Scheduler Details

Tom Rondeau

www.trondeau.com

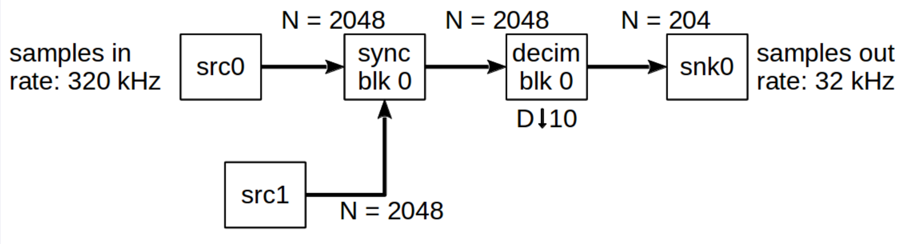
2013-09-26

Section 1

The Flowgraph

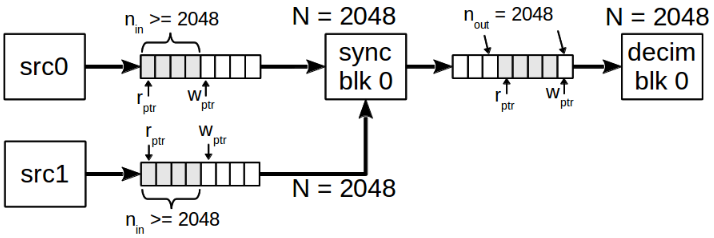
The flowgraph moves data from sources into sinks.

Example of data moving with rate changes.



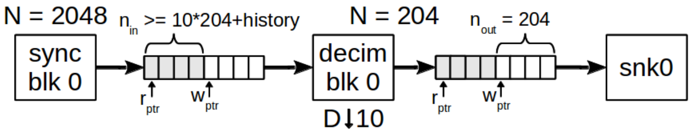
The flowgraph must check the bounds to satisfy input/output requirements.

All input streams and output streams must satisfy the constraints.



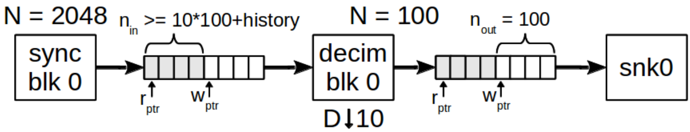
The boundary conditions can change with rate changing blocks.

Decimators need enough input to calculate the decimated output.



The conditions are independently established with each block.

This block is asking for less than it can on the input.



Section 2

The general_work and work functions

The input and output buffers

general_work / work have two vectors passed to it:

```
int
block::general_work(int noutput_items,
                    gr_vector_int &ninput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items)

int
block::work(int noutput_items,
            gr_vector_const_void_star &input_items,
            gr_vector_void_star &output_items)
```

- input_items is a vector of pointers to input buffers.
- output_items is a vector of pointers to output buffers.

general_work has not input/output relationship

It's told the number of output and input items:

```
int
block::general_work(int noutput_items,
                    gr_vector_int &ninput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items)
```

- noutput_items: minimum number of output available on all output buffers.
- ninput_items: vector of items available on all input buffers.

Number of input and output items?

noutput_items: how many output items work can produce

- general_work: no guaranteed relationship between inputs and outputs.
- work: knowing noutput_items tells us ninput_items based on the established relationship
  - gr::sync_block: ninput_items[i] = noutput_items
  - gr::sync_decimator: ninput_items[i] = noutput_items*decimation()
  - gr::sync_interpolator: ninput_items[i] = noutput_items/interpolation()
- Because of the input/output relationship of a sync block, only need to know one side

work operates off just $\text{noutput}_{\text{items}}$

From this number, we infer how many input items we have:

```
int
block::work(int noutput_items,
            gr_vector_const_void_star &input_items,
            gr_vector_void_star &output_items)
```

