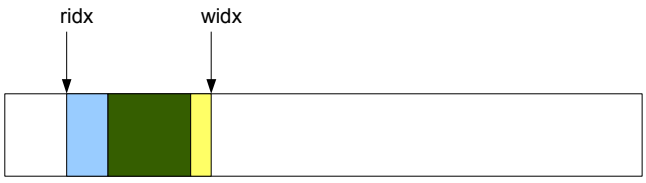
This paper aims to explain the an modification I propose to apply to Rockbox's buffering system. This enhancement should enable support for buffering multiple files simultaneously without significant performance overheads for current codecs' buffering behaviors.

First, I will discuss how Rockbox's current buffering system functions. The buffer is a ring, with two variables, “buf_widx” and “buf_ridx” respectively referring to the current write and read indices. The buffer is filled with “handles”, one per file, represented by struct memory_handle. Each handle has exactly one file associated with it, and some auxiliary state such as an offset and its own read and write index. The handles are stored in the buffer along with their data, and are identified by a numeric ID. Each handle also contains a pointer to the “next” one in the buffer, and global variables hold both the first and last handles' locations. This allows both looking up a handle by its ID and determining when reading data into a handle would cause it to cross into the next handle (which would be A Bad Thing).

This scheme has several limitations. First off, there is only one “active” handle. In order to add a new handle, the previous one must be “finished” by reserving enough space in the buffer to accommodate however much of its corresponding file has yet to be read. Thus, although multiple files can be “being buffered” at the same time, all but one of them must have complete space reservations; if we were to call add_handle twice in a row, the buffer must be larger than the size of the file passed to the first call in order for the second call to succeed.

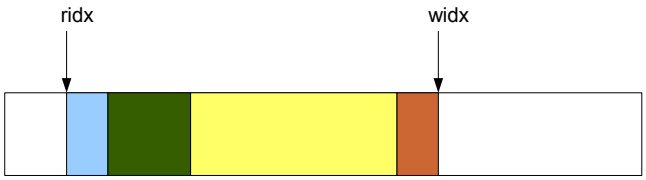
Secondly, the current scheme has special handling code for many different types of file, such as metadata information, bitmap images, cuesheets, codecs, etc. This is a consequence of the one-file limitation mentioned above.

Let's look at an illustration of how the current buffering code proceeds in the normal course of playback.



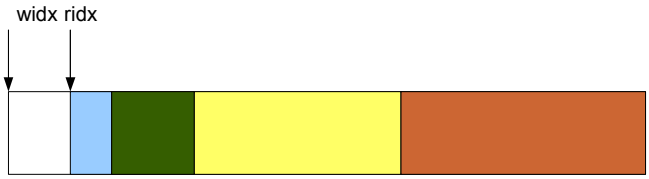
There are several handles already in the buffer, corresponding to codecs, metadata, and so on. Let's assume that the yellow handle on the right has some data that are not yet resident in memory. Although not shown, the first sizeof(struct memory_handle) bytes of each allocated region consists of its handle.

Now, what happens if we want to allocate a second handle? Let's say that the yellow area is our currently playing mp3 and we want to skip to the next track. add_handle must be called to start us up on buffering the new handle.

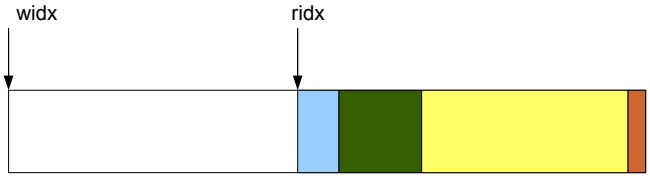


The result is above. Remember, we must “finish” the yellow handle; here the space reserved for it on the buffer is expanded to match the amount remaining to be read from its source file. In the event that the yellow region has insufficient room to grow, add_handle will fail. After growing the yellow handle, we place our new orange one at buf_widx and set current_handle to point to the new one.

