external objects
void *this_class;  // Required. Global pointing to this class
```

The next step is to define a data structure for the **bang** Max object. This structure *must* start with a field called a `t_object`. The `t_object` contains references to the **bang** object's class definition as well as some other information. It is used by Max to communicate with the **bang** object. The following is the data structure for the **bang** object:

```
typedef struct _bang  // Data structure for this object
{
  t_object b_ob;      // Must always be the first field; used by Max
  void *b_out;        // Pointer to an outlet
} t_bang;
```

It is a Max convention to start the names of each field in the data structure with a lower case letter followed by an underscore (e.g. `b_out`).

After the object's data structure has been declared, the class methods that will respond to Max messages need to be declared. Your object will do its work by responding to messages from the Max environment. Objects commonly receive integer and float messages in their inlets. Your object's methods will process these numbers in some way and then send out messages using the object's outlets.

Your code must include methods (functions) that can respond to each message your Max object will receive. The **bang** object will receive a "new" message when someone types its name into a box in a Max patcher window. Therefore it is necessary to provide a method that will handle this message and create a new instance of the **bang** object. The **bang** object is also expected to send out a "bang" message on the outlet, upon a receipt of a "bang" in the left inlet. Methods will have to be written to handle this message. The declaration (prototype) of these methods is shown below.

```
// Prototypes for methods: need a method for each incoming message
void *bang_new(void);      // object creation method
void bang_bang(t_bang *d); // method for bang message
```

The **bang** object: main()

When Max creates your object for the first time, Max will load your external object into memory and create the first instance of your class. At this time, Max will call your external's main function once and only once. The main function specifies how your object should be initialized. The main function needs to do the following:

1. Set up your class: allocate memory for the object and specify methods for the creation of instances of your object.
2. Define messages that the object can respond to and bind each message to a method.

Here is the `main()` function of the **bang** object:

```
void main(void)
{
  // set up our class: create a class definition
  setup((t_messlist **) &this_class, (method)bang_new, 0L, (short)sizeof(t_bang), 0L, 0);

  addbang((method)bang_bang); // bind method "bang_bang" to the "bang" message
}
```

The setup function creates a definition of the **bang** object class, which will be used by the `bang_new` method to create new instances of the **bang** object. In the above call to the setup function for the **bang** object, `this_class` is the global variable declared at the beginning of the code. The second argument, `bang_new`, is a pointer to the instance creation method `bang_new`. This is the method that will be called when the object receives a "new" message from the Max environment. Since the **bang** object does not require any special memory cleanup when it is removed from the Max environment, 0L is used in place of a pointer to a `bang_free` method. The memory occupied by the **bang** object and all of its inlets and outlets will be removed automatically by Max.

The next argument to setup allocates memory for the class. In this example, `sizeof(t_bang)` is used to determine the number of bytes of memory needed. Since we are not creating a user interface object, the next argument to `menufun` will be 0L. The final 0 indicates that there is no argument to this object.

As mentioned above, the code must provide a method for each message you want to respond to. In the main function, each method should respond to the message with the functions: `addint, addinx, addbang, addmess, addft,` or `addftx`. Since the **bang** object only responds to the "bang" message, only one method, `bang_bang`, is needed. In order to bind the `bang_bang` method, which will output a "bang", to a "bang" input message, we use the routine `addbang(bang_bang)`.

The **bang** object: The object creation function

When a user creates a new instance of your object by typing the name bang into a box in a Max patcher window, opening a file with your object already in it, or by cutting and pasting your object, your object will receive a "new" message. This is a request to your creation method to create an object that is an instance of your class. The creation function then handles any arguments that were typed in the box in the Max patcher window, initializes data fields, and creates the object's inlets and outlets. Finally, the creation function returns a pointer to the new instance of the object. These actions are shown in the method `bang_new` listed below.

```
void *bang_new(void)
{
  t_bang *bang;

  // create the new instance and return a pointer to it
  bang = (t_bang *)newobject(this_class);

  bang->b_out = bangout(bang);// create a bang outlet

  return(bang);               // must return a pointer to the new instance
}
```

The function, `newobject`, is used to create a new instance of the class **bang.** The argument, `this_class`, is the global variable that points to this class. This pointer was set by the setup function in the main function.

When your object is created, Max automatically creates one inlet, but other inlets and outlets must be explicitly defined. Using the `bangout` function, an outlet (that only outputs "bang" messages) will be created and returns a pointer, which will be stored in the object's data field `b_out`.

Finally, `bang,` the pointer to the new instance of our object that was returned by the call to `newobject`, must be returned from the function `bang_new`.

Now we have a new instance of our object represented as a "bang" box in a Max patcher window. It is now waiting to receive "bang" messages that will cause its method to do the specified operation, namely, ouput a "bang". We will now examine how this is done.

The **bang** object: Handling the "bang" message

```
void bang_bang(t_bang *bang)
{
  outlet_bang(bang->b_out);// send a bang to the outlet bang->b_out
}
```

When a "bang" message arrives at the object's left inlet, the `bang_bang` function (method) is called. This happens, because in the `main()` the "bang" message, was bound to this function `bang_bang()` by the function:

```
addbang((method)bang_bang);
```

The `bang_bang` method simply sends a "bang" messages via the outlet. The method calls the Max function `outlet_bang` to cause the "bang" to be output. In the object creation function, `bang_new` (see above), an outlet was created for this object with the statement:

```
bang->b_out = bangout(bang);
```

This function returned a pointer to the object's outlet which we stored in the struct field `bang->b_out`.

The **diff** object: Inlets and arguments

A simple object **diff** will be used to introduce how to add inlets and arguments to your object. This object basically functions as the Max built-in "**-**" object. It outputs the difference of two integers: the number coming in on the left inlet minus the number stored in the object which can be either specified via the right inlet or in the argument inside the object's box. The source code is shown in Figure 2.

```
/*diff.c --  97/03/24 IF (based on Dale Stammen's diff)
**           98/01/14 for PowerPC only
**This external defines an object similar to the standard "-" Max object.
**The diff object has 2 inlets and 1 outlet.  Left inlet accepts bang and integers,
**right inlet accepts integers, and outlet outputs the difference of the 2 inputs.
*/

#include "ext.h"      // Required for all Max external objects
void *this_class;     // Required. Global pointing to this class

typedef struct _diff  // Data structure for this object
{
  t_object d_ob;      // Must always be the first field; used by Max
  long d_valleft;     // Last value from left outlet
  long d_valright;    // Last value from right outlet
  long d_valtotal;    // Value to be sent to outlet
  void *d_out;        // Pointer to outlet, need one for each outlet
} t_diff;

// Prototypes for methods: you need a method for each message you want to respond to
void *diff_new(long value);                // Object creation method
void diff_int(t_diff *diff, long value);  // Method for message "int" in left inlet
void diff_in1(t_diff *diff, long value);  // Method for message "int" in right inlet
void diff_bang(t_diff *diff);              // Method for bang message


void main(void)       // main receives a copy of the Max function macros table
{
  // set up our class: create a class definition
  setup((t_messlist **) &this_class, (method)diff_new, 0L, (short)sizeof(t_diff),
        0L, A_DEFLONG, 0);

  addbang((method)diff_bang); // bind method "diff_bang" to the "bang" message

  addint((method)diff_int);   // bind method "diff_int" to int's received in the left inlet

  addinx((method)diff_in1,1); // bind method "diff_in1" to int's received in the right inlet
}


/************************************************************************************
diff_new(long value)

inputs:      value -- the integer from the typed argument in the object box
description: creates a new instance of our class diff.  Called once when the external
               object is loaded.
returns:     pointer to new instance
************************************************************************************/


void *diff_new(long value)
{
  t_diff *diff;

  diff = (t_diff *)newobject(this_class); // Create new instance and return a pointer to it

  diff->d_valright = value;// Initialize the difference value
  diff->d_valleft = 0;
  diff->d_valtotal = value;

  diff->d_out = intout(diff); // Create our outlet

  intin(diff, 1);             // Create the right inlet

  return(diff);               // Must return a pointer to the new instance
```

```
}
/************************************************************************
diff_int(t_diff *a, long value)

inputs:      diff  - pointer to t_diff object
             value - value received in the inlet
description: substracts the right value with the incoming value. Stores the new left inlet
                  value as well as the total and outputs the total.
returns:     nothing
************************************************************************/

void diff_int(t_diff *diff, long value)
{
  diff->d_valleft = value; // Store the value received in the left inlet

  diff->d_valtotal = diff->d_valleft - diff->d_valright;   // Subtracts the right inlet
                                                           // value from the left

  diff_bang(diff);      // Call bang method right away since it's the left inlet
}

/************************************************************************
diff_in1(t_diff *diff, long value)

inputs:      diff  - pointer to our object
             value - value received in the inlet
description: stores the new right value, calculates and stores the new difference between
                the left and right value
returns:     nothing
************************************************************************/

void diff_in1(t_diff *diff,long value)
{
  diff->d_valright = value;         // Store the value

  diff->d_valtotal = diff->d_valleft - value;   // Update new difference
}


/************************************************************************
diff_bang(t_diff *a)

inputs:      diff  - pointer to our object
description: method called when bang is received: it outputs the current difference
                of the left and right values
returns:     nothing
************************************************************************/

void diff_bang(t_diff *diff)
{
  outlet_int(diff->d_out, diff->d_valtotal);// put out the current total
}
```

Figure 2. Source code for the diff object

The **diff** object: Initialization

The data structure for the diff object is shown below. Note that three values are stored within the object.