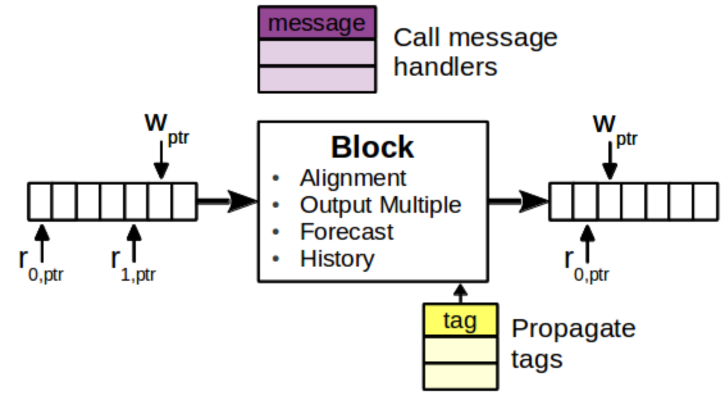
- noutput_items: minimum number of output available on all output buffers.
- ninput_items: calculated from noutput_items and type of sync block.

Section 3

Scheduler's job

Overview

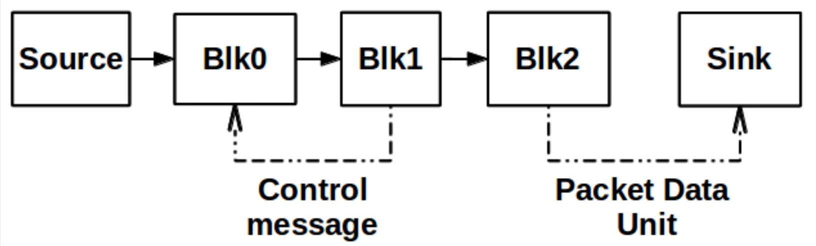
The scheduler handles the buffer states, block requirements, messages, and stream tags



Message Passing Layer

Send commands, metadata, and packets between blocks

Asynchronous messages from and to any block:



- `tb.msg_connect(Blk1, "out port", Blk0, "in port")`
- `tb.msg_connect(Blk2, "out port", Sink, "in port")`

Scheduler Handles the Asynchronous Message Passing

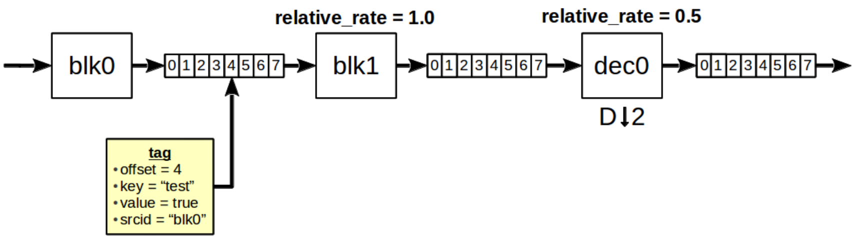
Asynchronous Message Passing:

1. When a message is posted, it is placed in each subscribers queue.
2. Messages are handled before `general_work` is called.
3. The scheduler dispatches the messages:
    1. Checks if there is a handler for the message type.
        1. If there is no handler, a queue of `max_nmsgs` is held.
        2. Oldest message is dropped if more than `max_nmsgs` in queue.
        3. `max_nmsgs` is set in preferences file in [DEFAULT]:max_messages.
    2. Pops the message off the queue.
    3. Dispatches the message by calling the block's handler.

Stream tag layer

Adds a Control, Logic, and Metadata layer to data flow

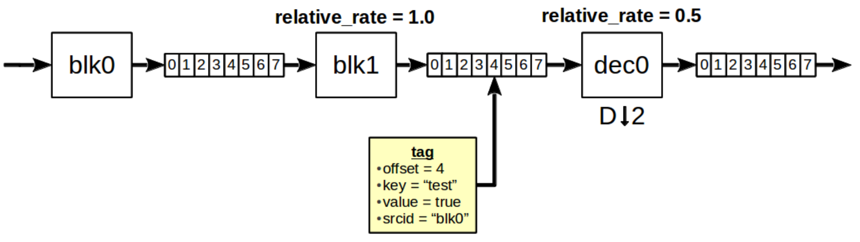
Tags carry key/value data associated with a specific sample.



Stream tag layer

Adds a Control, Logic, and Metadata layer to data flow

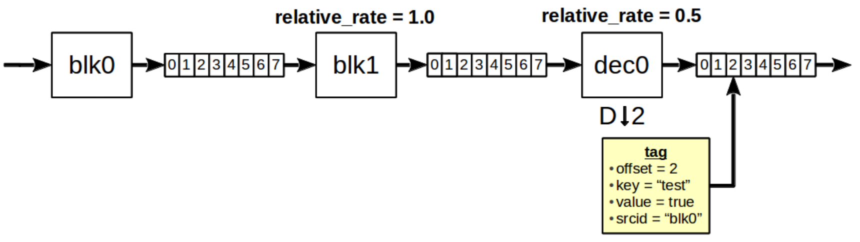
Tags are propagated downstream through each block.



