Message Passing Between Loosely-Coupled Threads by Means of Flexible Ring Buffers

Martin Lottermoser
http://home.htp-tel.de/lottermose2

Version 1.8 (2014-06-12)

Contents

1 Message Passing Through Shared Memory

There are a number of situations where one wishes to asynchronously exchange messages between different threads of execution. The first thread should be able to take some data from its own storage area, "send" it to the other, and continue working. The second thread should then, at a later time when it is ready, be able to discover that a message has been sent and "receive" it, copying the data into its own private storage area. Typically we want to have certain assurances concerning the properties of the data transfer:

- Messages are not modified in passing.
- Messages are not lost or, if that can happen, recipient or sender are informed that something has been lost.
- If more than one message can be "in transit" at the same time, sender and recipient know in advance how the order of sending them maps to the order on reception. (For message passing as considered here, only FIFO semantics seem useful.)

But the first problem to solve is to ensure that the actions of these threads do not interfere with each other and data can still be transferred between the two thread-specific storage areas. In most cases the operating system will offer such functionality (inter-process communication), but sometimes we have to deal with threads running on separate processors which are not under the control of the same operating system instance, are not connected by a full-blown communication network, but are still tightly connected by hardware.

In these cases one sometimes uses "dual-port memory", i.e., RAM chips which can be read and written from two different sides. Often, such chips also offer additional hardware support (e.g., a hardware semaphore) which can be used to design algorithms for consistent access to data structures in the RAM part of the chip [1].

A similar situation arises if one uses shared memory provided by a common operating system instance, except that additional support for coordination between the two sides is typically offered by system calls.

In both cases we now have a common area of memory which we can use. Obviously, we have to divide the area into a number of buffers, have one side put its next message into the "next free" buffer, and let the other side fetch it. The interesting problems are how to use the storage area efficiently and how to keep the two threads using it from interfering with each other without introducing unnecessary coupling. This article proposes a model for doing that.

2 The Basics

2.1 Fundamental Building Blocks and Their Properties

A *flexible ring buffer (FRB)* is assumed to have two interfaces or "sides", a *read interface* and a *write interface*. Through these interfaces the following three memory areas are visible (see figure 1):

- a *buffer area*
- a *write offset area* for publishing a *write offset* (next position to write to)
- a *read offset area* for publishing a *read offset* (next position to read from)

The write interface must permit the following operations:

- writing to the buffer area,
- writing the write offset, and
- reading the read offset.

Similarly, the read interface must permit:

- reading from the buffer area,
- reading the write offset, and
- writing the read offset.

Read and write operations are assumed to be serialized on each side separately, but it is also assumed that operations on different interfaces do not interact to a certain extent:

- Access operations involving different areas (buffer area, write offset area, read offset area) are independent.

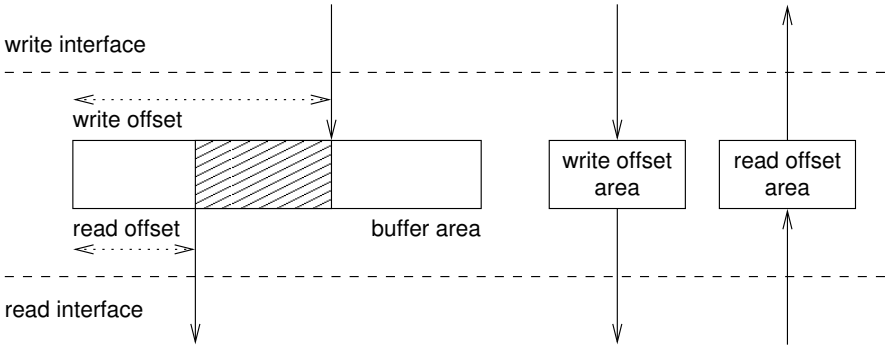

Figure 1: Building blocks of a flexible ring buffer

- Writing an offset is atomic, i.e., while a write operation is in progress from one of the interfaces, a sequence of read operations on the offset from the other interface will start yielding the old value and then switch to the new without jumping back or giving other values.
- The buffer area consists of a sequence of *independency units (IUs)* and has the property that any two access operations from different sides will not interfere with each other if the two storage areas, extended to the boundaries of independency units, do not overlap.