```
typedef struct _diff  // Data structure for this object
{
  t_object d_ob;      // Must always be the first field; used by Max
  long d_valleft;  // Last value sent to left outlet
  long d_valright; // Last value sent to right outlet
  long d_valtotal; // Value to be sent to outlet
  void *d_out;     // Pointer to outlet, need one for each outlet
} t_diff;
```

In the `setup` function in `main()` now has `A_DEFLONG` argument indicating that the object accept one integer argument in the object box.

```
setup((t_messlist **) &this_class, diff_new, 0L, (short)sizeof(t_diff), 0L, A_DEFLONG, 0);
```

Three methods are bound with the three types of messages: "bang" in the left inlet, integer entered in the left inlet, and integer entered in the right inlet.

```
addbang((method)diff_bang); // bind "diff_bang" to the "bang" message
addint((method)diff_int);  // bind "diff_int" to int received in the left inlet
addinx((method)diff_in1,1); // bind "diff_in1" to int received in the right inlet
```

The **diff** object: The object creation function

Unlike the **bang** object above, the `diff_new` function is passed an integer argument from the object box that the user may type. The object's variables are initialized, an outlet that output integer is created, and the right inlet, which accepts integers is also created:

```
void *diff_new(long value)
{
  t_diff *diff;

  diff = (t_diff *)newobject(this_class); // Create new instance and return a pointer to it

  diff->d_valright = value;// Initialize the diff values
  diff->d_valleft = 0;
  diff->d_valtotal = value;

  diff->d_out = intout(diff); // Create our outlet

  intin(diff,1);              // Create the right inlet

  return(diff);            // Must return a pointer to the new instance
}
```

The **diff** object: Methods

The `diff_int` method is called when an integer comes in on the left inlet. It stores the value in `d_valleft`, subtracts that value with `d_valright`, storing the result in `d_valtotal`, then calls the `diff_bang` method to output the result.

```
void *diff_int(t_diff *diff, long value)
{
  diff->d_valleft = value;    // Store the value received in the left inlet
  diff->d_valtotal = diff->d_valleft + diff->d_valright;   // Subtract right inlet value
                                                           // from the left
  diff_bang(diff);            // Call bang method right away since it's the left inlet
}
```

The `diff_in1` method is called when an integer comes in on the right inlet. It stores the new value in `d_valright` then updates the `val_total`.

```
void *diff_in1(t_diff *diff, long value)
{
  diff->d_valright = value;               // Store the value
  diff->d_valtotal = diff->d_valleft + value;    // Update new total
}
```

The `diff_bang` method is called when a "bang" comes in the left inlet or, indirectly via `diff_int` method, when an integer comes in the left inlet.

```
void *diff_bang(t_diff *diff)
{
  outlet_int(diff->d_out, diff->d_valtotal);// simply put out the current total
}
```

The **diff_assist** object: Adding to the Max's New Object list and assistance messages

Two enhancements will be added to the **diff** object: the object (**diff_assist**) will be included in the Patcher's New Object list and the assistance messages, which appears when the mouse is pointed at object's inlets and outlets. The complete listing of **diff_assist** object is in Figure 3.

To make an entry in the New Object list is very simple. All you need to do is to include the following function in your `main()`:

```
finder_addclass("All Objects", "diff_assist"); // add class to the New object list
```

If you want to add the object to the "Arithmetic/ Logic" list, you could add the following:

```
finder_addclass("Arithmetic/Logic", "diff_assist");
```

In order to add the assistance messages: a method must be defined, which must be bound to the Max message "assist", and since we will be using a resource for the string for the assistance messages, we need to copy the string from the resource. The binding and the copying is done in the `main()` as follows:

```
addmess((method)diff_assist, "assist", A_CANT, 0);    // bind method diff_assist to the
                                                      // assistance message
rescopy('STR#', ResourceID);  // copy the assistance messages resource into Max's
                              // temp file
```

ResourceID is a number that you define when creating the string resource. The `rescopy` function copies the string to Max's temporary file (Max Temp 1 in the Temporary Items folder). How to create this resource is explained next. The explanation of the `diff_assist` method will follow.

Creating a String Resource in ResEdit 2.1 for a Max External Object

1. Launch ResEdit.
2. Click on the clown to get rid of it.
3. Select New... from the File menu; move to your project folder.
4. Name your resource file with EXACTLY the same name as your project and append the name with .rsrc. For example, if your project is called projectname.μ, name your resource projectname.μ.rsrc. On recent IDEs the name is not crucial since you will add the file to your project window.
5. Click the New button
6. Select Create New Resource from the Resource menu.
7. Scroll down to the resource type STR# in the Select New Type window. Make sure you select STR# and not STR. Click on OK.
8. ResEdit will now create a window called STR# ID = 128. Click on the field 1) *****. Select Insert New Field(s) from the Resource menu. In the box after The string, type in your external Max object's assistance string for the first inlet. You may use a maximum of 60 characters. Repeat step 8 for as many inlets and outlets as your Max object will need. Create them in order, with the first string being the message for inlet 1, the second for inlet 2.
9. Select Get Resource Info from the Resource menu. Enter your resource ID number in the field ID:. This number MUST match the resource ID number you define in your Max object. If you wish, you may type in the name of your resource in the field Name. This will help you remember what the resource is used for in the "resource picker window".
10. Save your resource. Make sure it is saved to your project folder and that it has the same name as your project file with .rsrc added to the end of the name.
11. Add the resource file to the MetroWorks project window.

The **diff_assist** object: `diff_assist` method

```
void *diff_assist(t_diff *diff, Object *b, long msg, long arg, char *s)
{
  // copy the appropriate message to the destination string
  assist_string(ResourceID, msg, arg, 1, 3, s);
}
```

In the argument list for **diff_assist,** `diff` is a pointer to our object, `b` is a pointer to the object's box in the Max patcher window. `msg` will be one of two values: 1 if the cursor is over an inlet or 2 if it is over an outlet. `arg` is the inlet or outlet number starting at 0 for the left inlet. s is where you will copy a C string containing your assistance information.

The function `assist_string` handles the posting of the assistance string in the assistance area of the Max patcher window. It will copy the correct string from the resource in the memory specified by ResourceID. (ResourceID was defined at the beginning of the code.) This resource was copied into the Max's temp file by `rescopy()` in the `main` function. `msg` specifies if either an inlet or outlet was selected and `arg` is the inlet or outlets number. The argument 1 specifies that the first string in the resource corresponds to the first inlet. Likewise, the argument 3 specifies that the third string in the resource goes with the first outlet. The function `assist_string` will copy the selected resource string into `s`, which will then be displayed in the assistance area of the patcher window.

```
/*diff_assist.c --    97/03/24 IF (based on Dale Stammen's diff)
**                  98/01/14 for PowerPC only IF
**This external object defines an object similar to the standard "-" max object.
**The diff object has 2 inlets and 1 outlet.  Left inlet accepts bang and integers,
**right inlet accepts integers, and outlet outputs the difference of the 2 inputs.
*/

#include "ext.h"        // Required for all Max external objects

void *this_class;       // Required. Global pointing to this class

#defineResourceID 3999  // resource ID# for assistance strings created in ResEdit

typedef struct _diff  // Data structure for this object
{
  t_obj d_ob;      // Must always be the first field; used by Max
  long d_valleft;  // Last value sent to left outlet
  long d_valright; // Last value sent to right outlet
  long d_valtotal; // Value to be sent to outlet
  void *d_out;     // Pointer to outlet, need one for each outlet
} t_diff;

