# Stream Buffers

INTRODUCTION

Stream buffers are a way to implement task to task or interrupt to task communication. A stream of bytes is written by the sender to a buffer, where a receiver can read the data. Similar functionality can be achieved with a queue but a stream buffer is optimized for a single reader single writer scenario, where the writer places an arbitrary number of bytes in the stream buffer and the reader reads an arbitrary number of bytes. It is a lightweight alternative to queues for the mentioned scenario.

WHAT IS A STREAM BUFFER?

As the name suggests, the data structure to store the data is a buffer which operates in a first-in-first-out manner. Writing to the buffer is done via API calls to xStreamBufferSend or xStreamBufferSendFromISR and data is added to the end of the buffer. To signal the availability of data to a possibly waiting task, the task notification is used.

The stream buffer has a trigger level, which is the least number of bytes that must be written to the stream buffer before the task notification is triggered. Therefore, the trigger level can be a value between 1 and the length of the stream buffer. The length of the buffer is finite, that means if reading from the buffer doesn't happen fast enough, the buffer may become full.

Writing to the buffer can therefore be blocking (only the non-ISR function may block) and a block time can be specified. To notify the waiting writer task about the availability of space, for example if a reader reads from the stream buffer, the task notification is used. If not, all data may be sent within the block time period.

The buffer is filled with as many bytes as there is space and the API call returns with an error code and the amount of data that was written. Most of the time, this is due to the processing of the sent data being too slow (reading task has too low priority or data to be written is generated faster than expected).

Reading from the stream buffer is done via API calls xStreamBufferReceive or xStreamBufferReceiveFromISR. They return a status code and, if successful, the read data. The amount of read data depends on the number of available bytes in the stream buffer and the size of the specified array for reading the data. A blocking time can be specified (only the non-ISR function might block), after which a timeout is generated if no data was available when calling the API function and if the task wasn't unblocked by a task notification as described above. Despite the timeout occurring, there might be data in the stream buffer but not enough to reach the trigger level (if trigger level is greater than one). If so, the read returns successful with the available data. If data is read from a full stream buffer, a task notification is send to the task that is waiting to write data but was blocked due to it being full.

EXAMPLE

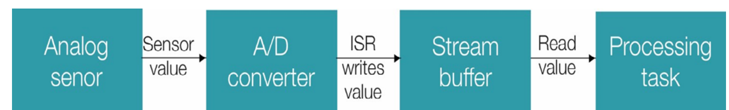


**Figure 1.** *Stream buffer example system*

An example application for the stream buffer is an analog to digital converter where an interrupt has generated whenever data is available. The converted data can be sent to the buffer by the interrupt service routine. The task which uses the data is then informed of the newly added data and can read from the stream buffer and process the data. For example, filter the data or calculate a controller step.

MESSAGE BUFFER

Message buffers are built on stream buffers but instead of sending and receiving an arbitrary length stream of bytes, discrete messages are exchanged. A message is of a defined length where the length of the message is prepended to the actual payload. This means when sending a message of 10 bytes the length of the message (the number 10) is added to the buffer prior to the actual payload, so the receiver can read this before reading the message to know how many bytes of payload to expect. The number of bytes to store the message length is configurable and must be predefined, so the receiver knows exactly how many bytes to read from the buffer to get the correct length of the next message. How to configure the number of bytes used to store the message length depends on the maximum possible message length, where a single byte length field can be used with messages up to 255 bytes and a 2-byte length field can hold message lengths up to 65535 bytes and so on.

The size of the buffer should also consider the maximum message length and the length field, for example a message of 10 bytes with a length field of 2 bytes takes 12 bytes in the buffer per message.

In SAFE**RTOS**®, stream buffers and message buffers use the same API functions. To distinguish the two, an additional flag in the xStreamBuffer struct is used, and can be specified at creation of the stream/ message buffer. The API functions behave slightly differently whether a stream buffer or message buffer is used.

As with stream buffers, reading and writing to a message buffer might be blocked if a non-zero block time is specified to the API functions (only the non-ISR functions can block). The behaviour of message buffers is very similar to stream buffers. A read operation from the message buffer will block if there is no message to receive in the message buffer. The task will block until either the block time expires, or enough data is sent to the message buffer to exceed the trigger level. As messages are never partially sent to a message buffer, in most cases a trigger level of 1 byte is the most useful, as it results in any single message unblocking a task waiting to read from the message buffer.