It should be noted that these properties need not be realized in the memory area implementations but may be shifted into the access operations (which would then need additional functionality not mentioned above). If, for example, a memory area has the property that the other side may not read while a write operation is in progress (perhaps the chip has sufficient power only for one of the operations), this can be solved by the writing side first establishing a state where reading will block or fail, then performing the writing, and finally permitting reading again. This, however, introduces a certain amount of coupling between the two threads involved and is inefficient because one thread has to wait for the other to finish; it is preferable to limit the necessity for such a situation. The good news, however, is that in particular the condition on the buffer area (independency for non-overlapping operations) can be satisfied by off-the-shelf hardware.

The condition imposed on the offset variables (atomic operations) seems to be somewhat trickier to satisfy at the hardware level. This is a point where hardware semaphore support becomes useful or even necessary. In such cases the offset areas are often located in the same memory region as the buffer area and occupy a number of dedicated IUs. Access operations on the offsets must then be protected by obtaining a semaphore first, unless each area takes up at most a region for which the hardware supports atomicity (some off-the-shelf hardware components do). Using semaphores for offset area protection is less problematic than in the case of the buffer area because the data size involved is constant and the duration of the interval the variable is blocked is therefore easier to control.

In designing access algorithms for an FRB one should keep in mind that reading or writing such inter-thread memory areas is often slower than operating on thread-specific storage. Accessing the FRB should therefore be limited to absolutely necessary operations.

2.2 Principles of Operation

Using an FRB is fairly simple in principle.

The writing side keeps the authoritative value for the write offset in thread-specific storage. The first step is then to read the other side's published read offset and to determine, from the distance between the two, the amount of free storage available. If that is sufficient, the writer stores its data into the buffer area at a position beginning at the current value of the write offset, after writing a length field, increments the local write offset by the amount of storage used, and finally writes the new value to the write offset area. If the end of the buffer area is reached, the remaining data are added in front.

The reading side first determines the distance from the read offset (kept locally) to the published write offset. If it is non-zero, the FRB contains data. The reader fetches the length field from the position indicated by the current read offset and then the following data; finally, it increases the read offset (local and published values) to point behind the data read.

Note that this method imposes an integrity condition on the FRB as a whole: the state of an FRB is only valid if, starting at the read offset and going through the length fields stored in the buffer area, we end up exactly at the write offset without bypassing it.

3 Initialization

3.1 Prerequisite Knowledge

An FRB can only be used if certain information is known to each of the two threads involved before they come into contact:

- What the rôle of this thread is, reader or writer.
- Everything needed in order to access the read or write interface, respectively, including the address and size of the memory areas and whatever is needed in order to satisfy the requirements described in section 2.1 (this includes the IU size).
- Certain conventions on how data are encoded which will be described later.

3.2 Initial State

The initial state of an FRB is often indeterminate because the hardware cannot guarantee particular initial values for storage areas and, even if it does, these values are not guaranteed to be suitable for an FRB. This means that the integrity condition described in section 2.2 is in general initially violated and this must be corrected. And even if the integrity condition holds initially, the resulting messages are fictitious or out of date and should be discarded.

Initialization of an FRB therefore requires configuring it not to contain any data by setting both offset areas as well as the underlying variables to the same value (preferably zero) before any of the two sides may proceed to using the FRB as intended for normal operations. However, the writer may not set its offset to an arbitrary value like zero if the reader is still operating on the FRB (perhaps the writer was restarted), and the reader may not do it if the writer is still using the FRB (in case the reader was restarted). This leaves us with two possibilities:

- Both sides more or less accept any values they find and try to work with them.
- Both sides synchronize initially before they are permitted to proceed independently.

The first variant requires the writer to have the ability to read its own published offset and must include rules for deriving valid values from invalid ones in both offset areas. Initialization then simply consists in both sides reading the write offset area and setting their underlying variable to the (possibly corrected) value found there; any corruption discovered by the reader in the buffer area at a later time is simply handled by reinitializing the reader. This variant is simple to implement but has the disadvantage that the writer does not obtain information on whether the reader discarded any messages; it can be considered a connection-less method of communication. In contrast, the second variant initially establishes a connection between the two sides; if the reader has any trouble, terminating the connection will make this visible to the writer. As the first variant does not seem to pose any interesting problems, the second is the one mainly considered in this article.