// Prototypes for methods: you need a method for each message you want to respond to
void *diff_new(long value);         // Object creation method
void diff_int(t_diff *diff, long value);  // Method for message "int" in left inlet
void diff_in1(t_diff *diff, long value);  // Method for message "int" in right inlet
void diff_bang(t_diff *diff);             // Method for bang message
void diff_assist(t_diff *diff, Object *b, long msg, long arg, char *s); // Assistance method

void main(void)
{
  // set up our class: create a class definition
  setup((t_messlist **) &this_class, (method)diff_new, 0L, (short) sizeof(t_diff), 0L,
        A_DEFLONG, 0);
  addbang((method)diff_bang); // bind method "diff_bang" to the "bang" message
  addint((method)diff_int);   // bind method "diff_int" to int's received in the
                              // left inlet
  addinx((method)diff_in1,1); // bind method "diff_in1" to int's received in the
                              // right inlet
  addmess((method)diff_assist, "assist",A_CANT,0);// bind method "diff_assist" to
                                                   // the assistance message
  rescopy('STR#', ResourceID);// copy the assistance messages resource into Max's
                              // temp file
  finder_addclass("All Objects", "diff_assist");  // add class to the New object list
}


/************************************************************************
diff_new(long value)

inputs:   value -- the integer from the typed argument in the object box
description: creates a new instance of our class diff.
       Called once when the external object is loaded.
returns:     pointer to new instance
************************************************************************/

void *diff_new(long value)
{
  t_diff *diff;

  diff = (t_diff *)newobject(this_class); // Create the new instance
  diff->d_valright = value;// Initialize the values
  diff->d_valleft = 0;
  diff->d_valtotal = value;
  diff->d_out = intout(diff); // Create our outlet
  intin(diff,1);              // Create the right inlet
  return(diff);            // Must return a pointer to the new instance
}
```

```
/************************************************************************************
diff_int(t_diff *a, long value)

inputs:      diff  - pointer to t_diff object
             value - value received in the inlet
description: subtracts the right value with the incoming value. Stores the new left inlet
             value as well as the difference and outputs the difference.
returns:     nothing
************************************************************************************/

void diff_int(t_diff *diff, long value)
{
  diff->d_valleft = value;   // Store the value received in the left inlet
  diff->d_valtotal = diff->d_valleft - diff->d_valright; //Subtracts right from the left
  diff_bang(diff);      // Call bang method right away since it's the left inlet
}

/************************************************************************************
diff_in1(t_diff *diff, long value)

inputs:diff  -- pointer to our object
       value -- value received in the inlet
description: stores the new right value, calculates and stores the
             new difference between the left and right value
returns:     nothing
************************************************************************************/

void diff_in1(t_diff *diff, long value)
{
  diff->d_valright = value;              // Store the value
  diff->d_valtotal = diff->d_valleft - value;  // Update new difference
}


/************************************************************************************
diff_bang(t_diff *a)

inputs:      diff -- pointer to our object
description: method called when bang is received: it outputs the current
             sum of the left and right values
returns:     nothing
************************************************************************************/

void diff_bang(t_diff *diff)
{
  outlet_int(diff->d_out, diff->d_valtotal); // simply put out the current total
}


/************************************************************************************
void diff_assist(a, b, msg, arg, s)

inputs:      diff  - pointer to t_diff object
             b     - pointer to the t_diff object's box
             msg   - specifies whether request for inlet or outlet info
             arg   - selected inlet or outlet number
             s     - destination for assistance string
description: method called when assist message is received: it outputs the correct
             assistance message string to the patcher window
returns:     nothing
************************************************************************************/

#if 1
void diff_assist(t_diff *diff, Object *b, long msg, long arg, char *s)
{
  // copy the appropriate message to the destination string
  assist_string(ResourceID, msg, arg, 1, 3, s);
}

#else
#define ASSIST_INLET (1)
#define ASSIST_OUTLET (2)
#include "string.h"


// Note: on CW 7.0 MSL_All_PPC.Lib
```

```
void diff_assist(t_diff *diff, Object *b, long msg, long arg, char *s)
// copy the appropriate message to the destination string
{
  if (msg == ASSIST_INLET)
  {
    switch (arg)
    {
       case 0: strcpy(s, "Left Operand");
             break;
       case 1: strcpy(s, "Right Operand");
    }
  }
  else if (msg == ASSIST_OUTLET)
    strcpy(s, "Result");
}
#endif
}
```

Figure 3 Source code for diff_assist object

The **absolut** object: Float, Atom, and list

Thus far, the only data type we have been using is an integer type, namely long. In this section, we'll introduce the float data type, the Atom data type, and the list, which is an array of Atoms.

The float data type is similar to long except that it involves floating-point numbers. Max provides macros and functions to handle floats very similar to longs, e.g., to add left inlets you would use:

```
addint(long_method);
```

for inlet that accepts long and use:

```
addfloat(float_method);
```

for inlet that accepts float.

An Atom is a special data type (a structure) that allows any of the four data types (long, float, Symbol, Object) used in Max to be stored. Here is how it is defined:

```
union word      /* union for packing any data type */
{
  long     w_long;
  float    w_float;
  t_symbol *w_sym;
  t_object *w_obj;
};

typedef struct atom   // and an atom which is a typed datum
{
  short      a_type;  // from the defs below
  union word a_w;
} t_atom;
```

The struct member a_type specifies what type of data is stored in a_w, and it could be any of the following:

```
#define A_NOTHING  0  // ends the type list
#define A_LONG     1  // Type-checked integer argument
#define A_FLOAT    2  // Type-checked float argument
#define A_SYM      3  // Type-checked symbol argument
#define A_OBJ      4  // for argtype lists; passes the value of sym (obsolete)
#define A_DEFLONG  5  // long but defaults to zero
#define A_DEFFLOAT 6  // float, defaults to zero
#define A_DEFSYM   7  // symbol, defaults to ""
#define A_GIMME    8  // request that args be passed as an array
                      // the routine will check the types itself
#define A_CANT     9  // cannot typecheck args
```

One common place where Atom is used is when an argument of an object could be of different data type, e.g. long or float. In the setup() function arguments are passed as an array of Atoms (which is the list in Max). The A_GIMME is specified in the type argument of the setup() function:

```
setup((t_messlist **) &this_class, (method)abs_new, 0L, (short)sizeof(t_abs), 0L,
         A_GIMME, 0);
```

When the instance creation function is called (in this case, abs_new), the number of arguments (ac) entered and the array of Atoms (*av) are passed:

```
void *abs_new(Symbol *s, short ac, t_atom *av)
```

```
/*absolut.c -- 02/10/01 IF
**
** Accepts either integers and floats depending on the argument
** provides assist messages without Resource
*/

#include "ext.h"        // Required for all Max external objects
void *this_class;       // Required. Global pointing to this class

typedef struct _abs   // Data structure for this object
{
  t_object a_ob;     // Must always be the first field; used by Max
  short a_mode; // Integer or float mode
  void *a_out;     // Pointer to outlet, need one for each outlet
} t_abs;

// Prototypes for methods: you need a method for each message you want to respond to
void *abs_new(Symbol *s, short ac, Atom *av);           // Object creation method
void abs_int(t_abs *abs, long value);  // Method for message "int" in left inlet
void abs_flt(t_abs *abs, float value); // Method for message "float" in left inlet


void main(void)
{
  // set up our class: create a class definition
  setup((t_messlist **) &this_class, (method)abs_new, 0L, (short)sizeof(t_abs), 0L,
        A_GIMME, 0);
  addint((method)abs_int); // bind method "abs_int" to int received in the left inlet
  addfloat((method)abs_flt);  // bind method "abs_flt" to float received in the left inlet
}


void *abs_new(Symbol *s, short ac, t_atom *av)
{
  t_abs *abs = (t_abs *)newobject(this_class); // Create the new instance
  abs->a_mode = A_LONG; // Initialize the values

post ("Type: %d", av[0].a_type);
  if (ac && av[0].a_type == A_FLOAT)
  {
     abs->a_out = floatout(abs); // Create float outlet
     abs->a_mode = A_FLOAT;
  }
  else
    abs->a_out = intout(abs); // Create int outlet
  return(abs);              // Must return a pointer to the new instance
}


void abs_int(t_abs *abs, long value)
{
  if (abs->a_mode == A_LONG)
    outlet_int(abs->a_out, (value >= 0) ? value: -value);
  else
    outlet_float(abs->a_out, (float)((value >= 0) ? value: -value));
}


void abs_flt(t_abs *abs, float value)
{
  if (abs->a_mode == A_LONG)
    outlet_int(abs->a_out, (long)((value >= 0) ? value: -value));
  else
    outlet_float(abs->a_out, (value >= 0) ? value: -value);

}
```

The **minium** object: List

A list in Max is simply an array of Atoms. A list will be used if you declare a method that responds to the "list" message and receive its arguments with A_GIMME:

```
addmess((method)minimum_list, "list", A_GIMME, A_NOTHING);
```

Then your method, `minimum_list` in the example above, will be passed a list. This is done by `argc` (`short`) and `argv (t_atom *)`. `argc` is the number of Atoms and `argv` points to the first Atom in the array. Here is an example:

```
void minimum_list(Minimum *x, Symbol *s, short argc, t_atom *argv)
```

The Symbol *s contains the message itself (in this case, "list"). The object **minimum** illustrates use of these data types (see Figure 4).

Notice that we use the struct notation argv[i].a_type to access the a_type field. It is also possible to use the pointer argv to access the field, i.e., `(argv + i)->`a_type. You may choose whatever style suits you best.

In the above example, if the Atom contains a long (i.e., a_type == A_LONG), we want to store the argument into our internal Atom list, a_list as a long. Likewise, if (a_type == A_FLOAT) we would store it as a float, and if (a_type == A_SYM) we would store the argument as a symbol. Max provides several macros for storing an item into an atom. These are:

```
SETLONG(Atom *a, long number);
SETFLOAT(Atom *a, float number);
SETSYM(Atom *a, Symbol *s);
```

Here are the current macro definitions as they appear in Max #include file "ext_mess.h".

