ZBUS channels description,

The basic description of the execution is as follows:

• The Channel lock is acquired;

• The Channel receives the new message via direct copy (by a raw memcpy());

• The event dispatcher logic executes the listeners, sends a copy of the message to the message subscribers, and pushes the channel's reference to the subscribers' notification message queue in the same sequence they appear on the channel observers' list. The listeners can perform non-copy quick access to the constant message reference directly (via the zbus_chan_const_msg() function) since the channel is still locked;

• At last, the publishing function unlocks the channel. To illustrate the VDED execution, consider the example illustrated below. We have four threads in ascending priority S1, MS2, MS1, and T1 (the highest priority); two listeners, L1 and L2; and channel A. Supposing L1, L2, MS1, MS2, and S1 observer channel A.

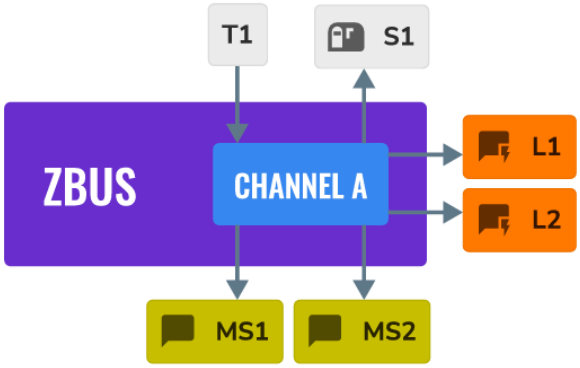


Fig. 30: ZBus VDED execution example scenario.

The following code implements channel A. Note the struct a_msg is illustrative only

```
 ZBUS_CHAN_DEFINE(a_chan, struct a_msg, NULL, NULL, ); /* Name
*/ /* Message type */ /* Validator */ /* User Data */
ZBUS_OBSERVERS(L1, L2, MS1, MS2, S1), /* observers */
ZBUS_MSG_INIT(0) /* Initial value {0} *
```

The figure above illustrates the actions performed during heVDED execution whenT1publishes to channel A. Thus, the table below describes the activities (represented by a letter) of the VDED execution. The scenario considers the following priorities: T1 > MS1 > MS2 > S1. T1 has the highest priority.

