SHMStream Version 2 IPC Interface

Robert Dorn, secunet AG

April 2013

無縁

Contents

List of Figures

1 Principles of Operation

This section explains the operation of the SHMStream interface in a general way. Details of the interface are given later.

1.1 Shared Memory Interface

SHMStream is an interface for transferring fixed sized packets between different subjects which operates on a shared memory region. By this it is independent from special system-specific interfaces for data exchange.

A SHMStream interface provides a nonblocking channel between a single writer and potentially multiple independent readers[1]. At any time the writer may be replaced by a different instance which does not depend on the state of the previous writer. Assuming conforming behaviour all packets are guaranteed to arrive in the order written. Furthermore packets are only lost when there are protocol mismatches, the writer overruns the reader or the writer is reset. Readers can detect and recover from these events.

If no asynchronous notification is available, SHMStream can be used by polling, otherwise an interrupt driven operation is possible.

[1]Typically there is exactly one reader for each writer.

1.2 Speculative Access

By design SHMStream is a purely unidirectional interface without any kind of back channel. Consequently no flow control or locking can be established and the buffer may change at *any* time. Therefore only atomic accesses can be guaranteed to be internally consistent. Data which cannot be accessed atomically (e.g. the payload) has to be read speculatively and can be deemed valid only if its validity can be confirmed by a further read to an atomic control structure.

1.3 Ring Buffer

Data is transferred in packets of fixed size which are stored in a ring buffer. As no bidirectional communication is desired, the reader maintains a private read counter (`RC`). In addition to the write counter (`WC`) which is updated when a write has completed, a write start counter (`WSC`) is introduced which is increased before data is written.

1.4 Overrun Detection and Resynchronization

A read operation can start whenever the writer is ahead of the reader ($\texttt{WC} > \texttt{RC}$). *After* reading the packet(s) it must be verified that the writer did not begin overwriting the data read by checking the write start counter ($\texttt{WSC} \leq \texttt{RC} + \texttt{Elements}$). Otherwise the data read is not valid and an overrun has to be signalled to the application.

Recovery from an overrun is possible by increasing the read counter so that it references packets which are not yet overwritten (e.g. $\texttt{RC} := \texttt{WC}$).

1.5 Reset Detection

As writers may be reset without notification, readers must be able to detect a restarted writer. For this purpose an `Epoch` field is set to a unique value whenever the SHMStream interface is initialized[2].

Data from any other field is only valid if the reader can assure that the `Epoch` is not zero and did not change between accesses.

Whenever a new epoch is detected, the reader signals the change of epoch to the application and resets the read counter.

2 Data Layout

The shared memory region consists of a fixed header of 64 bytes and an array of `Elements` packets consisting of `Size` bytes each. There is no padding or alignment before or between packets. If alignment is necessary, packet sizes have to be chosen accordingly.

Any additional memory should be zero. The fields `Protocol`, `Size`, and `Elements` may only be written when the interface is inactive ($\texttt{Epoch} = 0$).

The header consists of eight 64 bit values stored in machine byte order:

[2]If the writer cannot be trusted to enforce this property, out-of-band measures have to be used for reliable detection of transitions between epochs

Field	**Offset** [Bytes]	**Size** [Bytes]	Description
`Transport`	0	8	SHMStream marker. Fixed value `16#4873_12b6_b79a_9b6d#`. May be zero when inactive.
`Epoch`	8	8	Number chosen by writer. Changed on each reset. Zero when inactive. Atomic value.
`Protocol`	16	8	Nonzero number used to detect protocol mismatches.
`Size`	24	8	Size of a packet in bytes.
`Elements`	32	8	Number of packets contained in the buffer.
`Reserved`	32	8	Zero, ignored.
`WSC`	48	8	(Write Start Counter) Increased for each packet before it is written. Initially zero. Atomic value.
`WC`	56	8	(Write Counter) Total number of packets written. Initially zero. Atomic value.

3 Writer

Note that the following pseudocode shows only a specific possibility for implementation. The reader may not rely on any details following from the pseudocode.

3.1 Initialization

Before any changes can be made it must be assured that any previous epoch is ended. This is performed by setting `Epoch` to zero.

```
Buffer.Header.Epoch      := 0;
```

It may be desired to zero the complete memory area in order to remove any potential remainders of the previous epoch.