```
#define SETSYM(ap, x)    ((ap)->a_type = A_SYM, (ap)->a_w.w_sym = (x))
#define SETLONG(ap, x)   ((ap)->a_type = A_LONG, (ap)->a_w.w_long = (x))
#define SETFLOAT(ap, x)  ((ap)->a_type = A_FLOAT, (ap)->a_w.w_float = (x))
```

These macros accomplish two things. First the macro sets the a_type field of the Atom to the correct type. This means that SETLONG will set the a_type field of the Atom to A_LONG, SETFLOAT sets it to A_FLOAT, and SETSYM sets it to A_SYM. The macro then puts the long, float, or the pointer to the symbol into the union a_w. Remember that a pointer to the symbol is stored in the union, and not the actual symbol.

In the above example we used the following line of code to call SETLONG:

```
SETLONG(a->a_list + i, argv[i].a_w.w_long);
```

In this call, a is a pointer to our Object. We use it to access the array of Atoms called a_list that is in our object's data structure. Since SETLONG requires a pointer to an Atom, we must give it a pointer to the *i* th Atom in the array. When i == 0, a->a_list + i is a pointer to the first Atom in the array a_list. Likewise, if i == 5, a->a_list + i is a pointer to the 6th Atom in the array.

Notice how we access the long field of the union a_w in the argv Atom list. We write argv[i] to access the *i* th Atom in the argv list. argv[i].a_w accesses the union a_w field of the struct atom. Finally, argv[i].a_w.w_long accesses the long value

stored in the union a_w. We first access the atom, then the union, and finally the data.

Another way of putting a long value into an Atom is:

```
a->a_list[i].a_type = A_LONG;
a->a_list[i].a_w.w_long = 100;
```

Using this method you are responsible for setting the a_type field yourself.

You can use SETFLOAT the same way as SETLONG. SETFLOAT will set the a_type field to A_FLOAT, and place the float value in the float field of the union a_w (i.e., a_w.w_float). To access a float field of an Atom in the argv list in the above example, we write:

```
argv[i].a_w.w_float or (argv + i)->a_w.w_float
```

Likewise, to access this value in our internal array of Atoms we write:

```
a->a_list[i].a_w.w_float or (a->a_list + i)->a_w.w_float
```

```
/* minimum.c -- output the minimum of a group of numbers ------- */
// The type of output (long or float) is determined by the first number in the argument
// If no argument, it defaults to long
// From the Max 3.5 distribution. Slightly modified by IF 97/04/02
// For PowerPC only 98/01/14 IF
// Minor cleanup 02/09/29 IF
// Topics covered: floats, Atoms, lists


#include "ext.h"

#define MAXSIZE 32
#define ResourceID 3008

typedef struct minimum
{
  t_object m_ob;
  Atom m_args[MAXSIZE];
  long m_count;
  short m_incount;
  short m_outtype;
  void *m_out;
} Minimum;

void *class;
void DoAtomMin(Atom *min, Atom *new);
void minimum_bang(Minimum *x);
void minimum_int(Minimum *x, long n);
void minimum_in1(Minimum *x, long n);
void minimum_float(Minimum *x, double f);
void minimum_ft1(Minimum *x, double f);
void minimum_list(Minimum *x, Symbol *s, short ac, Atom *av);
void minimum_assist(Minimum *x, void *b, long m, long a, char *s);
void *minimum_new(Symbol *s, short ac, Atom *av);

void main(void)
{
  setup((t_messlist **)&class, (method)minimum_new,0L, (short)sizeof(Minimum),
       0L, A_GIMME, 0);
  addbang((method)minimum_bang);
  addint((method)minimum_int);
  addinx((method)minimum_in1, 1);
  addfloat((method)minimum_float);
  addftx((method)minimum_ft1, 1);
  addmess((method)minimum_list, "list", A_GIMME, 0);
  addmess((method)minimum_assist, "assist", A_CANT, 0);
  finder_addclass("Arith/Logic/Bitwise", "minimum");
  rescopy('STR#', ResourceID);
}
```

```
void DoAtomMin(Atom *min, Atom *new)   // check to see if new minimum,
                      //depending on the data types
{
  if (min->a_type==A_NOTHING) // At startup set minimum
  {
    *min = *new;
    return;
  }
  if (min->a_type==A_FLOAT) // old is FLOAT
  {
    if (new->a_type==A_FLOAT) // new is FLOAT
    {
       if (new->a_w.w_float < min->a_w.w_float)
          min->a_w.w_float = new->a_w.w_float;
    }
    else //new is LONG, old is FLOAT
    {
       if ((float)new->a_w.w_long < min->a_w.w_float)
          min->a_w.w_float = (float)new->a_w.w_long;
    }
  }
  else // old is LONG
  {
    if (new->a_type==A_LONG) // new is LONG
    {
       if (new->a_w.w_long < min->a_w.w_long)
          min->a_w.w_long = new->a_w.w_long;
    }
    else // new is float, old is LONG
    {
       if ((long)new->a_w.w_float < min->a_w.w_long)
          min->a_w.w_long = (long)new->a_w.w_float;
    }
  }
}

void minimum_bang(Minimum *x)
{
  short i;
  Atom themin;
  long res;
  double fres;

  themin.a_type = A_NOTHING;
  for (i=0; i < x->m_count; i++) // check if any of the input is a new minimum
    DoAtomMin(&themin,x->m_args+i);
  if (x->m_outtype==A_LONG)
  {
    if (themin.a_type==A_LONG)
       res = themin.a_w.w_long;
    else
       res = (long)themin.a_w.w_float;
    outlet_int(x->m_out,res);
  }
  else
  {
    if (themin.a_type==A_FLOAT)
       fres = themin.a_w.w_float;
    else
       fres = (float)themin.a_w.w_long;
    outlet_float(x->m_out,fres);
  }
}

void minimum_int(Minimum *x, long n)
{
  SETLONG(x->m_args,n);
  minimum_bang(x);
}

void minimum_in1(Minimum *x, long n)
{
  SETLONG(x->m_args+1,n);
  x->m_count = 2;
}
```

```
void minimum_float(Minimum *x, double f)
{
  SETFLOAT(x->m_args,f);
  minimum_bang(x);
}

void minimum_ft1(Minimum *x, double f)
{
  SETFLOAT(x->m_args+1,f);
  x->m_count = 2;
}

void minimum_list(Minimum *x, Symbol *s, short ac, Atom *av)
{
  short i;

  if (ac >= MAXSIZE)
    ac = MAXSIZE - 1;
  for (i = 0; i < ac; i++, av++)
  {
    if (av->a_type == A_LONG)
       SETLONG(x->m_args + i, av->a_w.w_long);
    else if (av->a_type == A_FLOAT)
       SETFLOAT(x->m_args + i, av->a_w.w_float);
  }
  x->m_count = ac;
  minimum_bang(x);
}

void minimum_assist(Minimum *x, void *b, long m, long a, char *s)
{
  assist_string(ResourceID, m, a, 1, 3, s);
}

void *minimum_new(Symbol *s, short ac, t_atom *av)
{
  Minimum *x;

  x = (Minimum *)newobject(class);
  x->m_count = 2;
  if (ac)
  {
    x->m_args[1] = *av;    // initialize with the first argument
    if (av->a_type==A_LONG)
    {
       x->m_args[0].a_type = x->m_outtype = A_LONG;
       x->m_out = intout(x);
       x->m_args[0].a_w.w_long = 0;
       intin(x, 1);
    }
    else if (av->a_type==A_FLOAT)
    {
       x->m_args[0].a_type = x->m_outtype = A_FLOAT;
       x->m_out = floatout(x);
       x->m_args[0].a_w.w_float = 0;
       floatin(x, 1);
    }
  }
  else // if no argument, set to a default
  {
    x->m_outtype = A_LONG;
    intin(x,1);
    x->m_out = intout(x);
    SETLONG(x->m_args + 1, 0L);
    SETLONG(x->m_args, 0L);
  }
  return (x);
}
```

Figure 4. Source code for the minimum object

More Atoms and list

Max uses Atoms when passing messages between objects. If your object is going to be able to send a list out of its outlet, it will have to use a list of Atoms. Likewise, if you wish to receive lists, or more than 7 typed data in arguments from your object's box in the Max patcher, you will again have to deal with Atoms. Remember, Atoms are simply a struct that have a field of type union that allows them to contain different types of data.

It is now necessary to examine the structure of a message in Max. Consider the following message box:

This message box contains 5 items, the symbol "play", the long integers 100 and 200, the float 2.5, and finally the symbol "stop". If this message is sent to your object, your object will actually receive the message "play", followed by a list of 4 atoms containing 100, 200, 2.5 and "stop". In other words, "play" is the message and the remaining items are its arguments. One way to make your object understand this message is to use addmess() in its main function.