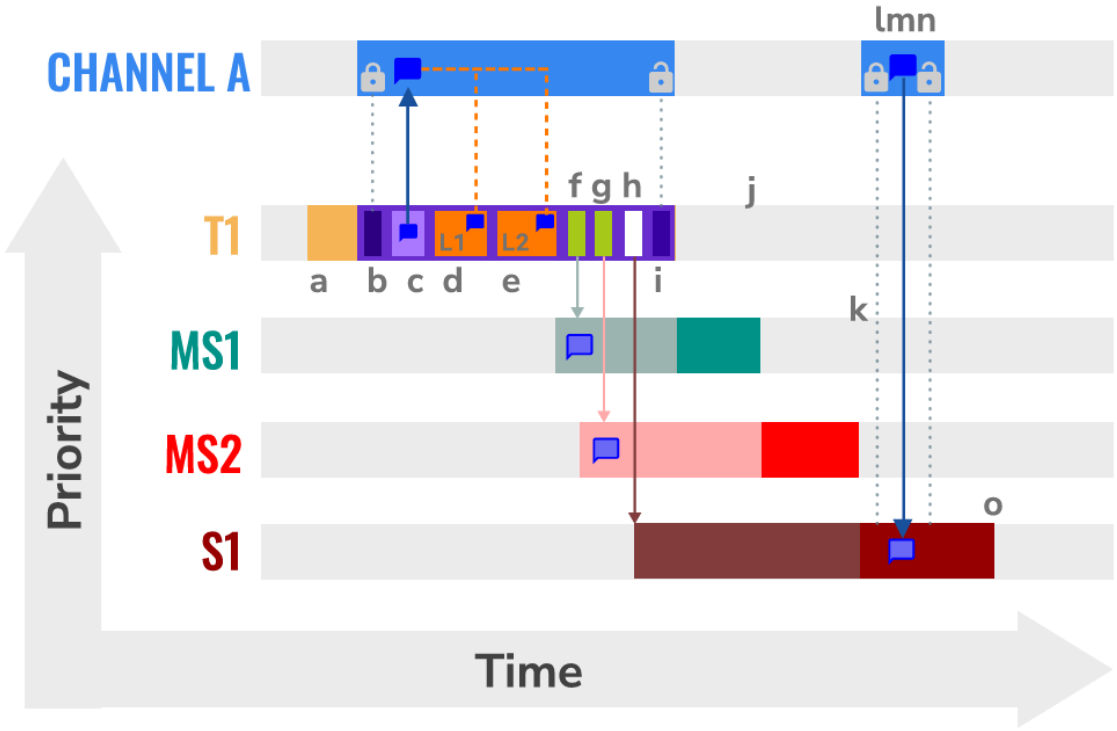


Fig. 31: ZBus VDED execution detail for priority T1 > MS1 > MS2 > S1.

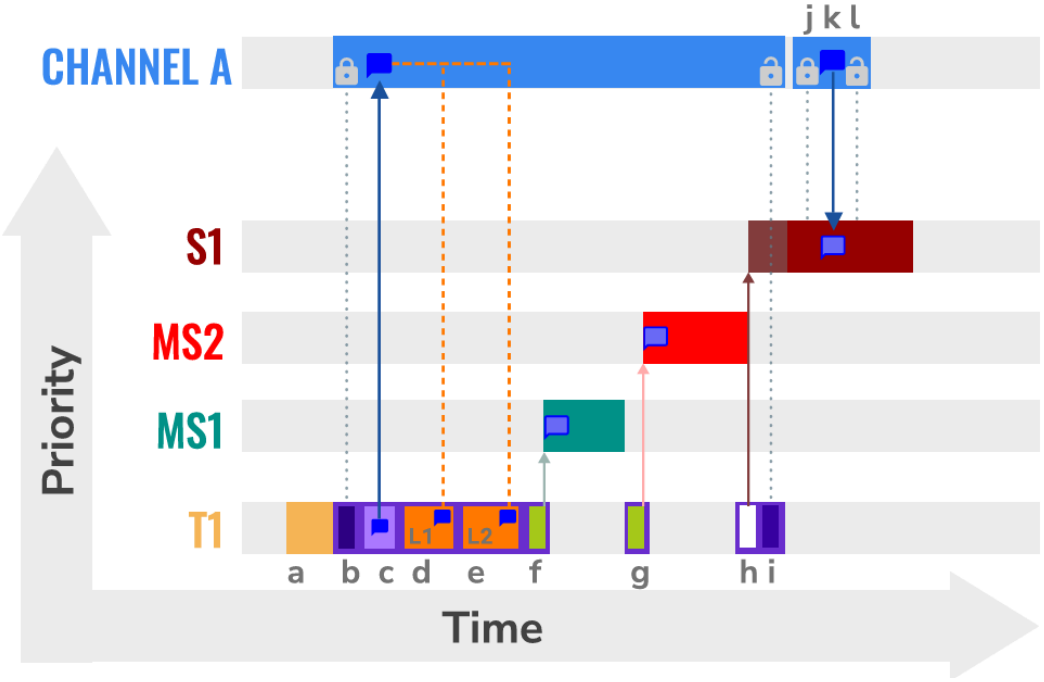
Actions Description a T1 starts and, at some point, publishes to channel A. b The publishing (VDED) process starts. The VDED locks the channel A. c The VDD copies the T1 message to the channel A message. d, e The VDEDexecutes L1 and L2 in the respective sequence.

Inside the listeners, usually, there is a call to the zbus_chan_const_msg()function, which provides a direct constant reference to channel A's message. It is quick, and no copy is needed here. f, g The VDED copies the message and sends that to MS1 and MS2 sequentially. Notice the threads get ready to execute right after receiving the notification.

However, they go to a pending state because they have less priority than T1. h The VDED pushes the notification message to the queue of S1. Notice the thread gets ready to execute right after receiving the notification. However, it goes to a pending state because it cannot access the channel since it is still locked. i

VDED finishes the publishing blockingchannel.The MS1 leaves the pending state and starts executing. j MS1 finishes execution. The MS2 leaves the pending state and starts executing. k

MS2 finishes execution. The S1 leaves the pending state and starts executing. l, m, n The S1 leaves the pending state since channel A is not locked. It gets in the CPU again and starts executing. As it did receive a notification from channel A, it performed a channel read (as simple as lock, memory copy, unlock), continued its execution and left the CPU. o S1 finishes its workload.



