\textit{\muRoar} Manual

Philipp "ph3-der-loewe" Schafft

April 26, 2010
Contents

1 Programming with libmuroar  
  1.1 Connecting to server and creation of streams .............. 2
  1.2 Portable IO with extended IO functions ................... 3
  1.3 Checking if a server exists ............................... 3
  1.4 Sending a notify beep .................................. 4

2 Quick reference  
  2.1 Constants and Magics ................................... 7
     2.1.1 Directions and Stream Types ...................... 7
     2.1.2 Codecs ........................................... 8
     2.1.3 Network Constants ................................. 8
     2.1.4 Command Constants ................................. 8
  2.2 Basic functions ....................................... 9
  2.3 Extended IO functions ................................. 10
Chapter 1

Programming with libmuroar

1.1 Connecting to server and creation of streams

In order to create a stream on a server you first need to open the connection to the server. After the connection has been successfully established you can connect the stream to the server.

To connect you have to call the function `muroar_connect()` like this:

```c
int fh;
if ( (fh = muroar_connect(server_address, "myApp")) == -1 ) {
    // handle errors here
}
```

After the connection has been established you now can connect the stream to the server. To do this you need to call the function `muroar_stream()` like this:

```c
if ( muroar_stream(fh, MUROAR_PLAY_WAVE, NULL, MUROAR_CODEC_PCM,
            rate, channels, bits) == -1 ) {
    // handle errors here
}
```

Now the stream is ready to write audio data to it. In case you want to write MIDI or Light Control data you need to change the direction parameter. See section 2.1.1 for a list of possible values. In case you want to use non native signed PCM data you need to change the codec to the correct value. See section 2.1.2 for symbolic codec constants.

At end of stream you need to close the socket. This need to be done by calling `muroar_close()` like this:
On how to do portable IO on your stream see the next section.

1.2 Portable IO with extended IO functions

As different Operating Systems need different functions to read and write to sockets you should consider using µRoar’s portable IO functions for reading and writing. They will handle all the Operating System depending stuff.

Those functions work basically like the read() and write() function on a POSIX system. They will return the number of bytes actually read or written or −1 on error. If they do not return the same number of bytes as requested you should wait before calling them another time. Calling them again with the rest of the buffer is not needed as they will loop internally until really no data can be read or written at the moment any more.

Example of reading some data and writing them to another socket:

```c
ssize_t ret_read, ret_write;
ret_read = muroar_read(fh, buffer, buffer_len);
if ( ret_read == -1 ) {
    // handle errors here
} else if ( ret_read == 0 ) {
    // nothing left or EOF
} else {
    ret_write = muroar_write(fh, buffer, ret_read);
    if ( ret_write == -1 ) {
        // handle errors here
    } else if ( ret_write < ret_read ) {
        // not all bytes have been written
    } else {
        // all bytes have been written, no error
    }
}
```

1.3 Checking if a server exists

In order to check the presence of a server you can simply call muroar_connect() and check the return value. If the function returns -1 the server is not existing at the given address or does not respond.
If the connection can be established you need to call `muroar_quit()` to
disconnect from the server. The socket is closed and all used buffers are freed by
using this function. You must not call `muroar_close()` after this call returned.

A complete example of a testing function may look like this:

```c
// return a true value in case a server is found:
int test_server(char * server) {
    int fh;

    if ( (fh = muroar_connect(server, "myApp")) == -1 )
        return 0;

    if ( muroar_quit(fh) == -1 )
        return 0;

    return 1;
}
```

1.4 Sending a notify beep

To send a notify beep you first need to connect to the server using `muroar_connect()`. After you have connect you can send a notify beep using `muroar_beep()`. The function takes a open file handle to the server that is not connected to a stream. You must not call `muroar_stream()` on the file handle before calling `muroar_beep()`. After you are done you can disconnect using `muroar_quit()`.

The beep will end as soon as it finish (the playback time of the beep is
over) or you close the connection to the server. This means that you need to
keep the connection up as long as the beep is still playing. If your application
uses beeps regularly you should keep the connection to the server open and close
it at program termination.