```
addmess(max_play, "play", A_LONG, A_LONG, A_FLOAT, A_SYM, 0); // bind method max-play to the
"play" message"
```

or with optional arguments, so that if some of the arguments are not specified by the user, the object will set them to a default values:

```
addmess(max_play, "play", A_DEFLONG, A_DEFLONG, A_DEFFLOAT, A_DEFSYM, 0);
```

But this approach requires that you always have two longs, a float and a symbol in the right order. You are also limited to a total of seven arguments using this declaration method.

There is another way for your object to receive messages and their arguments. When you declare a method to receive its arguments with A_GIMME, the arguments will be passed to your object in an argc, argv list. More about this argc, argv stuff later.

In order to tell Max to give you all of the arguments in a message, you bind your method to the message in your main function with the Max function addmess. For example, to bind the method atoms_play with the above message you would write in your main function:

```
addmess(atoms_play, "play", A_GIMME, 0);  // bind method "atoms_play" to the "play" message"
```

This call binds the method atoms_play to the message "play". Whenever the object receives the message "play", Max will call the method atoms_play and pass it the message and a list of arguments.

A_GIMME tells Max to pass the message and its arguments without typechecking them. You are now responsible for typechecking them yourself.

You now need to write a method that will be able to receive this message and its arguments. The method atoms_play would be declared as:

```
void *atoms_play(Example *a, Symbol *mess, int argc, Atom *argv)
```

In this function declaration, a is a pointer to your object, mess is pointer to the message that called this method (in this example the, "play" message). The integer

argc is the number of arguments contained in the atom list and argv is a pointer to an array of atoms containing the actual arguments. Up to 65,536 arguments can be received by a method.

If your object receives the message “play 100 200 2.5 stop”, Max will call your play function. Your atoms_play function will receive a pointer to the symbol “play” in mess, the integer 4 in argc, and finally a pointer to a list of atoms containing the values 100 200 2.5 “stop”. The code in Figure 5 shows you how to typecheck and access the data in the atom list.

```
#define MAX_ARGS  20

typedef struct example        // data structure for this object
{
  Object a_ob;
  Atom   a_list[MAX_ARGS];  // array of Atoms: list
  int    a_size;          // number of Atoms in the list
} Example;

void *atoms_play(Example *a,int argc, Atom *argv)
{
  int i;

  a->a_size = argc;
  if (a->a_size > MAX_ARGS)
    a->a_size = MAX_ARGS;

  for(i = 0; i < a->a_size; i++)
    switch(argv[i].a_type)    // type check each argument
    {
       case A_LONG:
          SETLONG(a->a_list + i, argv[i].a_w.w_long);
          post(“argument %ld is a long: %ld”, (long)i, argv[i].a_w.w_long);
          break;
       case A_FLOAT:
          SETFLOAT(a->a_list + i, argv[i].a_w.w_float);
          post(“argument %ld is a float: %f”, (long)i, argv[i].a_w.w_float);
          break;
       case A_SYM:
          SETSYM(a->a_list + i, argv[i].a_w.w_sym);
          post(“argument %ld is a symbol: %s”,(long)i, argv[i].a_w.w_sym->s_name);
          break;
    }
}
```

Figure 5. Type checking an argc, argv list of atoms

This example receives a list of arguments from Max whenever the object receives the “play” message. It then checks the type of each Atom in the argv list and stores it into an internal array of Atoms. Finally, it reports to the Max window the type and value of the argument.

When working an Atom, you must be able to correctly access its various fields. In Figure 5, we examine the a_type field of an Atom to determine the type of data contained in the union. As mentioned above a_type will be either A_LONG, A_FLOAT, or A_SYM. These constants are declared in the Max #include file “ext_mess.h”.

When you want to store a symbol into an Atom, or access a symbol already in an Atom, you must remember that a pointer to the symbol is stored in the Symbol field of the union a_w. The field in the union a_w is defined as Symbol *w_sym. Therefore, in order to store a symbol into an Atom you store the pointer to the symbol and not the symbol itself. Likewise, when you access a symbol, you need to access what the pointer in the Symbol field points to. In other words, to get at a symbol, there is yet another stage of indirection.

In the above example, we use SETSYM to set the pointer to the symbol contained in the argv list into our internal Atom list a_list. Therefore, SETSYM wants a pointer to the symbol as its second argument.

```
SETSYM(a->a_list + i, argv[i].a_w.w_sym);
```

Notice how we post the actual symbol to the Max window. We use the following post function:

```
post("argument %ld is a symbol: %s", (long) i, argv[i].a_w.w_sym->s_name);
```

Note that in order to access our actual symbol, we must access what the symbol pointer points to:

```
argv[i].a_w.w_sym->s_name
```

In the Max #include file "ext_mess.h" a symbol is defined as the following struct:

```
struct symbol
{
  char *s_name;          /* name */
  struct object *s_thing;  /* possible binding to an object */
} Symbol;
```

Therefore, in order to access a symbol in an Atom, first access the Atom, then the union a_w, then the w_sym field and finally the s_name field of the Symbol, i.e., argv[i].a_w.w_sym->s_name.

Now that you have a list of Atoms in your object you can send it to an outlet. To do this you need to create a list outlet using the Max function:

```
Outlet *listout (void *owner)
```

In our example we would create the list outlet in the object's creation function example_new.

```
a->a_list_outlet = listout(Example *x);
```

To send the internal list a_list out this outlet, one would use the Max function:

```
void *outlet_list(Outlet *x, Symbol *msg, int argc, Atom *argv);
```

We would call this function with the following arguments:

```
outlet_list(a->a_list_outlet, "list", a->a_size, &(a->a_list));
```

where a->a_list_outlet is a pointer to the outlet we created with listout, "list" is the message to be sent, a->a_size is the number of Atoms in the internal Atom list, and &(a->a_list) is a pointer to the first Atom in this list.

The **atoms** object:

Notice the addmess functions:

```
addmess(atoms_list,"play",A_GIMME,0);  // bind method "atoms_list" to "play" message
addmess(atoms_list,"list",A_GIMME,0);  // bind method "atoms_list" to "list" message
```

Both of these lines of code cause the function `atoms_list` to be called when the object receives either the "play" message of the "list" message. Also notice that we requested that Max send to our object the arguments of the message as a list of atoms. This was accomplished by using A_GIMME.

The `list` Method

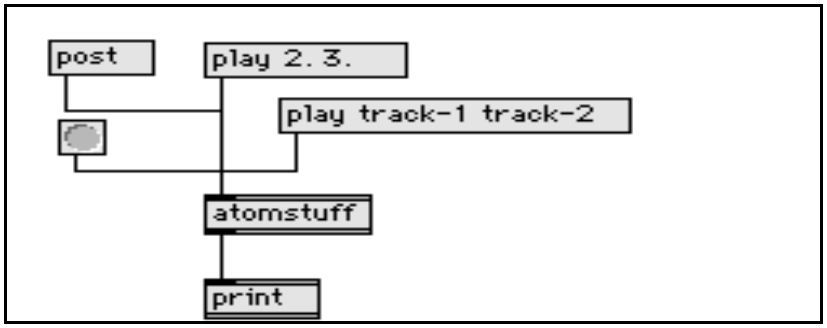
This method receives a list of atoms from Max contain the items of the list sent to your object. The number of items (or atoms) in the list is in argc. The actual atoms are stored in argv. Actually these are pointers to atoms. We then check each atom for its type before we put it in our list. The list method then sends the list of atoms out its outlet using `outlet_list`. Notice we use `&(a->a_list)` to point to our list of Atoms.

The `bang` Method

When our object receives the bang message, it simply sends the current contents of its list out its outlet using the `outlet_list` function.

The `post` Method (Zicarelli 1996, 71)

When the object receives the "bang" message, it posts the contents of its Atom list to the Max window using the function `postatom` (Zicarelli 1996, 72). Here is how to post a list of atoms:



The **mymetro** object: Clock routines

This example uses the **Clock** object (Zicarelli 1996, 60–3), which allows scheduling in Max. The routines associated with the **Clock** objects allows events to happen in the future. This is accomplished by assigning a function to be executed when the clock goes off and indicate when the clock is to go off. More specifically:

1. Use `clock_new()` to create a **Clock** object and assign the function to be executed when it goes off.
2. Use `clock_delay()` to schedule the execution of the clock function in the future relative to current time. Use clock_fdelay() for floating point version. Use `clock_unset()` to stop the **Clock.**
3. When the **Clock** is no longer needed, it should be removed with `freeobject` function.