Under special circumstances, a synchronization may be trivial. For example, if one side of the FRB may stop the other and if this dominant side can write all memory areas, the dominant side may simply initialize the FRB while the other side is incapable of interfering. However, such a method introduces a really strong coupling between the two threads and is therefore not acceptable as a general procedure.

A better solution is to use the two offset areas for exchanging messages initially. At least some of the values chosen for this purpose must then, of course, be recognizable as being special, i.e., they must not be possible offset values. Such values are likely to be available for a number of reasons[1].

[1] Offset areas can represent values in the range 0 to $2^n - 1$ for some integer number of bits n. The critical situation is then that of the buffer area having a "size" of 2^n. The main reason for some special offset values being free is now that the offset areas are often part of the RAM chip also hosting the buffer area; this makes it likely that the entire RAM size is actually 2^n and hence the length of the buffer area will be a bit less than 2^n. (And if it isn't, we can always refrain from using a few bytes at the upper end.) We will also see later that offsets are multiples of IU chunks, hence in case of multi-byte IUs we have additional byte offset values to play with, or we may choose to count offsets in larger chunks.

In general, each of the two threads which may use a connection-oriented FRB goes through three phases:

- Not using the FRB
- Initializing
- Normal operations

The initial state for a side is one where it isn't using the FRB; it then requests the other side to participate in an initialization. The side requesting this must then wait for the other side's consent. After that has been obtained, the requesting side may set its offset to zero; this also tells the other side that this side has finished initialization. In the end we should reach a state where both offsets are zero, the reader is waiting for the writer to insert data into the buffer, and the writer has permission to write. Because the writer has the initiative during normal operations, obtaining that permission is obviously the last step to achieve. Going backwards in time, we can construct the following list of final states:

- Normal operation with an empty buffer: both offsets are zero, the reader expects the writer to insert data into the buffer, and the writer has permission to do that.
- Waiting for reader initialization: the read offset has a special value, the write offset is already zero, and the writer waits for the reader to set its offset to zero.
- Waiting for writer initialization: both offsets have special values and the reader waits for the writer to set its offset to zero.

This means the following sequence of events, now going forwards in time:

1. The reader finds a special value in the write offset area and writes the special value START_INIT to the read offset area.
2. The writer finds START_INIT in the read offset area and writes zero to the write offset area.
3. The reader finds zero in the write offset area, writes zero to the read offset area, and starts normal operation.
4. The writer finds zero in the read offset area and starts normal operation.

This sequence of steps obviously achieves the desired initialization, so the remaining question is how we can reach the state needed to start it without causing an inconsistency.

For the following discussion, let's assume that the values in the offset areas have been set deliberately and following the rules above, except that terminating the execution of the steps at an arbitrary point (e.g., because of a crash) is permitted (though undesirable) behaviour for a thread. In contrast, random initial values will be discussed in section 3.6.

As the final decision to start initialization is the reader's, the primary problem is how the reader can decide that the writer is not in a state which could lead to an inconsistency. The situation to be avoided is that the writer interprets the zero value about to be written into the read offset as a value occurring while operating normally and not as part of the initialization. Hence the reader must be satisfied that the writer is not in the state of normal operations *and will not enter it* before the reader sets the read offset to zero. The former is simple to diagnose: if the write offset area shows a special value, the writer can't be in the normal state. The latter looks trickier but it's actually not: if the write offset is a special value, the writer cannot have finished step 2 above. In the worst case (e.g., previous reader incarnation crashed during initialization and we have a very slow writer), the writer will now complete that step and then wait before step 4 for the read offset to become zero. Hence the initial condition that the write offset must have a special value ensures that the writer will not enter the phase of normal operations until the FRB is in a consistent state, even if execution of the steps overlaps partially.

Consider now the situation of the writer. Obviously, its first action must be to write a special value (I'll call it NOT_IN_NORMAL_STATE) to the write offset area. It then must examine the read offset area and wait for it to contain START_INIT. As soon as it finds that value, it can execute step 2 and continue. Note that START_INIT implies that the reader is not in the normal operating state and cannot yet have finished step 3. In the worst case the reader will now do that while the writer executes step 2. This leads to two possible outcomes: if the reader examines the write offset after the writer has set it to zero or already to some larger valid offset, the initialization will have been successful; if it examines the write offset while it still holds NOT_IN_NORMAL_STATE, the reader must leave the state of normal operations it has just entered and start a new attempt at initialization.