A simple example:

```c
#include <muroar.h>
#include <unistd.h>

int main (void) {
    int fh;

    if ( (fh = muroar_connect(NULL, "muroarbeep")) == -1 )
        return 1;

    muroar_beep(fh); // may return -1 on error.
        // real world applications need to check this!

    sleep(1);
```
muroar_quit(fh); // we ignore errors here, too, as above.

    return 0;
}

A more complex example using a global connection.

#include <muroar.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

void do_something(int i);

int roar_connection = -1;

static void roar_exit_callback (void) {
    if ( roar_connection != -1 )
        muroar_quit(roar_connection);
}

int main (void) {
    int i;

    roar_connection = muroar_connect(NULL, "muroarbeep");
    if ( roar_connection == -1 ) {
        fprintf(stderr, "Can not connect to RoarAudio Server!\n");
        return 1;
    }

    atexit(roar_exit_callback);
    for (i = 0; ; i++)
        do_something(i);

    return 1;
}

void do_something(int i) {
    printf("%i...
", i);

    if ( i % 2 )
        muroar_beep(roar_connection); // we ignore errors here,
                                        // real world applications
                                        // need to check!

    sleep(1);
    if ( i == 8 )
        exit(0);
This example program counts from zero to eight beeping every odd number.

The output with <beep> being a beep via RoarAudio looks like this:

```
0...
1... <beep>
2...
3... <beep>
4...
5... <beep>
6...
7... <beep>
8...
```

The function `roar_exit_callback()` in the given example will disconnect the connection to the server as soon as the program terminates normally using return from `main()` or `exit()`.
Chapter 2

Quick reference

2.1 Constants and Magics

2.1.1 Directions and Stream Types

#define MUROAR_PLAY_WAVE 1
#define MUROAR_PLAY_MIDI 12
#define MUROAR_PLAY_LIGHT 14
#define MUROAR_RECORD_WAVE 2
#define MUROAR_MONITOR_WAVE 3
#define MUROAR_MONITOR_MIDI 13
#define MUROAR_MONITOR_LIGHT 15
#define MUROAR_FILTER_WAVE 4

Those constants are the symbolic names for the direction parameter of the `muroar_stream()` function. The name consists of two important parts: the real direction and the subsystem the stream should use. The direction can be `PLAY` for a playback stream (a stream that sends data to the sound server) or `MONITOR` for monitoring streams which are used to read the mixed signal from the sound server. In addition there is `RECORD` and `FILTER` which are currently only supported by the Waveform Subsystem. A record stream is a stream that reads the audio data from one stream before it is mixed. This is similar to using a thru stream. The only difference is that a record stream supports transcoding. As this is not needed by other subsystems you may simply use thru streams. The filter stream is a stream that can be used to filter audio data. You need to read data from it, filter it, and write it back to the stream. You need to use exactly the same values for the stream as the server’s mixer has. Filter streams are currently only supported by the Waveform Subsystem.

The last part of the name is the name of the subsystem to use. `WAVE` for the Waveform Subsystem used by PCM or PCM like data, `MIDI` for MIDI music and `LIGHT` for the Light Control Subsystem which may be used to control a light setup. For Audio playback you normally want to use the Waveform Subsystem.
2.1.2 Codecs

Constant codecs:

```c
#define MUROAR_CODEC_PCM_S_LE 0x01
#define MUROAR_CODEC_PCM_S_BE 0x02
#define MUROAR_CODEC_PCM_S_PDP 0x03
#define MUROAR_CODEC_PCM_U_LE 0x05
#define MUROAR_CODEC_PCM_U_BE 0x06
#define MUROAR_CODEC_PCM_U_PDP 0x07
#define MUROAR_CODEC_OGG_VORBIS 0x10
#define MUROAR_CODEC_RIFF_WAVE 0x20
#define MUROAR_CODEC_ALAW 0x30
#define MUROAR_CODEC_MULAW 0x34
#define MUROAR_CODEC_MIDI 0x60
#define MUROAR_CODEC_DMX512 0x70
#define MUROAR_CODEC_ROARDMX 0x71
```

Those constants are the symbolic names for the used codecs. For default and native byte order constants see the next section.

