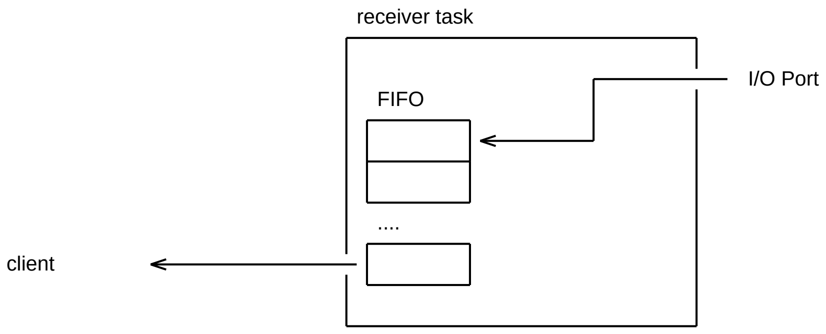
A buffered receiver

`version`	1.1.1
`scope`	Example. This code is provided as example code for a user to base their code on.
`description`	A buffered receiver
`boards`	Unless otherwise specified, this example runs on the SliceKIT Core Board, but can easily be run on any XMOS device by using a different XN file.

This example shows a receiver task that reads data off external I/O pins, and buffers that data. A client task can read the data out of the buffer.



Buffering data in this way decouples the client task from the receiver so that the client is not bound by the real time constraints of the I/O.

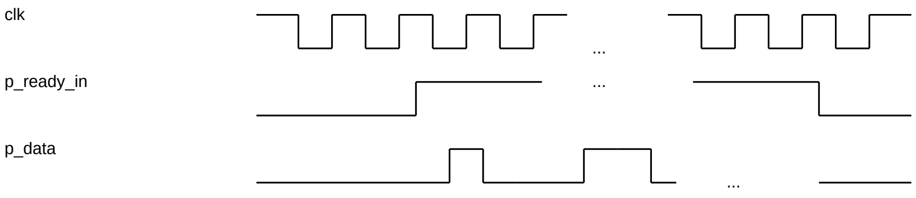
The receiver receives data on a simple clocked port, 32 bits of data at a time. When it receives data on the port, it will notify the client and place the data in a FIFO buffer. The client can then pull 32-bit words out of this FIFO.

The first thing to define in the program is the interface between the receiver task and its client.

```
interface receiver_if {

  // This *notification* function signals to the client when there is
  // data in the buffer. Since it is a notification function it is
  // special - instead of being called by the client it is used
  // by the client to event when
  // data is ready.
  [[notification]] slave void data_ready();

  // This function can be used by the client to pull data out of the fifo.
  // It clears the notification raised by the data_ready() function.
  [[clears_notification]] unsigned get_data();

  // This function can be called to check if the receiver has any data in
  // the buffer.
  // Generally, you do not need to poll the receiver with this function
  // since you can use the data_ready() notification instead
  unsigned has_data();
};
```

The receiver task takes arguments to set it up. It takes the server end of an interface connection using the `receiver_if` interface - this will be connected to the client. It also takes the buffer size required and the ports and clock block variable for using the external I/O pins. The I/O interface requires a data pin, a clock block and a pin to provide a readyIn signal.

The I/O protocol is a simple protocol directly supported by the XMOS Hardware-Response port blocks. The clock block provides a clock for the data. When the externally driven *p_ready_in* signal is driven high, it signals the start of the data. After that data is clocked into the *p_data* port on the rising edge of the clock. Since the data port is a `buffered` port, the data is deserialize into 32-bit chunks, so the program will receive a 32-bit word at a time from the port.



This is the prototype of the receiver task:

```
void receiver(server interface receiver_if i,
              static const unsigned bufsize,
              in buffered port:32 p_data,
              clock clk,
              in port p_ready_in)
{
```

Within definition of the task the first thing required is to define the local state. The `buffer` array provides the memory space for the FIFO. To implement a FIFO, the `fifo_first_elem` and `fifo_last_elem` hold the indices to the first and last element in the FIFO. All the array elements between these indices hold the data (the FIFO may wrap around the end of the array back to the beginning).

```
unsigned buffer[bufsize];
unsigned fifo_first_elem = 0, fifo_last_elem = 0;
```

The initial part of the task sets up the port block. The protocol on the pins is one supported by the Hardware-Response port blocks, so you can configure the port using a library function to set it to *Strobed Slave* mode (i.e. data input is governed by the readyIn signal). The port configuration functions are found in `xs1.h`.

```
configure_in_port_strobed_slave(p_data, p_ready_in, clk);
```

The main body of the task is a loop with a `select` inside. This select will either react to the port providing input, or to a request from the client over the interface connection:

```
while (1) {
  select {
  case p :> unsigned data:
    // handle port input
    ..
  case i.get_data() -> unsigned result:
    // request from client to get data, pop element off fifo
    // and place it in the return value 'result'
    ..
  case i.has_data() -> unsigned result:
   // request form client to determine if there is data in buffer
   // put 1 or 0 in the retrun value 'result'
   ..
  }
}
```

When the port signals that it has data, the tasks reads it into the `data` variable.

```
case p_data :> unsigned data:
```

To handle the port input the program works out where it needs to be added to the FIFO by adding one to the last element index (and wrapping round in the buffer if needed).

```
unsigned insert_point = fifo_last_elem + 1;
if (insert_point == bufsize)
  insert_point = 0;
```

If the insert point wraps all the way around to the beginning of the FIFO, there is buffer overflow. In this case the task just drops the data but different overflow handling code could be added here.

```
if (insert_point == fifo_first_elem) {
  //handle buffer overflow
  break;
}
```

If there is room in the buffer, the data is inserted into the array and the last element index is updated.

```
buffer[insert_point] = data;
fifo_last_elem = insert_point;
```

Finally, the server calls the `data_ready` notification. This signals to the client that there is some data in the buffer.

```
i.data_ready();
```

The following case responds to a client request for data. The return value back to the client is declared as a variable `result`. The body of this case can set this variable to pass a return value back to the client.

```
case i.get_data() -> unsigned result:
```

The data to extract from the FIFO is in the array at the position marked by the first element index variable. However, if this is the same as the last element then the buffer is empty. In this case the task returns the value 0 to the client, but different buffer underflow handling code could be added here.

```
if (fifo_first_elem == fifo_last_elem) {
  // handle buffer underflow
  result = 0;
  break;
}
```

To pop an element from the FIFO, the `result` variable needs to be set and the first element index variable needs to be incremented (possibly wrapping around the buffer array).

```
result = buffer[fifo_first_elem];
fifo_first_elem++;
if (fifo_first_elem == bufsize)
  fifo_first_elem = 0;
```

Finally, if the FIFO is not empty, the task re-notifies the client that data is available.

```
if (fifo_first_elem != fifo_last_elem)
  i.data_ready();
```

The final request the receiver task handles is from the client requesting whether data is available. This case is quite simple, just needing to return the current state based on the index variables for the FIFO.

```
case i.has_data() -> unsigned result:
  // request form client to determine if there is data in buffer
  // put 1 or 0 in the retrun value 'result'
  result = (fifo_first_elem != fifo_last_elem);
  break;
```