Obviously, both sides must agree on the encoding of offset values. This includes the bit order but also a suitable rule for recognizing the special values START_INIT and NOT_IN_NORMAL_STATE. If n is the

number of bits in the two offset areas, one possibility is to choose NOT_IN_NORMAL_STATE $= 2^n - 1$ and START_INIT $= 2^n - 2$.

3.3 Procedure for the Writer

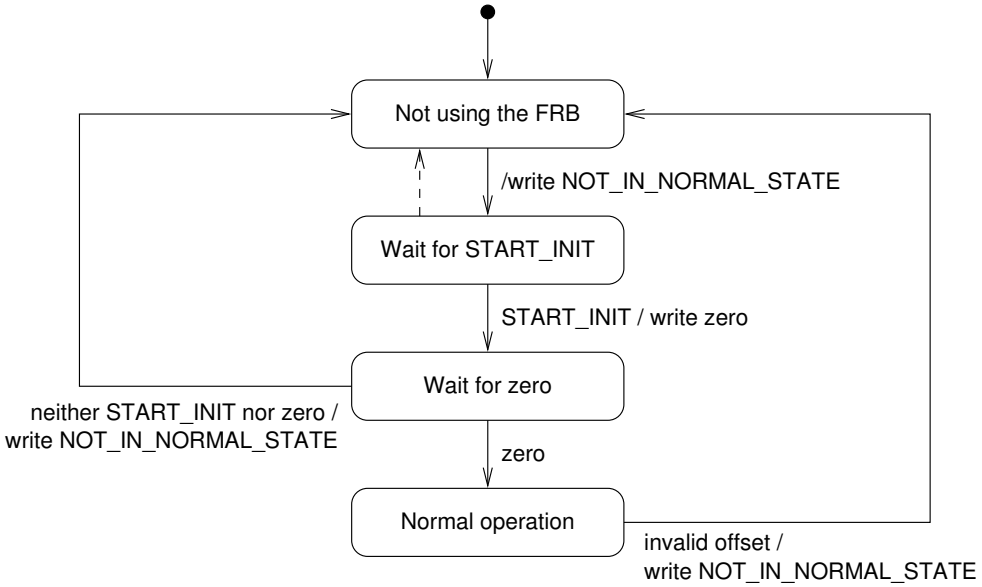

Figure 2: State transitions for the writer of a connection-oriented FRB

Figure 2 shows the initialization procedure from the point of view of the writer of a connection-oriented FRB:

1. Write NOT_IN_NORMAL_STATE to the write offset area.
2. Read the read offset area until it holds the value START_INIT.
3. Write zero to the write offset area.
4. Wait for the read offset area to contain zero. If it switches to another value, go back to step 1.
5. Start normal operations.

If during normal operations the read offset becomes invalid as a buffer offset (this includes the two special values), go back to step 1.

The writer should also execute step 1 if it stops using the FRB for whatever reason. This includes the case that the writer deliberately aborts the initialization which is always permitted (e.g., because of a timeout while waiting for the reader).

3.4 Procedure for the Reader

For the reader of a connection-oriented FRB, initialization consists of (see figure 3):

1. Write NOT_IN_NORMAL_STATE to the read offset area.
2. Wait for the write offset area to contain NOT_IN_NORMAL_STATE.
3. Write START_INIT to the read offset area.
4. Wait for the write offset area to contain zero. If it switches to another value, go back to step 1.[2]
5. Write zero to the read offset area.
6. Start normal operations.

[2]Actually, the reader might continue waiting, but this seems more robust for purely paranoid reasons.