Magical codecs:

```c
#define MUROAR_CODEC_PCM_S MUROAR_CODEC_PCM_S_ *
#define MUROAR_CODEC_PCM_U MUROAR_CODEC_PCM_U_ *
#define MUROAR_CODEC_PCM MUROAR_CODEC_PCM_S
```

Those constants are aliases for the similar named constants for the native byte order. For example on a big endian system `MUROAR_CODEC_PCM_S` will be an alias for `MUROAR_CODEC_PCM_S_BE`. `MUROAR_CODEC_PCM` is an alias for `MUROAR_CODEC_PCM_S`. Those constants are to be used as default codecs for the waveform subsystem.

2.1.3 Network Constants

```c
#define MUROAR_PORT 16002
```

The constant `MUROAR_PORT` defines the default TCP port for the RoarAudio protocol.

2.1.4 Command Constants

```c
#define MUROAR_CMD_NOOP 0
#define MUROAR_CMD_IDENTIFY 1
#define MUROAR_CMD_AUTH 2
#define MUROAR_CMD_NEW_STREAM 3
#define MUROAR_CMD_SET_META 4
#define MUROAR_CMD_EXEC_STREAM 5
#define MUROAR_CMD_QUIT 6
#define MUROAR_CMD_GET_STANDBY 7
#define MUROAR_CMD_SET_STANDBY 8
```
CHAPTER 2. QUICK REFERENCE

#define MUROAR_CMD_SERVER_INFO 9
#define MUROAR_CMD_SERVER_STATS 10
#define MUROAR_CMD_SERVER_OINFO 11
#define MUROAR_CMD_ADD_DATA 12
#define MUROAR_CMD_EXIT 13
#define MUROAR_CMD_LIST_STREAMS 14
#define MUROAR_CMD_LIST_CLIENTS 15
#define MUROAR_CMD_GET_CLIENT 16
#define MUROAR_CMD_GET_STREAM 17
#define MUROAR_CMD_KICK 18
#define MUROAR_CMD_SET_VOL 19
#define MUROAR_CMD_GET_VOL 20
#define MUROAR_CMD_CON_STREAM 21
#define MUROAR_CMD_GET_META 22
#define MUROAR_CMD_LIST_META 23
#define MUROAR_CMD_BEEP 24
#define MUROAR_CMD_GET_STREAM_PARA 27
#define MUROAR_CMD_SET_STREAM_PARA 28
#define MUROAR_CMD_ATTACH 29
#define MUROAR_CMD_PASSFH 30
#define MUROAR_CMD_GETTIMEOFDAY 31
#define MUROAR_CMD_WHOAMI 32
#define MUROAR_CMD_OK 254 /* return value OK */
#define MUROAR_CMD_ERROR 255 /* return value ERROR */

The above constants define the values for the command parameter used in the messages on the wire. They are of no interest for most developers. For their meaning please see the complete RoarAudio Manual.

2.2 Basic functions

int muroar_connect(char * server, char * name);

This function connects to a RoarAudio server. It opens a control connection.

int muroar_quit (int fh);

This function closes a control connection opened with muroar_connect().

int muroar_stream (int fh, int dir, int * stream,
                  int codec, int rate, int channels, int bits);

This function creates, connects and execs a stream on the already connected control connection converting the connection into a data connection.

int muroar_close (int fh);

This function closes a data connection created with muroar_stream().

int muroar_bEEP (int fh);
This function asks the server to do a simple beep. You can use this on file handles opened with `muroar_connect()` which are not used with `muroar_stream()` nor closed with `muroar_close()`.

### 2.3 Extended IO functions

```c
ssize_t muroar_write (int fh, const void * buf, size_t len);
ssize_t muroar_read   (int fh, void * buf, size_t len);
```

Those both functions behave mostly like the POSIX functions `read(2)` and `write(2)` expect that they will retry reading or writing until a final error happens. This is done to avoid different maximum write sizes on different operating systems so the application does not need to care about them.