Stream tag layer

Adds a Control, Logic, and Metadata layer to data flow

Tags are updated by data rate changes.



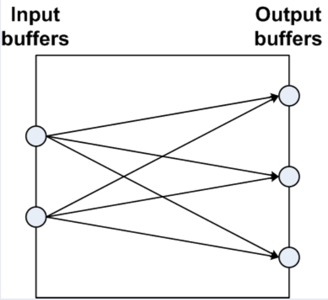
Propagate tags downstream based on the `tag_propagation_policy`

Tag propagation:

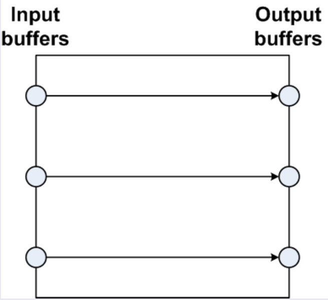
1. `tag_propagation_policy` typically set in block's constructor.
   1. Defaults to `block::TPP_ALL_TO_ALL`.
2. Called after `general_work`.
3. If propagating:
   1. Gets tags in window of last work function.
   2. If `relative_ratio` is 1, copies all tags as is.
   3. Otherwise, adjusts `offset` of tag based on `relative_ratio`.

Review of propagation policies

block::TPP_ALL_TO_ALL



block::TPP_ONE_TO_ONE



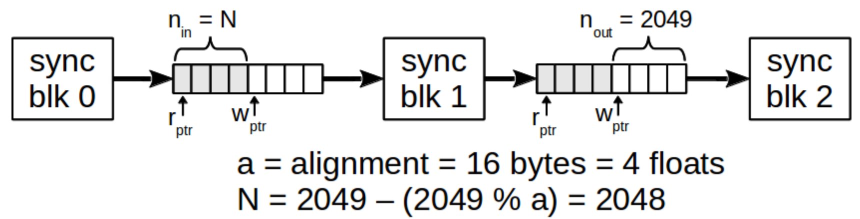
block::TPP_DONT

- Tags are not propagated and are removed from the stream.
- Can allow block to handle propagation on its own.

Alignment

`set_alignment(int multiple)`

Set alignment in number of items.

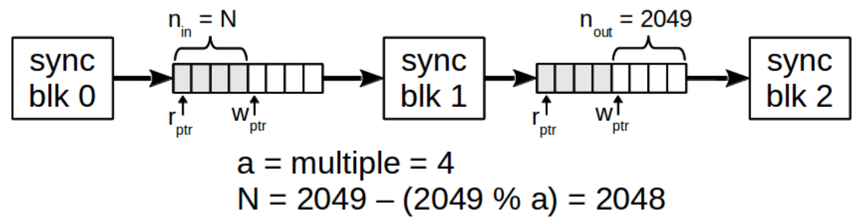


- Restricts number of items available to multiple of alignment.
- Not guaranteed, but recovers quickly if unalignment unavoidable.

Output Multiple

`set_output_multiple(int multiple)`

Set output multiple in number of items.



- Restricts number of items available to set multiple.
- Similar to alignment, but this is guaranteed.
- If not enough for alignment, will wait until there is.
- Cannot be set dynamically.

Forecast

Overloaded function of the class

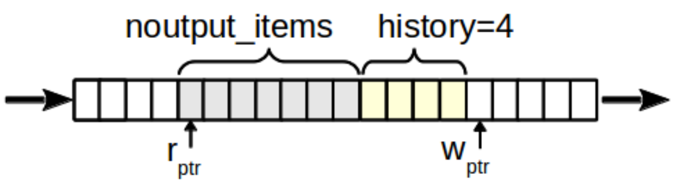
Tells scheduler how many input items are required for each output item.