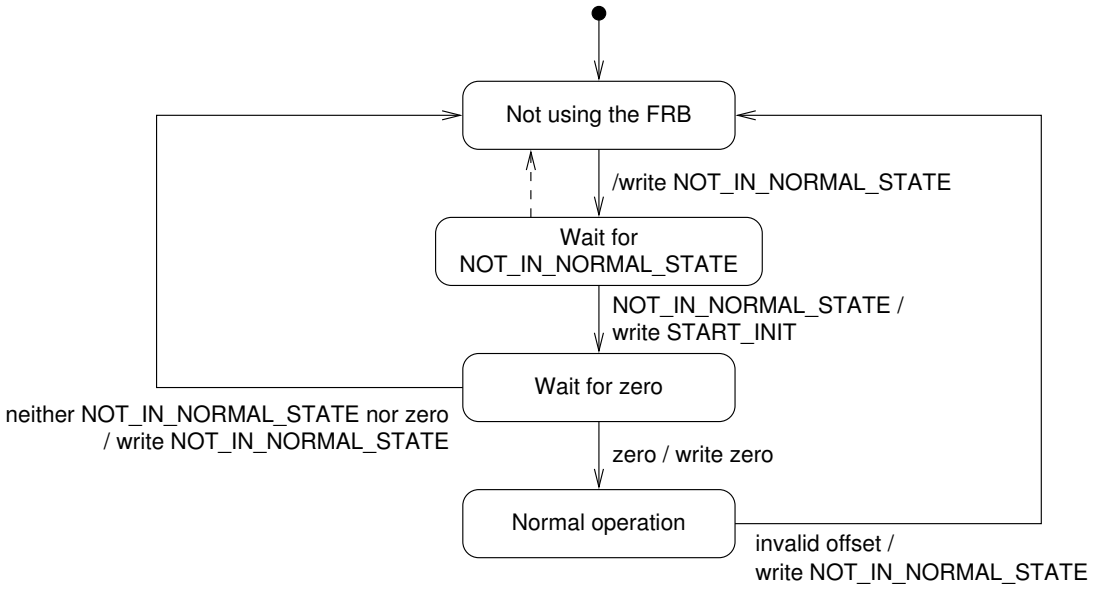

Figure 3: State transitions for the reader of a connection-oriented FRB

If during normal operations the write offset becomes invalid as a buffer offset (this includes NOT_IN_NORMAL_STATE), go back to step 1.

The reader should also execute step 1 if it stops using the FRB for whatever reason. This includes the case that the reader deliberately aborts the initialization (e.g., because of a timeout while waiting for the writer).

3.5 Assertions

In my experience, considering usage scenarios for a connection-oriented FRB can be very confusing because possibilities seem to multiply fairly quickly because of concurrency issues. The following assertions helped me in controlling these branches. They are concerned with what can be deduced about the other side's state from that other side's offset value and one's own state. The proofs for these statements are based on a number of assumptions:

- The initial memory content is undefined.
- There are no hardware failures. In particular, memory is stable, i.e., stored values change only when someone writes to the storage cell.
- Only reader and writer modify the memory areas.
- Reader and writer follow the rules in sections 3.3 and 3.4.
- Reader and writer may, however, crash. This is considered to be a transition to the state "Not using the FRB" without modifying the side's own offset.
- The fundamental operations (reading, writing, transitioning, crashing) are atomic.
- If an offset-modifying operation should be followed by a state transition, either that transition or a crash follows the operation immediately.

Assertion 1 *If the other side's offset area contains a value which is valid as a buffer area offset, that side has either not set it at all (i.e., the value is meaningless) or, immediately after it wrote the value, it either crashed (leading to "Not using the FRB") or was in one of the states "Normal operation" or "Wait for zero" (the latter only in case of the writer).*

Proof: The first possibility is obvious, hence let's assume that the offset has been set deliberately. Looking at the transition diagrams for sections 3.3 and 3.4 we see that the other half of the statement follows. —

Assertion 2 *If the other side's offset area contains a value which is not valid as a buffer area offset (this includes the special values), that side has either not set it at all (i.e., the value is meaningless) or the other side was not in the state "Normal operation" after it wrote the value.*

Proof: Again, the first half is obvious, hence let's assume that the value has been set deliberately. Disregarding crashes for the moment, consider first the situation of the writer: if it wrote an invalid offset value (which can only have been NOT_IN_NORMAL_STATE), the subsequent state was either "Not using the FRB" or "Wait for START_INIT". The conclusion is similar in case of the reader: it must have written either NOT_IN_NORMAL_STATE or START_INIT and must then have been in one of the three states which are not "Normal operation". Finally and for both sides, if a crash occurred immediately after writing, the thread was then in "Not using the FRB". —

Assertion 3 *If a side sees an offset value of zero when in the state "Wait for zero", the other side has set this value deliberately, i.e., the value isn't random. All subsequent values are then also non-random.*

Proof: Each side first waits for a special value (START_INIT or NOT_IN_NORMAL_STATE, respectively) and then for zero. Because one of the assumptions above is that memory does change only through writing, a value of zero after having seen another value implies that zero was written deliberately. This applies then to all following values as well. —