```
/*  Defines the object "mymetro" which is similar to the standard
  "metro" Max object. The metro object has 2 inlets and 2 outlets.

  "bang"  in left inlet starts metronome
  "stop"  in left inlet stops metronome
  integer in right inlet sets tempo in ms

  left output sends bangs at each metronome interval
  right outlet outputs current time

  The object also posts messages to the Max window indicating the current state of
  mymetro.
*/

#include "ext.h"      // Required for all Max external objects

#defineResourceID     3999  /* resource ID# for assistance strings */
#define DEFAULT_TEMPO    1000
#define MIN_TEMPO       40

typedef struct _metro       /* data structure for this object */
{
  t_object   m_ob;     /* must always be the first field; used by Max */
  void *m_clock;       /* pointer to clock object */
  long  m_interval;       /* tempo in milliseconds */
  void *m_bang_outlet; /* pointers to bang outlet */
  void *m_time_outlet; /* pointers to time outlet */
} t_metro;

void *metro_new(long value);
void metro_in1(t_metro *m, long value);
void metro_bang(t_metro *m);
void metro_assist(t_metro *m, t_object *b, long msg, long arg, char *s);
void metro_free(t_metro *m);
void metro_stop(t_metro *m);
void clock_function(t_metro *m);

void *class;    // Required. Global pointing to this class

void main(void)
{
  /* set up our class: create a class definition */
  setup((t_messlist **) &class, (method)metro_new, (method)metro_free,
          (short)sizeof(t_metro), 0L, A_DEFLONG, 0);

  /* bind method "metro_bang" to the "bang" message */
  addbang((method)metro_bang);
```

```
  /* bind method "metro_in1" to int's received in the right inlet */
  addinx((method)metro_in1,1);

  /* bind method "metro_stop" to the "stop" message" */
  addmess((method)metro_stop,"stop",0);

  /* bind method "metro_assist" to the assistance message" */
  addmess((method)metro_assist,"assist",A_CANT,0);

  /* copy the assistance messages resource into the Max temp file */
  rescopy('STR#',ResourceID);

  /* add class to the New object list */
  finder_addclass("All Objects","mymetro");
}

/***********************************************************************
metro_new(long value)

inputs:   value - the integer from the typed in argument in the object box
description: creates a new instance of this class metro.
returns:     pointer to new instance
***********************************************************************/
void *metro_new(long value)
{
  t_metro *m;

  m = (t_metro *)newobject(class); // create the new instance and return a pointer to it

  if (value > MIN_TEMPO)      // initialize
  {
    m->m_interval = value;    // save tempo argument from box
    post("mymetro tempo set to %ld", value);
  }
  else
  {
    m->m_interval = DEFAULT_TEMPO;  // set to default tempo
    post("mymetro set to default tempo of %ld ms", DEFAULT_TEMPO);
  }
  m->m_clock = clock_new(m, (method)clock_function);  // create the metronome clock

  intin(m, 1);                    // create the right inlet
  m->m_time_outlet = intout(m);  // create right outlet for time
  m->m_bang_outlet = bangout(m); // create left outlet for ticks

  return(m);
}

/***********************************************************************
metro_in1(t_metro *m, long value)

inputs:m    -- pointer to our object
    value   -- value received in the inlet
description:   stores the new metronome tempo value
***********************************************************************/
void metro_in1(t_metro *m, long value)
{
  m->m_interval = value;   // store the new metronome interval
  post("metronome tempo changed to %ld", value);
}
```

```
/************************************************************************************
void metro_bang(t_metro *m)

inputs:   m -- pointer to our object
description: method called when bang is received: it starts the metronome
************************************************************************************/
void metro_bang(t_metro *m)
{
  long time;

  time = gettime();           // get current time
  clock_delay(m->m_clock, 0L);   // set clock to go off now
  post("clock started at %ld", time);
}

/************************************************************************************
void metro_stop(t_metro *m)

inputs:   m -- pointer to our object
description: method called when myMetro receives "stop" message. Stops the metronome
************************************************************************************/
void metro_stop(t_metro *m)
{
  long time;

  time = gettime();            // get current time
  clock_unset(m->m_clock); // remove the clock routine from the scheduler
  outlet_int(m->m_time_outlet, time);
  post("metronome stopped at %ld", time);
}

/************************************************************************************
void clock_function(t_metro *m)

inputs:   m -- pointer to our object
description: method called when clock goes off: it outputs a bang to be sent to the
     outlet and resets the clock  to go off after the next interval.
************************************************************************************/
void clock_function(t_metro *m)
{
  long time;

  time = gettime();     // get current time
  clock_delay(m->m_clock, m->m_interval); // schedule another metronome click

   outlet_bang(m->m_bang_outlet);         // send out a bang
  outlet_int(m->m_time_outlet, time);       // send current time to right outlet
  post("clock_function %ld", time);

}

/************************************************************************************
metro_free(t_metro *m)

inputs:   m -- pointer to our object
description: method called when t_metro objects is destroyed. It is used to free memory
       allocated to the clock.
************************************************************************************/
void metro_free(t_metro *m)
{
  clock_unset(m->m_clock); // remove the clock routine from the scheduler
  clock_free(m->m_clock);  // free the clock memory
}

void metro_assist(t_metro *m, Object *b, long msg, long arg, char *s)
{
  // copy the appropriate message to the destination string
  assist_string(ResourceID, msg, arg, 1L, 3L, s);
}
```

Figure 6. Source code for mymetro.c

Writing MSP External Objects

Creating a MSP external object is very similar to creating a Max externals. There are two additional methods you need to create: **perform method** and **dsp method.** To create a CodeWarrior project, follow the instruction for Max externals described above **Creating a PPC Max External Object (shared library) with CodeWarrior Pro.** Make sure to add the Max Audio Library in Step 8.

An explanation of how to setup an MSP external is given below using a very simple object called **thru~,** which simply outputs the input. (See Figure 7 for the complete source code.)

```
/* thru~.c
** one channel thru object (a minimal MSP object)
** 99/04/05 IF
** 00/02/21 IF mono only, see thru2~ for the stereo version
*/
#include "ext.h"     // Required for all Max external objects
#include "z_dsp.h"   // Required for all MSP external objects

void *thru_class;

typedef struct _thru  // Data structure for this object
{
    t_pxobject x_obj;
} t_thru;

void *thru_new(void);
t_int *thru_perform(t_int *w);
void thru_dsp(t_thru *x, t_signal **sp, short *count);

void main(void)
{
     setup((t_messlist **)&thru_class, (method)thru_new, (method)dsp_free,
            (short)sizeof(t_thru), 0L, 0);
     addmess((method)thru_dsp, "dsp", A_CANT, 0);
     dsp_initclass();
}

void *thru_new(void)
{
  t_thru *x = (t_thru *)newobject(thru_class);
  dsp_setup((t_pxobject *)x, 1);          // left inlet
  outlet_new((t_pxobject *)x, "signal");     // left outlet
  return (x);
}

void thru_dsp(t_thru *x, t_signal **sp, short *count)
{
  post("thru~ inlet: %d outlet: %d", sp[0]->s_vec, sp[1]->s_vec);
   post("thru~ size of buffer: %d", sp[0]->s_n);
  dsp_add(thru_perform, 3, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n);
}

t_int *thru_perform(t_int *w)
{
   t_float *inL = (t_float *)(w[1]);
   t_float *outL = (t_float *)(w[2]);
  int n = (int)(w[3]);

  while (n--)
   *outL++ = *inL++;
   return (w + 4); // always add one more than the 2nd argument in dsp_add()
}
```

Figure 7. Source code for thru~.c

The **thru~** object: Initialization

In addition to the required "ext.h," "z_dsp.h" must be included. z_dsp.h and other MSP-specific header files can be found in the **MSP #includes** folder.

```
#include "ext.h"   // Required for all Max external objects
#include "z_dsp.h" // Required for all MSP external objects
```

The data structure to define a data structure for the MSP object is different from Max objects. The first field in the `struct` is the `t_pxobject` instead of the `t_object`. The following is the data structure for the **thru~** object:

```
typedef struct _thru  // Data structure for this object
{
    t_pxobject x_obj;
} t_thru;
```

A minimum of three functions must be called in the `main()`. The first is the `setup()`, which is same as the call in the Max objects, except that `dsp_free` must be passed as parameter to free memory. The message binding function, `addmess()` is called to bind the system message "dsp" to your dsp method, which will add your perform method to the DSP chain. The message "dsp" is sent to your object whenever the user starts DAC or ADC. Finally, `dsp_initclass()` is called to transparently add other methods to your object. As in Max objects, if there are other messages to be bound, it should be done here.