- Given `noutput_items`, calculates `ninput_items[i]` for each input stream.
  - Default: ninput_items[i]=noutput_items+history()-1;
  - Decim: ninput_items[i]=noutput_items*decimation()+history()-1;
  - Interp: ninput_items[i]=noutput_items/interpolation()+history()-1;
- Use this to reduce the book-keeping checks in a block.
  - Can guaranteed `ninput_items[i] > noutput_items`
  - Don't have to check both conditions.

History

$\text{set}_{\text{history}}(\text{nitems}+1)$

History sets read pointer `history()` items back in time.



- Makes sure we have valid data `history()` items beyond `noutput_items`.
- Used to allow causal signals between calls to `work`.

Buffer Size and Controlling Flow and Latency

Set of features that affect the buffers

- `set_max_noutput_items(int)`
  - Caps the maximum `noutput_items`.
  - Will round down to nearest output multiple, if set.
  - Does not change the size of any buffers.
- `set_max_output_buffer(long)`
  - Sets the maximum buffer size for all output buffers.
  - Buffer calculations are based on a number of factors, this limits overall size.
  - On most systems, will round to nearest page size.
- `set_min_output_buffer(long)`
  - Sets the minimum buffer size for all output buffers.
  - On most systems, will round to nearest page size.

Scheduler Manages the Data Stream Conditions

General tasks:

1. Calculate how many items are available on the input.
2. Calculate how much space is available on the output.
3. Determine restrictions: alignment, `output_multiple`, forecast requirements, etc.
4. Adjust as necessary or abort and try again.
5. Call the `general_work` function and pass appropriate pointers and number of items.
6. Take returned info from `general_work` to update the pointers in the `gr::buffer` and `gr::buffer_reader` objects.

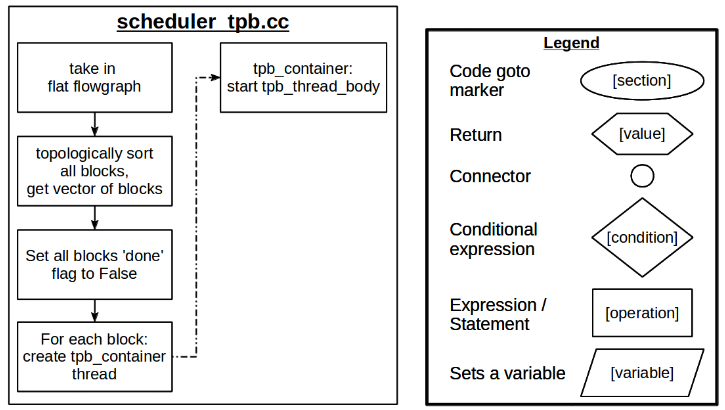
Section 4

Scheduler Flow Chart

Scheduler Flow Chart: `top_block.start()`

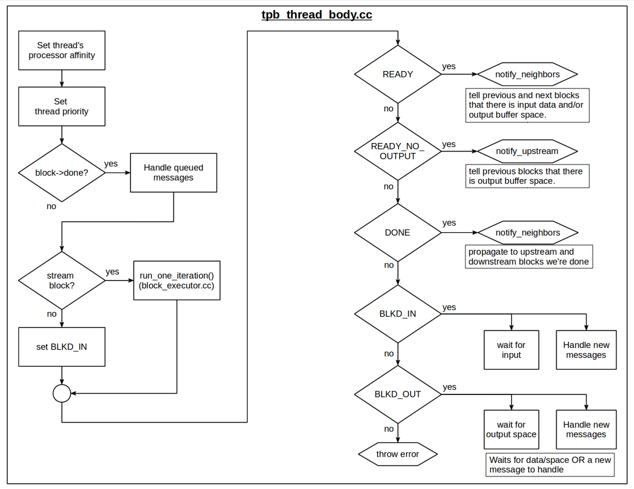
Start in `scheduler_tpb.cc`

Initialize thread for each block:



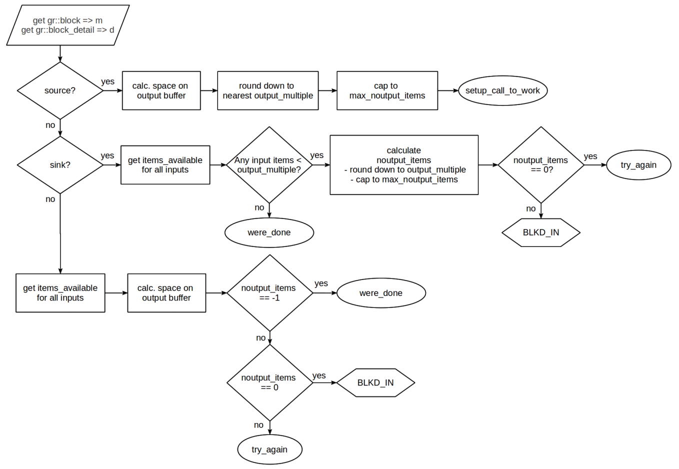
Each block's thread runs the loop until done

Handles messages, state, and calls `run_one_iteration`:



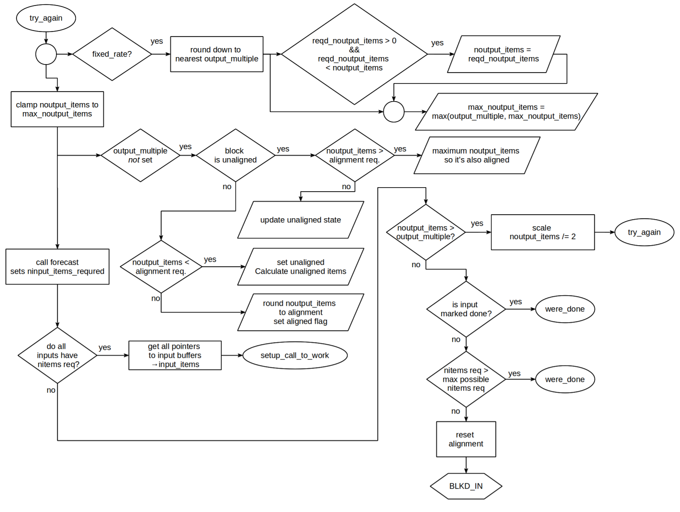
run_one_iteration in block_executor.cc

Start of the iteration:



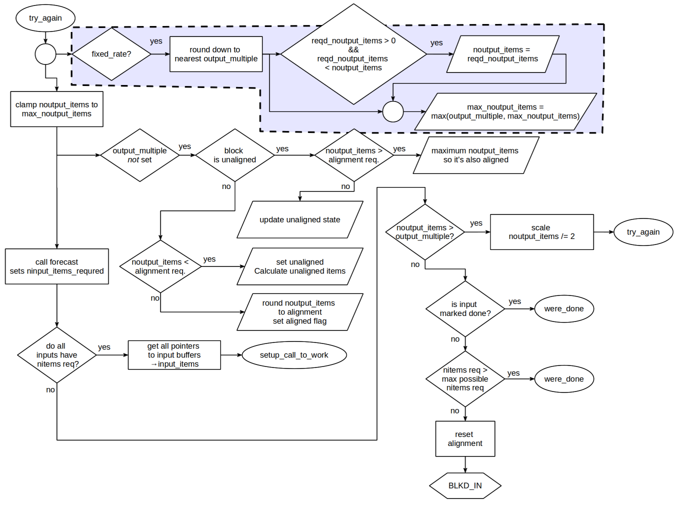
run_one_iteration::try_again

If block has inputs (sinks/blocks), handle input/output reqs.:



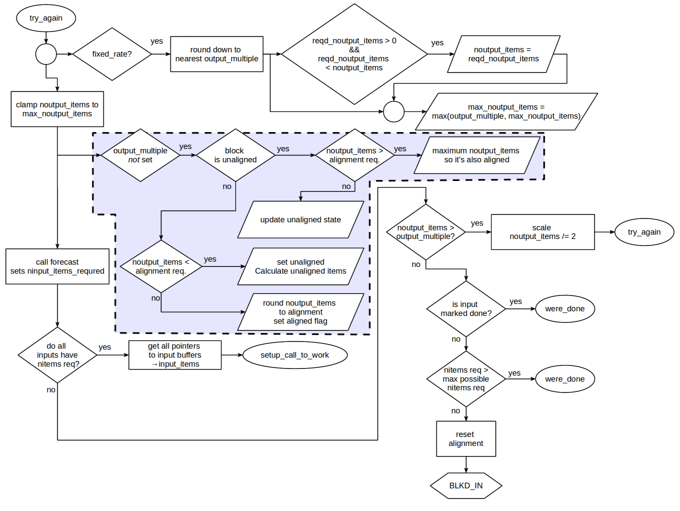
run_one_iteration::try_again: Fixed Rate