Assertion 4 *If the writer is in the state "Normal operation", the reader is not in "Wait for zero" (i.e., either in "Normal operation", "Not using the FRB", or "Wait for NOT_IN_NORMAL_STATE").*

Proof: A writer can have reached "Normal operation" only by coming from "Wait for zero", hence it saw the reader deliberately setting the read offset to zero (assertion 3). At that time, the reader must have left its "Wait for zero" to either enter "Normal operation" or to crash (leading to "Not using the FRB"). Because the write offset has been non-special since before that transition (when the offset was zero), the reader cannot since then have found the value NOT_IN_NORMAL_STATE in the write offset which would have been necessary in order to enter "Wait for zero". —

The converse of assertion 4 (if the reader is in "Wait for zero", the writer is not in "Normal operation") is the statement derived in section 3.2 to show that the reader may set its offset to zero without causing an inconsistency.

3.6 Random Initial Values

After power on, the offset areas are in an undefined state. Because both, reader and writer, wait for particular special values initially (NOT_IN_NORMAL_STATE and START_INIT, respectively), this cannot be a problem unless the initial values randomly happen to be the expected ones. Even in that case, however, no problems arise which we have not already considered because both sides have to wait for a switch to a particular other value (zero) and that will not happen randomly unless there is a hardware error. Therefore this situation is not different from finding values from a previous initialization where the other side has crashed before completing it.

3.7 Permanently Failing Initialization

We have already seen during the derivation of the preceding rules that the two sides will not reach states where they interpret the other side's offset area incorrectly. But will they always reach the state of normal operations or is it possible that initialization fails permanently or at least repeatedly?

The rules for transition between states normally guarantee that each side progresses linearly from "Not using the FRB" to "Normal operation". However, for both sides there is a branch in "Wait for zero" which takes that side back to its initial state. These branches are therefore candidate causes for a permanent failure.

Let's first try to exploit the writer's branch, i.e., consider step 4 in section 3.3. In order for that exit to be taken, the read offset must change from START_INIT to another value which is not zero. According to the rules the only permissible value for that is NOT_IN_NORMAL_STATE, and that can only happen if the reader crashed or abandoned the initialization. Hence the scenario unfolds as follows:

- Reader and writer somehow both write NOT_IN_NORMAL_STATE. The writer then waits for START_INIT.
- The reader finds NOT_IN_NORMAL_STATE, writes START_INIT and crashes.
- The writer finds START_INIT, writes zero, and now waits for zero. (This is the true starting point for this scenario; the preceding description is merely illustrative.)
- The reader starts again, writes NOT_IN_NORMAL_STATE and waits for NOT_IN_NORMAL_-STATE. Note that from this point on the two sides have synchronized, hence we don't have to consider overlapping actions.
- The writer finds NOT_IN_NORMAL_STATE, aborts the initialization, writes NOT_IN_NORMAL_-STATE, and now waits for START_INIT.
- The reader finds NOT_IN_NORMAL_STATE and the initialization continues as desired.

Hence, if the writer's abortive branch is taken, the next attempt will be successful (unless there is another crash, of course).

Now consider the case of the reader aborting the initialization. This happens if the write offset switches from NOT_IN_NORMAL_STATE to a value other than zero. However, according to the rules, zero is the *only* permissible next value for the write offset, even if the writer crashes. Hence this branch should actually never be taken unless there is a software or hardware failure.

Therefore the two sides will always reach the state of normal operation. However, that is not quite good enough to conclude that an effectively permanent failure is impossible: there are two other paths which take each side back to square one, namely the exits caused by invalid buffer offsets in the state of normal operation. Therefore it is in principle possible that there do exist scenarios in which both sides reach the desired normal state, only for at least one of them to leave it immediately. (The attentive reader will remember that I've discussed such a possibility already in section 3.2.)

I'll first consider the case of the writer leaving the state of normal operations. Because it has reached this state, it found zero in the read offset. Therefore the reader — unless it crashed — had also reached the state of normal operations and the only rule-conforming way causing the writer to go back to its initial state is if the reader takes its exit branch. We therefore only have to consider the latter case.