```
void main(void)
{
    setup((t_messlist **)&thru_class, (method)thru_new, (method)dsp_free,
            (short)sizeof(t_thru), 0L, 0);
    addmess((method)thru_dsp, "dsp", A_CANT, 0);
    dsp_initclass();
}
```

The **thru~** object: New instance creation

In the instance creation function, `thru_new()`, a new object is created via the `newobject()` function. Inlet that accepts signal data is created by the `dsp_setup()` function. This function must be called even if your object does not have any signal inlet, in which case 0 should be passed as the second parameter. The signal outlets are created using the `outlet_new()` function. Use multiple call to `outlet_new()` to create additional outlets. Non-signal inlets, when there are no signal inlets, can be created using the standard functions (e.g. `intin, floatin, inlet_new`).

```
void *thru_new(void)
{
  t_thru *x = (t_thru *)newobject(thru_class);
  dsp_setup((t_pxobject *)x, 1);          // left inlet
  outlet_new((t_pxobject *)x, "signal");    // left outlet
  return (x);
}
```

The **thru~** object: The dsp method

The dsp method will be called via the "dsp" message from MSP when it is building the DSP call chain (when audio is turned on). Your task is to add your perform method to the chain using the `dsp_add()`. When MSP calls this function, it is passed the object, an array containing buffers to inlets and outlets defined, and another array that indicates the number of connections to the inlets and outlets. Note that you must add a perform method even if no inlets and outlets are connected.

```
void thru_dsp(t_thru *x, t_signal **sp, short *count)
{
    dsp_add(thru_perform, 3, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n);
}
```

The first parameter in `dsp_add()` is the name of your perform method, the second number indicates the number of arguments in the perform method, followed by the arguments, all of which must be the size of a pointer or a long and appears in your perform method as an array of `t_int`. The second parameter passed to the dsp method, `t_signal **sp`, is an array of pointers to `struct t_signal` (defined in **z_dsp.h**).

```
typedef struct _signal
{
    int s_n;                    // number of samples: Signal Vector Size
    t_sample *s_vec;            // pointer to inlet/outlet buffer
    float s_sr;                 // sample rate
    struct _signal *s_next;
    struct _signal *s_nextused;
    short s_refcount;
    short s_zero;
} t_signal;
```

The array is ordered left to right for inlets then left to right for outlets.

The **thru~** object: The perform method

The perform method is where the actual signal processing take place. The method is added to the DSP signal chain by the dsp method (see above) and will be repeatedly called as long as the audio is on.

```
t_int *thru_perform(t_int *w)
{
  t_float *inL = (t_float *)(w[1]);
  t_float *outL = (t_float *)(w[2]);
  int n = (int)(w[3]);

  while (n--)
    *outL++ = *inL++;
  return(w + 4); // always add one more than the 2nd argument in dsp_add()
}
```

The parameters passed to the perform method are typically pointers to buffers representing the inlets and outlets, and the size of the buffer, which is usually the Signal Vector Size (DSP Status).

The MSP objects: Summary of events and related functions

As with Max objects, when the user creates the object for the first time, the main() is called once. The *object*_new() function is called every time the object is created. The *object*_dsp() function is called every time the DAC/ADC is started or sometimes when connection to object is changed, which initiates the building of the DSP chain. Finally, the *object*_perform() function is called at every audio interrupt.

The **thru2~** object: Stereo version of **thru~**

Figure 8 is the source code for the thru2~ object, which is the stereo version of the thru~ object with two inlets and two outlets. Note that in the `thru2_new()`, a flag (x->x_obj.z_misc) is set so that inlet and outlet points to two different memory locations. This, in most cases, is not needed or even desired but used here as an illustration. Among the standard MSP object only fft~/ifft~ and tapout` set this flag. When this flag is not set, which is the default, the input and output buffer maybe the same.

```
/* thru2~.c  -- two channel thru object
** 99/04/05 IF
** 00/02/21 IF set x->x_obj.z_misc
*/
#include "ext.h"     // Required for all Max external objects
#include "z_dsp.h"   // Required for all MSP external objects

void *thru2_class;

typedef struct _thru2    // Data structure for this object
{
    t_pxobject x_obj;
} t_thru2;

void *thru2_new(void);
t_int *thru2_perform(t_int *w);
void thru2_dsp(t_thru2 *x, t_signal **sp, short *count);

void main(void)
{
     setup((t_messclass **)&thru2_class, (method)thru2_new, (method)dsp_free,
               (short)sizeof(t_thru2), 0L, 0);
     addmess((method)thru2_dsp, "dsp", A_CANT, 0);
     dsp_initclass();
}

void *thru2_new(void)
{
  t_thru2 *x = (t_thru2 *)newobject(thru2_class);
  dsp_setup((t_pxobject *)x, 2);          // two inlets
  outlet_new((t_pxobject *)x, "signal");     // left outlet
  outlet_new((t_pxobject *)x, "signal");     // right outlet
  x->x_obj.z_misc = Z_NO_INPLACE;
  return (x);
}

void thru2_dsp(t_thru2 *x, t_signal **sp, short *count)
{
  dsp_add(thru2_perform, 5, sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec,
          sp[3]->s_vec, sp[0]->s_n);
}

t_int *thru2_perform(t_int *w)
{
  t_float *inL = (t_float *)(w[1]);
  t_float *inR = (t_float *)(w[2]);
  t_float *outL = (t_float *)(w[3]);
  t_float *outR = (t_float *)(w[4]);
  int n = (int)(w[5]);
  while (n--)
  {
    *outL++ = *inL++;
    *outR++ = *inR++;
  }
    return (w + 6); // always add one more than the 2nd argument in dsp_add()
}
```

Figure 8. Source code for thru2~.c

The **thru2~** object: New instance creation

In the instance creation function, `thru2_new()`, a new object is created with two signal inlets by specifying 2 as the second argument of `dsp_setup()`. Two outlets are created by calling `outlet_new()` twice. As mentioned before the flag `x->x_obj.z_misc` is set (via `Z_NO_INPLACE`, which is defined in z_proxy.h) to guarantee that the input buffer and output buffer point to different memory locations.

```
void *thru2_new(void)
{
  t_thru2 *x = (t_thru2 *)newobject(thru2_class);
  dsp_setup((t_pxobject *)x, 2);          // two inlets
  outlet_new((t_pxobject *)x, "signal");  // left outlet
  outlet_new((t_pxobject *)x, "signal");  // right outlet
  x->x_obj.z_misc = Z_NO_INPLACE;
  return (x);
}
```

The **thru2~** object: The dsp method

Because there are two inlets and two outlets five arguments must be passed to the perform method. The ordering of `sp[]` array is inlets left to right then outlets left to right. So that `sp[0]` points to the left inlet, sp[1] to the right inlet, `sp[2]` to the right outlet, and `sp[3]` to the right outlet.

```
void thru2_dsp(t_thru2 *x, t_signal **sp, short *count)
{
  dsp_add(thru2_perform, 5, sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec,
          sp[3]->s_vec, sp[0]->s_n);
}
```

The **thru2~** object: The perform method

This perform method has two inputs and two outpus.

```
t_int *thru2_perform(t_int *w)
{
  t_float *inL = (t_float *)(w[1]);
  t_float *inR = (t_float *)(w[2]);
  t_float *outL = (t_float *)(w[3]);
  t_float *outR = (t_float *)(w[4]);
  int n = (int)(w[5]);
  while (n--)
  {
    *outL++ = *inL++;
    *outR++ = *inR++;
  }
    return (w + 6); // always add one more than the 2nd argument in dsp_add()
}
```

The **thru0~** object: No op version of **thru2~**

Depending on how you connect objects it may be that the input and output buffers are the same, so you don't actually have to do "anything" to have signals go through an object (see Figure 9). Do not assume that this will always be the case, however.