Fixed rate blocks have special restrictions:



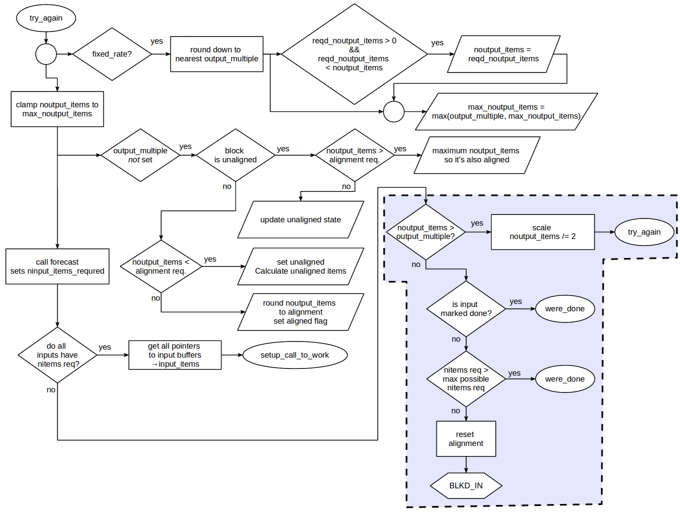
run_one_iteration::try_again: Alignment

Works to keeps buffers aligned if possible:



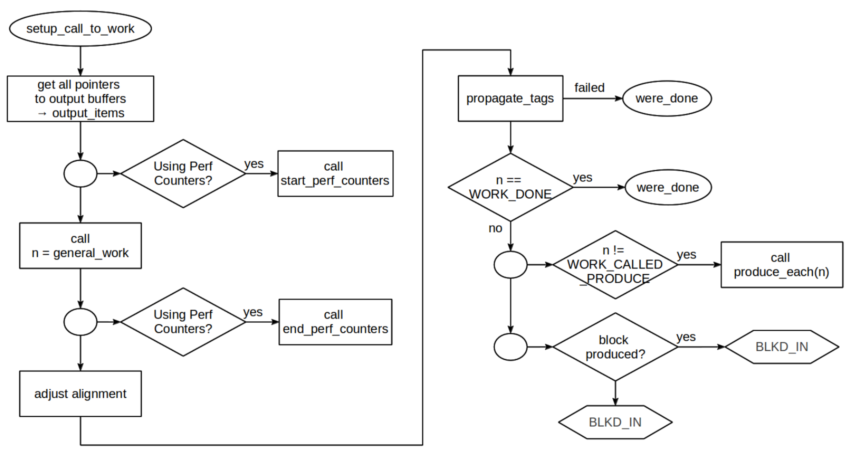
run_one_iteration::try_again: Failure

If something goes wrong, try again, fail, or block and wait:



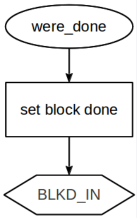
run_one_iteration::setup_call_to_work

Call `work` and do book-keeping:



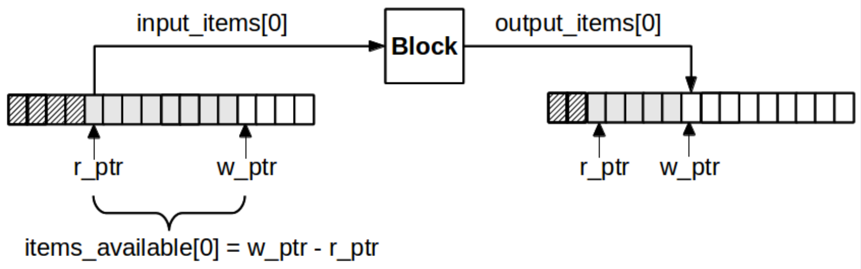
run_one_iteration::were_done

When the flowgraph can't continue, end it:



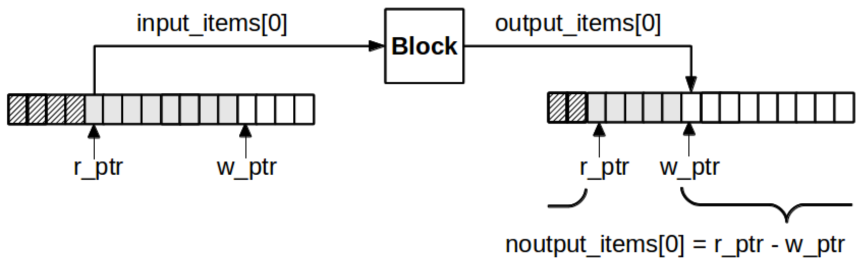
"Get items_available for all inputs"

Gets difference between write pointers and read pointers for all inputs:



"Calc space on output buffer"

Space available is the difference between write pointers to the first read pointer. `noutput_items` is the minimum for all output buffers:



"call forecast, sets ninput_items_required"

Given noutput_items, forecast calculates the required number of items available for each input.

```
void
sync_decimator::forecast(int noutput_items,
                         gr_vector_int &ninput_items_required)
{
  unsigned ninputs = ninput_items_required.size();
  for(unsigned i = 0; i < ninputs; i++)
    ninput_items_required[i] = \
        fixed_rate_noutput_to_ninput(noutput_items);
}

int
sync_decimator::fixed_rate_noutput_to_ninput(int noutput_items)
{
  return noutput_items * decimation() + history() - 1;
}
```

"Do all inputs have nitems req.?"

Tests that `items_available[i]` >= `ninput_items_required[i]` for all `i`.

- If yes, run the `setup_call_to_work` section.
- Otherwise, we're in a fail mode:
  - If we still have enough output space, goto `try_again`.
  - If the input is marked done, goto `were_done`.
  - If block requires more than is possible, goto `were_done`.
  - Otherwise, we're blocked so we exit and will start over on next iteration.

Section 5

Buffer Creation

Buffers are handled almost completely behind the scenes

Standard Creation

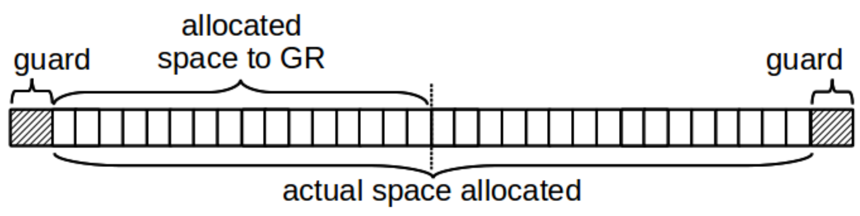
- GNU Radio selects the best option for how to create buffers.
- Allocated at the start of a page.
- Length is a multiple of the page size.
- Memory mapped second half for easy circular buffer.
- Guard pages one either side.

User controls

- Minimum buffer size.
- Maximum buffer size.

Circular buffers in memory

Shows guard pages and memory-mapped half



Buffer creation techniques

Controlled by the `vmcircbuf` classes

- Selects from:
  - `vmcircbuf_createfilemapping.h`
  - `vmcircbuf_sysv_shm.h`
  - `vmcircbuf_mmap_shm_open.h`
  - `vmcircbuf_mmap_tmpfile.h`
- Reads from a preference file, if set.
- Tests all factories, saves preferred to preference file.

Buffer creation: Create File Mapping

Generally used for MS Windows

- `size` required to be a multiple of the page size.
  - Uses CreateFileMapping to get a handle to paging file.
- Allocates virtual memory of `2*size`.
  - Uses VirtualAlloc to get `first_tmp`.
- Map the paging file to the first half of the virtual memory.
  - Uses MapViewOfFileEx with `first_tmp` as pointer base.
- Map the paging file to the second half of the virtual memory.
  - Uses MapViewOfFileEx with `first_tmp+size` as pointer base.
- Both first and second half are mapped to the same paging file.

Buffer creation: Memory-mapped Temp File

Generally used for OSX

- `size` required to be a multiple of the page size.
  - Creates a temp file with permissions 0x0600.
- Uses `unlink` to hide file and remove it when program closes.
- Sets length of temp file to `2*size`.
  - Uses `ftruncate`.
- Map the first half of the file to a pointer `first_copy`.
  - Uses `mmap` to point to start of temp file.
- Map the second half of the file to a pointer `second_copy`.
  - Uses `mmap` to point to `first_copy+size`.
- Resets temp file to `size` with `ftruncate`.
- Uses `first_copy` as the buffer's base address.