A write to the buffer can block if there is not enough space for the complete message until enough bytes/ messages are read from the buffer to make room for the message or the block time has passed. In the latter case, no data is added to the message buffer.

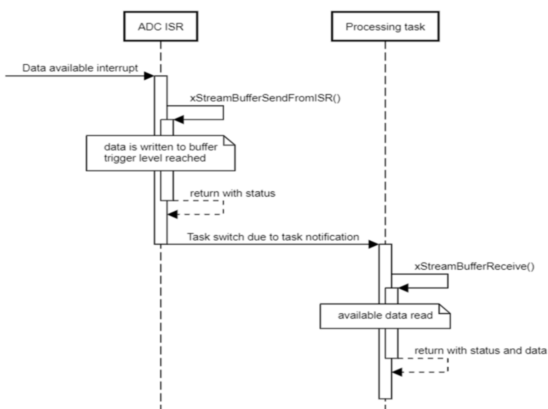


**Figure 2.** *Stream buffer sequence diagram*

## MULTICORE USAGE IN SAFE**RTOS**®

Stream buffers and message buffers can also be used to send data from a task/interrupt on one core to a task/interrupt on another core of a multicore microcontroller. In an asymmetric multi-processor configuration, each processor runs its own instance of the RTOS. To share data between the cores there must be shared memory that can hold the stream buffer data.

To notify the reading core about new data, an interrupt from one core to the other should be available.

The SAFE**RTOS**® API has several functions to deal with multicore stream buffer usage, but the necessary functionality is heavily dependent on the processor architecture. That's why these functions are weakly defined stubs and must be overwritten by the application code to implement the core specific functionality.

To notify the reading core about a completed write to the buffer, API functions vStreamBufferSendCompletedMulticore and vStreamBufferSendCompletedFromISRMulticore can be used. These functions are called if a task on another core tried to read from the empty stream/message buffer and is waiting for data to be sent. These functions are used instead of the task notification in a single core scenario and could trigger an interrupt in the waiting core that, in turn, unblocks the waiting reader.

Similarly, to notify the writing core about available space in the buffer, API functions vStreamBufferReceiveCompletedMulticore and vStreamBufferReceiveCompletedFromISRMulticore can be used. These functions are called if a task on another core tried to write to the full buffer and is waiting to add the data. These functions are used instead of the task notification in a single core scenario and could trigger an interrupt in the waiting core that in turn unblocks the waiting writer.

One important aspect of multicore usage is the coordination of the access to the stream/message buffer data. On a single core device, only one task or interrupt can be active at a time and if a task needs to access a data structure without being interrupted it might do so inside a critical section. With more than one core active at the same time, the parallel access to a shared data structure must be prevented. Therefore, a locking mechanism must be used to ensure core exclusive access to the data structure. SAFE**RTOS**® API has the functions xStreamBufferAttemptToLock and vStreamBufferReleaseLock to deal with this functionality. These functions are also weakly defined stubs that application code must override to implement the processor specific way to coordinate the access.

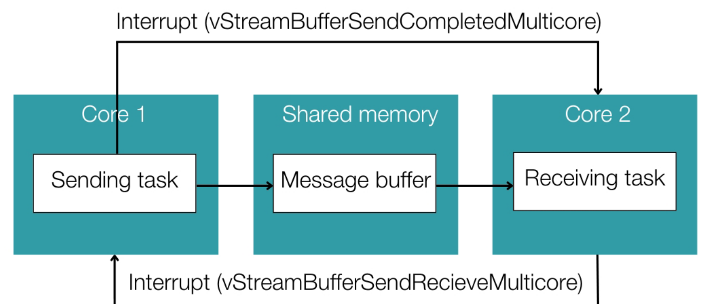


**Figure 3.** *Multicore example system*

WITTENSTEIN high **integrity** systems

Worldwide Sales and Support
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@high**integrity**systems.com
Web: www.high**integrity**systems.com

Copyright 2024 WITTENSTEIN high integrity systems Ltd. V1.4. Correct at time of issue.