Now let's say we proceed with buffering our orange handle. As we do so, the amount of space occupied by this handle increases until it reaches the end of the buffer. Oops! But no worries, this is a ring buffer; because this is a PACKET_AUDIO type handle it's allowed to wrap around.



Note that buf_widx < buf_ridx now. As we continue buffering, we'll fill from the start of the buffer. We're going to have a problem, though – the blue, green, and yellow handles aren't moving or being released. When our orange handle runs into the blue one the buffering thread will stop expanding the orange handle. If the blue handle were to be freed we would happily consume its space (and continue growing provided the green and yellow handles are released until we hit the start of the orange handle; if the orange handle fills the entire buffer we must abort because we just don't have enough buffer space to do what the codec is asking), but that can't happen in this example.



Shown above is one possible solution to our problem when we run into the blue handle. We simply move the blue handle to another location. Another potential solution is to copy what we've already buffered of the orange handle to a new location (probably just after the yellow handle) and continue growing from there. Although the latter solution is what the current buffering code does, that's not really relevant: the important thing is that some copying/moving of data already in the buffer must take place. This has a detrimental effect on performance.

Now, let's look at extending these methods to support multiple “current” handles (handles which are still being buffered and do not have 100% of the space they require already reserved in the buffer). To do this, I propose we allow the caller requesting a handle to be able to ask for “chunky” handles. A “chunky” handle will behave just like a normal one, EXCEPT in the following case:

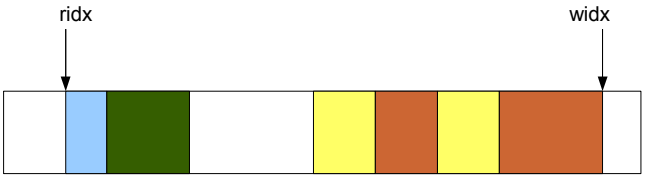
- A new “chunky” handle is added while the current handle is already “chunky”.

In this case, instead of “finishing” the first chunky handle, we only “finish” a chunk of it. From now on the chunky files may consist of a series of handles in the buffer, each corresponding to the same file, but having different offsets and potentially different sizes, as shown below:

Although multiple handles may correspond to the same file, only the first one has the file's true read index and only the last has the file's write index. Each chunk contains a pointer not only to the next handle in the buffer, but also to the next handle corresponding to the same file. Thus, when a method copying data from the buffer is called, it will traverse a series of (potentially) small handles, deallocating any which are exhausted in the process.

Because this methodology only has an effect when two “chunky” files are allocated in sequence, it has no impact on “normal” codecs. If a “normal” handle is added, buffer space for any “chunky” handles in progress must be completely allocated. Because of this behavior, the performance impact of this technique on existing codecs is very minimal.

Now, the benefits. Because only a small amount of the file is requested at a time, and because there is an implicit ordering of the chunks within a file, it is possible for a “chunk” file to “fill in the gaps” in the buffer. Imagine that both the yellow and orange “chunky” handlesets in the above example were advanced. The buffer would look like this:



Is there really a good reason why the next allocation must happen at buf_widx? Because the files are "chunky", it is possible for the yellow file to grow into a new handle located to the right of the green handle, or even at the start of the buffer. This makes better use of available space. In fact, if ALL file allocations were "chunky", then any amount of space free in the buffer could be utilized. The only overhead is the space required to store the handle headers, a few dozen bytes per chunk.

There is a down side. Codecs which rely on reading small amounts of data repeatedly from the buffer (as opposed to using some fixed chunk side repeatedly or copying large amounts out into a codec-internal buffer) may have their performance negatively impacted by chunky allocations in the presence of other chunky files. But because a handle always grows when it is able, in the event that there is no fragmentation in the buffer there will be no memory copies beyond those that were originally present. And, as has been demonstrated, in the event that no file using "chunky" allocations is present there will be no fragmentation, the only issue is when "chunky" and "normal" handles must coexist in the buffer. And even in this case, the only loss is a few extra memory copy operations, an acceptable overhead for the buffer utilization and functionality gains provided by this method.