Let's assume therefore that the reader has found a value in the write offset area which is invalid as an offset. Because the reader is in the state of normal operation, it must have found a valid value (including zero) before. Hence the writer must have been in one of the states "Wait for zero" or "Normal operation". Apart from a crash, it can only have left them by writing NOT_IN_NORMAL_STATE and can now at most have entered "Wait for START_INIT", waiting for the reader. If the reader now reacts by restarting the initialization, further actions on both sides will progress as intended.

Therefore, unless at least one of the participants crashes repeatedly, there will be no permanent failure.

3.8 Application Beyond FRBs

Note that the rules in the preceding sections do not use the FRB's buffer area. Therefore the handshake algorithm proposed here can also be used in situations where the values in the offset areas have another meaning than that of offsets in a buffer area.

For example, if the two sides merely wish to exchange numerical values of fixed length, the two offset areas are already sufficient for that. In that case, one should probably modify the writer's rule for leaving "Wait for zero" to any non-special value being acceptable.

4 Normal Operation

4.1 Buffer Area Content

Logically, digital memory is a sequence of bits. For the purpose of storing a number in memory, we have to indicate the starting bit, the number of bits, and the bit order within the resulting bit field. However, commonly available hardware divides memory into units having a fixed number of bits (typically multiples of 8 bits) and prefers to operate on these units. At the abstraction level provided by programming languages this is usually broken down again to units of bytes (nowadays almost always consisting of 8 bits each), we are able to address individual bytes, and each byte is already represented as a number making its bit

order irrelevant to the programmer. The three parts of information needed for identifying a number in a bit sequence therefore transform into byte address, number of bytes, and the byte order. I'm going to follow this terminology in the subsequent sections, but readers should keep in mind that this is merely a convention and does not imply any restriction on hardware properties.

4.1.1 Length Field

The basic idea for storing messages in the buffer area is to store a length field first, indicating the size of the message, followed by the actual message. Obviously, both sides must agree on the encoding for the length field, i.e.:

- size (number of bytes used)
- byte order (numerical weight of each byte in the field)
- unit used for counting
- whether the length includes the length field or not

Actually, this list is already more specific than logically necessary: if buffer space is really scarce and we mainly expect short messages, we might prefer to use a length field of variable size. Note, however, that the size of the entire buffer area is fixed in advance, hence we don't have to consider arbitrarily large messages, and it's also unlikely that memory dedicated to data interchange between just two threads has a large size to begin with. Independent of available memory, there might also be an application-level upper bound on the length of individual messages to be stored. All this makes it likely that the question of whether a variable-length encoding is worth implementing boils down to using either 1 or 2 bytes to encode a length — hardly a substantial saving of buffer space. Therefore I'm not going to discuss this possibility further.

Concerning the unit used for counting it might seem obvious that the length field should identify the number of bytes in the message. However, apart from the possibility that we might want to store only messages with lengths which are integer multiples of more than 1 byte (where we could still denote the length in bytes), there is also the situation where messages to be stored have lengths measured in arbitrary numbers of bits. Therefore the unit to be used for counting message length should be agreed upon explicitly.

Whether the value in the length field includes the field's length and its padding as well as the length of the message is mostly a matter of convention. However, only the convention of *not* including the length field in the count works in all possible cases (consider the case where the field's length is not an integer multiple of the unit used for counting the length of the message), hence that should be the preferred convention.

4.1.2 Padding Conventions

The buffer area is divided into a part reserved for access by the writer (free buffer space) and a part reserved for the reader (message entries). The values in the offset areas determine which of the two sides has the right to access a particular region of storage. In order to avoid access conflicts, the two parts must each consist of an integer number of independency units (IUs; see section 2.1). From this it follows that every single message entry (length field followed by the message itself) must also consist of an integer number of IUs. This implies that every entry must start on an IU boundary and must be padded to the end of the last IU.

In addition and depending on the properties of the two bus systems used by reader and writer to access the FRB, it may be advantageous to align length field or user data on a particular byte boundary. This implies that there may be padding at the beginning of an entry or between the length field and the data. If we assume that the alignment requirements translate to the distance from the start of the buffer area to be an integer multiple of an *alignment size*, clearly the first entry does not need initial padding. As every entry should have the same layout, all entries should then also start at a multiple of the alignment size.

These two requirements lead us to the definition of the *buffer area chunk size* as the least common multiple of IU size and alignment size. Every entry must then start at an integer multiple of the buffer area chunk size (hence all non-special offset values must be multiples of that size) and no initial padding will be needed. Padding between length field and user data is then determined by the alignment size. If one simply uses the buffer area chunk size for this purpose as well, the FRB implementation does not even need to know about IU size or alignment size.