```
/* thru0~.c
** two channel thru object (absolute minimal MSP object)
** 99/04/05 IF
** 00/02/21 IF see thru2~ for the not-in-place version
*/
#include "ext.h"
#include "z_dsp.h"

void *thru_class;

typedef struct _thru
{
    t_pxobject x_obj;
} t_thru;

void *thru_new(double val);
t_int *thru_perform(t_int *w);
void thru_dsp(t_thru *x, t_signal **sp, short *count);

void main(void)
{
    setup(&thru_class, thru_new, (method)dsp_free, (short)sizeof(t_thru), 0L, 0);
    addmess((method)thru_dsp, "dsp", A_CANT, 0);
    dsp_initclass();
}

void *thru_new(double val)
{
  t_thru *x = (t_thru *)newobject(thru_class);
  dsp_setup((t_pxobject *)x, 2);          // two inlets
  outlet_new((t_pxobject *)x, "signal");     // right outlet
  outlet_new((t_pxobject *)x, "signal");     // left outlet
  return (x);
}

void thru_dsp(t_thru *x, t_signal **sp, short *count)
{
  dsp_add(thru_perform, 0);
}

t_int *thru_perform(t_int *w)
{
    return (w + 1);
}
```

Figure 9. Source code for thru0~.c

The **thrugain~** object: signal and non-signal inlets and outlets

The thrugain~ object demonstrates how to combine signal inlets and outlets with non-signal inlets and outlets. The non-signal (bang, int, float, messages) inlets and outlets must appear to the right of signal inlets and outlets. Therefore the order in which the inlets and outlets are created in the new function is strictly obeyed. The non-signal inlet creation methods (initin, floatin) must precede dsp_setup() and non-signal outlet creation methods (bangout, intout, floatout, etc.) must precede signal's outlet_new() methods.

```
/* thrugain~.c one signal inlet, one float inlet, one signal outlet, one float outlet
**
** 02/11/04 IF
*/
#include "ext.h"     // Required for all Max external objects
#include "z_dsp.h"   // Required for all MSP external objects

void *thrugain_class;

typedef struct _thrugain    // Data structure for this object
{
    t_pxobject t_obj;
    t_float t_gain;
    void *t_out;   // float outlet
} t_thrugain;

void *thrugain_new(void);
t_int *thrugain_perform(t_int *w);
void thrugain_dsp(t_thrugain *x, t_signal **sp, short *count);
void thrugain_float(t_thrugain *x, t_float val);

void main(void)
{
     setup((t_messlist **)&thrugain_class, (method)thrugain_new, (method)dsp_free,
            (short)sizeof(t_thrugain), 0L, 0);
     addmess((method)thrugain_dsp, "dsp", A_CANT, 0);
     addftx((method)thrugain_float, 9);    // 9 is the maximum, it has to match the number
     dsp_initclass();                       // in floatin()
}

void *thrugain_new(void)
{
  t_thrugain *x = (t_thrugain *)newobject(thrugain_class);
  x->t_gain = 1.0;
  floatin((t_pxobject *)x, 9);          // right float inlet
  dsp_setup((t_pxobject *)x, 1);        // left signal inlet, must come after floatin
  x->t_out = floatout((t_pxobject *)x); // right float outlet
  outlet_new((t_pxobject *)x, "signal");  // left signal outlet, must come after floatout
  return (x);
}

void thrugain_float(t_thrugain *x, t_float val)
{
  post("Float in: %f", val);
  x->t_gain = val;
  outlet_float(x->t_out, x->t_gain);
}

void thrugain_dsp(t_thrugain *x, t_signal **sp, short *count)
{
  dsp_add(thrugain_perform, 4, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n, x);
}
```

```
t_int *thrugain_perform(t_int *w)
{
    t_float *inL = (t_float *)(w[1]);
    t_float *outL = (t_float *)(w[2]);
  int n = (int)(w[3]);
  t_thrugain *x = (t_thrugain *)w[4];
  while (n--)
   *outL++ = x->t_gain * *inL++;

    return (w + 5); // always add one more than the 2nd argument in dsp_add()
}
```

Figure 10. Source code for thrugain~.c

The **thrugain1~** object: using left inlet for both signal and non-signal input

The thrugain1~ object demonstrates how to use the left inlet for both signal and non-signal input. The following example also accepts bangs.

```
/* thrugai1n~.c one signal/float inlet, one signal outlet, one float outlet
**
** 02/11/05 IF
*/
#include "ext.h"     // Required for all Max external objects
#include "z_dsp.h"   // Required for all MSP external objects

void *thrugain_class;

typedef struct _thrugain    // Data structure for this object
{
    t_pxobject t_obj;
    t_float t_gain;
    void *t_out;   // float outlet
} t_thrugain;

void *thrugain_new(void);
t_int *thrugain_perform(t_int *w);
void thrugain_dsp(t_thrugain *x, t_signal **sp, short *count);
void thrugain_float(t_thrugain *x, t_float val);
void thrugain_bang(t_thrugain *x);

void main(void)
{
    setup((t_messlist **)&thrugain_class, (method)thrugain_new, (method)dsp_free,
(short)sizeof(t_thrugain), 0L, 0);
    addmess((method)thrugain_dsp, "dsp", A_CANT, 0);
    addfloat((method)thrugain_float);
    addbang((method)thrugain_bang);
    dsp_initclass();
}

void *thrugain_new(void)
{
  t_thrugain *x = (t_thrugain *)newobject(thrugain_class);
  x->t_gain = 1.0;
  dsp_setup((t_pxobject *)x, 1);        // left signal inlet
  x->t_out = floatout((t_pxobject *)x); // right float outlet
  outlet_new((t_pxobject *)x, "signal");  // left signal outlet, must come after floatout
  return (x);
}
```

```
void thrugain_float(t_thrugain *x, t_float val)
{
  post("Float in: %f", val);
  x->t_gain = val;
  outlet_float(x->t_out, x->t_gain);
}

void thrugain_bang(t_thrugain *x)
{
  post("Bang");
  outlet_float(x->t_out, x->t_gain);
}

void thrugain_dsp(t_thrugain *x, t_signal **sp, short *count)
{
  dsp_add(thrugain_perform, 4, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n, x);
}


t_int *thrugain_perform(t_int *w)
{
     t_float *inL = (t_float *)(w[1]);
     t_float *outL = (t_float *)(w[2]);
  int n = (int)(w[3]);
  t_thrugain *x = (t_thrugain *)w[4];
  while (n--)
   *outL++ = x->t_gain * *inL++;

     return (w + 5); // always add one more than the 2nd argument in dsp_add()
}
```

Figure 11. Source code for thrugain1~.c

The **thrugain2~** object: inlets for both signal and non-signal input

The thrugain2~ object is a stereo thru object that accepts float (and bang) input on both inlets. It exploits the proxy nature of the signal inlets.

```
/* thrugai2n~.c two signal/float inlet, two signal outlets, one float outlet
**
** 02/11/05 IF
*/
#include "ext.h"     // Required for all Max external objects
#include "z_dsp.h"   // Required for all MSP external objects

void *thrugain_class;

typedef struct _thrugain    // Data structure for this object
{
     t_pxobject t_obj;
     t_float t_gain;
     void *t_out;   // float outlet
} t_thrugain;

void *thrugain_new(void);
t_int *thrugain_perform(t_int *w);
void thrugain_dsp(t_thrugain *x, t_signal **sp, short *count);
void thrugain_float(t_thrugain *x, t_float val);
void thrugain_bang(t_thrugain *x);
```

```
void main(void)
{
     setup((t_messlist **)&thrugain_class, (method)thrugain_new, (method)dsp_free,
(short)sizeof(t_thrugain), 0L, 0);
     addmess((method)thrugain_dsp, "dsp", A_CANT, 0);
     addfloat((method)thrugain_float);
     addbang((method)thrugain_bang);
     dsp_initclass();
}

void *thrugain_new(void)
{
  t_thrugain *x = (t_thrugain *)newobject(thrugain_class);
  x->t_gain = 1.0;
  dsp_setup((t_pxobject *)x, 2);        // left signal inlet
  x->t_out = floatout((t_pxobject *)x); // right float outlet
  outlet_new((t_pxobject *)x, "signal");  // middle signal outlet, must come after
floatout
  outlet_new((t_pxobject *)x, "signal");  // left signal outlet, must come after floatout
  return (x);
}

void thrugain_float(t_thrugain *x, t_float val)
{
  post("Float in: %f", val);
  x->t_gain = val;
  outlet_float(x->t_out, x->t_gain);
}

void thrugain_bang(t_thrugain *x)
{
  post("Bang");
  outlet_float(x->t_out, x->t_gain);
}

void thrugain_dsp(t_thrugain *x, t_signal **sp, short *count)
{
  dsp_add(thrugain_perform, 6, sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec, sp[3]->s_vec,
sp[0]->s_n, x);
}


t_int *thrugain_perform(t_int *w)
{
     t_float *inL = (t_float *)(w[1]);
     t_float *inR = (t_float *)(w[2]);
     t_float *outL = (t_float *)(w[3]);
     t_float *outR = (t_float *)(w[4]);
  int n = (int)(w[5]);
  t_thrugain *x = (t_thrugain *)w[6];

  while (n--)
  {
   *outL++ = x->t_gain * *inL++;
   *outR++ = x->t_gain * *inR++;
  }

     return (w + 7); // always add one more than the 2nd argument in dsp_add()
}
```

Figure 12. Source code for thrugain2~.c