Buffer creation: System V Shared Memory

Generally used for Linux/POSIX

- size required to be a multiple of the page size.
    - Uses shmget to get 2*size (plus guard pages) as schmid2.
    - Uses shmget to get size as shmid1.
- Attach shmid1 to first half of schmid2 with shmat.
- Attach shmid1 to second half of schmid2 with shmat.
- Memory in both halves of shmid2 are mapped to the same virtual space.
- Keep guard pages as read-only.
- Return memory in shcmid2+pagesize as buffer base location.
- Keeps 2*size allocated.

Buffer creation: Memory-mapped Shared Memory

Alternative implementation for Linux/POSIX

- `size` required to be a multiple of the page size.
  - Creates a shared memory segment with `shm_open`.
- Sets length of memory segment to `2*size`.
  - Uses `ftruncate`.
- Map the first half of the file to a pointer `first_copy`.
  - Uses `mmap` to point to start of memory segment.
- Map the second half of the file to a pointer `second_copy`.
  - Uses `mmap` to point to `first_copy+size`.
- We should reset memory segment to `size` with `ftruncate`.
  - on OSX this isn't allowed, though; not actually compiled.
- Uses `first_copy` as the buffer's base address.

VM circular buffer preference setting

Working VM Circular Buffer technique is stored in a prefs file

- Handled by `vmcircbuf_prefs` class.
- Path:
  `$HOME/.gnuradio/prefs/vmcircbuf_default_factory`
- Single line that specifies the default factory function:
  - e.g., `gr::vmcircbuf_sysv_shm_factory`
- If no file, we find the best version and store it here.
- Should only be created once on a machine when GNU Radio is first run.

Building a gr::buffer

Buffers are built and attached at runtime

- When start is called, flowgraph is flattened and connections created.
- gr::block_details are created and a gr::buffer for each output.
  - Buffer size is calculated as the number of items to hold.
    - min/max restrictions applied, if set.
- Connects inputs by attaching a gr::buffer_reader.

Calculating gr::buffer size

gr::flat_flowgraph::allocate_buffer

- Takes in item_size.
- Calculates number of items: nitems = s_fixed_buffer_size*2/item_size.
  - s_fixed_buffer_size = 32768 bytes.
  - doubling the size to allow double buffering.
- Checks that nitems is at least 2x output_multiple.
- Checks max_output_buffer & min_output_buffer settings.
  - Both default to -1, which means no limit.
- Checks that nitems is greater than decimation*output_multiple+history.
  - Must have enough to read in all of this at one time.

Calculating `gr::buffer` size: granularity

`gr::buffer::allocate_buffer` handles the actual creation

- Checks if we have the minimum number of items:
  - Based on system granularity (i.e., page size) and item size.
  - Rounds up to a multiple of this minimum number of items.
- Rounding up based on item size may result in unusual buffer sizes.
  - We handle this; just sends a warning to the user.
- Calls VM circular buffer default factory function.
- Sets `d_base`, the address in memory for the buffer, from the circular buffer.

Controlling the size of buffers: min/max

User interface allows us to set min/max buffer for all blocks

- Methods of `gr::block`:
  - `gr::block::set_min_output_buffer(port, length)`
  - `gr::block::set_max_output_buffer(port, length)`
- Methods to set all ports the same:
  - `gr::block::set_min_output_buffer(length)`
  - `gr::block::set_max_output_buffer(length)`
- Will still round up to the nearest granularity of a buffer.
- Can only be set before runtime to have an effect.

Section 6

Wrap-up

Review:

This presentation covered:

- The responsibility of the scheduler.
- And understanding of the user interaction with the scheduler.
- The roles the scheduler plays in the three data streams:
  - Overview of the data stream, message passing, and tag streams.
  - Alignment, output multiple, forecast, and history.
- Flow chart of the threaded loops each block runs.
  - Launching the thread body.
  - Handling messages.
  - calling `run_one_iteration` and its tasks.
- In-depth look into how the scheduler makes its calculations.
- Buffer structure, calculations.

Purpose:

From the information in this presentation, you should be able to:

- Better interact with the three data stream models
- Use the features of the data flow model (forecast, history, etc.) to improve logic, performance
- Understand how the buffer system works
  - and how to extend or alter it for different architectures