It will also simplify buffer wrapping if the buffer area chunk size is larger than or equal to the size of the length field.

4.2 Writing

Whenever the writer of a connection-oriented FRB wishes to insert a message of length n into the FRB, the following rules must be adhered to (in the state "Normal operation"):

1. If the message is too large to be inserted independent of the current fill state of the FRB, writing is refused.
2. Read the read offset. If it is not a valid buffer offset, write NOT_IN_NORMAL_STATE to the write offset area, switch to "Not using the FRB", and possibly reenter the phase of initialization.
3. Determine the amount of free space in the FRB:

   $$\text{free} = (\text{buffer\_size} - \text{write\_offset} + \text{read\_offset}) \bmod \text{buffer\_size}$$

   If it is zero, add buffer_size. Then determine the amount of space needed,

   $$\text{needed} = \text{size\_of\_length\_field} + n + \text{padding},$$

   round it up to the next integer multiple of the buffer area chunk size, and compare: if needed $\geq$ free[3], writing is presently not possible and processing stops here.
4. Write the length field and the message itself. If the end of the buffer area is reached, add the remaining data in front.
5. Increase the write offset by the rounded-up value of "needed" (modulo buffer_size) and write it to the write offset area.

It is not necessary for the writer to regularly inspect the read offset area if no message is to be written: even if the reader wishes to reinitialize the FRB, as long as no messages are available nothing is lost.

4.3 Reading

Whenever the reader of a connection-oriented FRB wishes to check for a message and possibly extract it from the FRB while in the state "Normal operation", it must proceed as follows:

1. Read the value from the write offset area. If it is not a valid buffer offset, write NOT_IN_NORMAL_STATE to the read offset area, switch to "Not using the FRB", and possibly reenter the phase of initialization.
2. Determine

   $$\text{used} = (\text{write\_offset} - \text{read\_offset}) \bmod \text{buffer\_size},$$

   the amount of storage currently used by message entries. If it is zero, there is no message and processing stops. Otherwise, subtract the size of the length field and any padding between length field and message; call the result m. If "used" is not an integer multiple of the buffer area chunk size or m is negative, we have a protocol error: write NOT_IN_NORMAL_STATE to the read offset area, switch to "Not using the FRB", and possibly reenter the phase of initialization.
3. Read the length field, starting at the current value of the read offset. If length field values include length field size or padding, subtract these values. Lets call the result n. If $n < 0$ or $n > m$, we have another protocol error: write NOT_IN_NORMAL_STATE to the read offset area, switch to "Not using the FRB", and possibly reenter the phase of initialization.
4. Read a message of length n from the buffer entry, continuing at the beginning of the buffer area if the end is reached.
5. Increase the read offset by

   $$\text{size\_of\_length\_field} + n + \text{padding}$$

   (modulo buffer_size) and write it to the read offset area.

In order to prevent unnecessary polling and to improve latency, some method of letting the writer signal the reader about a message having been added to a previously empty FRB can be useful.

[3] Yes, $\geq$ and not $>$: we can't use the full buffer because the two offset values alone are not sufficient to distinguish between completely empty and completely full (the two offsets are equal in both cases). The usable size of the buffer area is therefore buffer_size − buffer_area_chunk_size.

Appendix

References

[1] Michael J. Miller. *Dual-Port SRAMs With Semaphore Arbitration.* Integrated Device Technology, Inc., San Jose (California, USA), March 1999. Application Note AN-14.
URL `http://www.idt.com/document/14-dual-port-w-semaphore-arbitration`

Abbreviations

FIFO	First in, first out
FRB	Flexible ring buffer
IU	Independency unit
RAM	Random-access memory

Copyright and License

Address:

Martin Lottermoser
Greifswaldstrasse 28
38124 Braunschweig
Germany

History

Version 1.6: published 2013-05-10 on my web site.

Version 1.8 (2014-06-12): introduced the connection-less variant in section 3.2 and reworded subsequent sections with respect to this distinction, replaced the expression “message chunk size” with “buffer area chunk size” and redefined it as the least common multiple of IU size and alignment size, changed the license from CC BY-ND 3.0 to CC BY-ND 4.0.