The Header is initialized in arbitrary sequence.

```
Buffer.Header.Transport  := 16#4873_12b6_b79a_9b6d#;
Buffer.Header.Protocol   := PROTOCOL_ID;
Buffer.Header.Size       := PACKET_SIZE;
Buffer.Header.Elements   := BUFFER_ENTRIES;
Buffer.Header.WSC        := 0;
Buffer.Header.WC         := 0;
```

Finally a new epoch is initiated. The method to determine the number of the `Epoch` field is platform dependent.

```
Buffer.Header.Epoch      := Get_Epoch;
```

3.2 Writing

It is assumed that the buffer has been initialized. At first the `WSC` is incremented.

```
WC                       := Buffer.Header.WC;
Position                 := WC mod Buffer.Header.Elements;
WC                       := WC + 1;
Buffer.Header.WSC        := WC;
```

The increased `WSC` has invalidated the previous data element at `Position` which can now be overwritten.

```
Buffer.Data (Position) := Packet_To_Be_Written;
```

The new data has been written and can now be made visible by increasing `WC`.

```
Buffer.Header.WC       := WC;
```

Note that the writer may choose to write multiple packets at the same time.

3.3 Deactivation

```
Buffer.Header.Epoch    := 0;
```

4 Reader

The reader may rely on the correct behaviour of the writer for transferring packets. Should the writer violate the protocol *no* attempts of the reader to recover are necessary. If a protocol error is detected, the reader should wait for a change of epoch.

As the writer may be completely untrusted it shall nevertheless be assured that a rogue writer cannot compromise the integrity or availability of the reader. This includes:

- The reader may not read packets which are not fully contained in the shared memory area.
- The reader may not affect unrelated data.
- The reader may not block.
- The reader must resynchronize itself on a new epoch.

4.1 Synchronization

As the first step the reader has to store the epoch for later comparison.

```
Reader.Epoch := Buffer.Header.Epoch;

if Reader.Epoch = 0 then
  Result := Inactive;
  -- Try again later.
```

If the epoch is not zero the reader determines if the interface is suitable. Readers may choose to expect fixed interface parameters.

```
else
  Reader.Protocol := Buffer.Header.Protocol;
  Reader.Size     := Buffer.Header.Size;
  Reader.Elements := Buffer.Header.Elements;
  Reader.RC       := 0;

  if not Is_Valid (Reader.Protocol,
                   Reader.Size,
                   Reader.Elements,
                   MEMORY_REGION_SIZE) then
    Result := Incompatible_Interface;
    -- Wait for better times
  else
    Result := Success;
    -- Interface can be used if the epoch did
    -- not change behind our back
  end if;
end if;
```

Before the result can be trusted, the reader has to assure that the epoch did not change. Otherwise the reader must ignore the previous result and try again.

```
if Reader.Epoch /= Buffer.Header.Epoch then
  Result := Epoch_Changed;
  -- Try again later.
end if;
```

4.2 Reading

Packets can only be read after a successful synchronization. At first it must be assured that a packet is available:

```
if Reader.RC >= Buffer.Header.WC then
  -- Note: Reader.RC > Buffer.Header.WC indicates a
  --       protocol error.
  Result      := No_Data;
else
  Packet      := Buffer.Data (Reader.RC mod Reader.Elements);
```

After a packet is copied it must be assured that it was not overwritten while being read:

```
  if Buffer.Header.WSC > Reader.RC + Reader.Elements then
    Result    := Overrun_Detected;
    Reader.RC := Buffer.Header.WC; -- Recover
  else
    Result    := Success;
    Reader.RC := Reader.RC + 1; -- Increment buffer
  end if;
end if;
```

After *every* read attempt it must be assured that the epoch has not changed. Otherwise the result has to be discarded and a resynchronization is necessary.

```
if Reader.Epoch /= Buffer.Header.Epoch then
  Result := Epoch_Changed;
  -- Resynchronization necessary;
end if;
```

A Example SHMStream memory region

```
6D 9B 9A B7 B6 12 73 48 -- "SHMStream20=" marker
01 00 00 00 00 00 00 00 -- epoch 1
EF BE EF BE EF BE 00 00 -- 0xBEEFBEEFBEEF protocol
02 00 00 00 00 00 00 00 -- two bytes per packet
08 00 00 00 00 00 00 00 -- the buffer contains eight packets
00 00 00 00 00 00 00 00 -- reserved, zero
0a 00 00 00 00 00 00 00 -- ten packets written or to be written
09 00 00 00 00 00 00 00 -- nine packets written
09 09 0a 02 03 03 04 04 -- second packet is overwritten
05 05 06 06 07 07 08 08 -- packet three to nine are readable
```