ThefigurebelowillustratestheactionsperformedduringtheVDEDexecutionwhenT1publishes to channel A. The scenario considers the priority boost feature and the following priorities: T1 < MS1<MS2<S1.

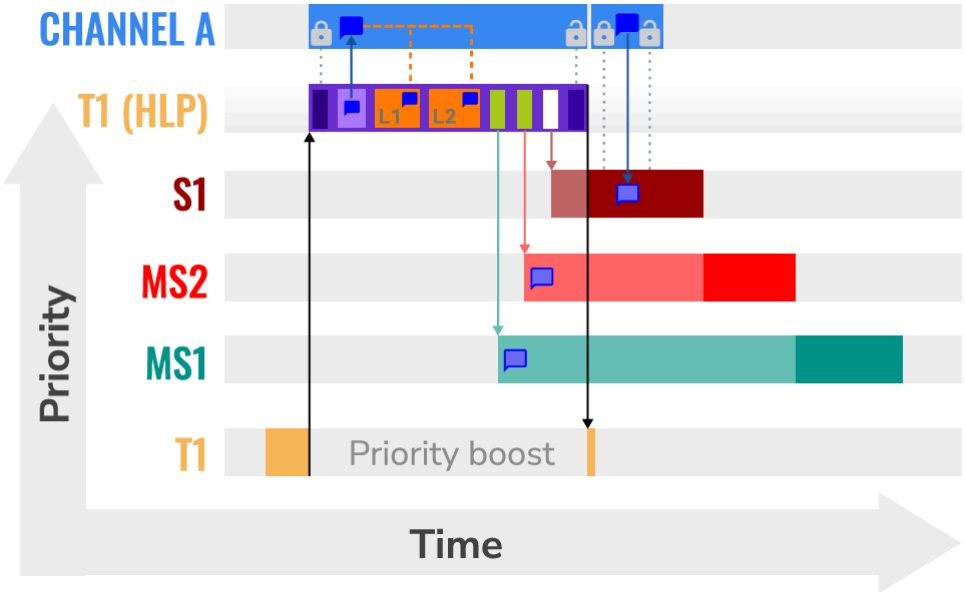


Fig. 33: ZBus VDED execution detail with priority boost enabled and for priority T1 < MS1 < MS2 < S1. To properly use the priority boost, attaching the observer to a thread is necessary. When the subscriber is attached to a thread, it assumes its priority, and the priority boost algorithm will consider the observer's priority. The following code illustrates the thread-attaching function.

```
ZBUS_SUBSCRIBER_DEFINE(s1, 4);

void s1_thread(void *ptr1, void *ptr2, void *ptr3) {
ARG_UNUSED(ptr1); ARG_UNUSED(ptr2); ARG_UNUSED(ptr3);

const struct zbus_channel *chan;
zbus_obs_attach_to_thread(&s1); while (1) { zbus_sub_wait(&s1,
&chan, K_FOREVER);
/* Subscriber implementation */

 } } K_THREAD_DEFINE(s1_id, CONFIG_MAIN_STACK_SIZE, s1_thread,
NULL, NULL, NULL, 2, 0, 0);
```

In summary, the benefits of the feature are:

• TheHLPismoreeffective for zbus than the mutexes priority inheritance;

• Noboundedpriority inversion will happen among the publisher and the observers;

• No other threads (that are not involved in the communication) with priority between T1 and S1 can preempt T1, avoiding unbounded priority inversion; • Message subscribers will wait for the VDED to finish the message delivery process. So the VDEDexecution will be faster and more consistent;

• The HLP priority is dynamic and can change in execution;

• ZBus operations can be used inside ISRs; • The priority boosting feature can be turned off, and plain semaphores can be used as the channel lock mechanism;

• The Highest Locker Protocol's major disadvantage, the Inheritance-related Priority Inversion, is acceptable in the zbus scenario since it will ensure a small bus latency

Usage ZBus operation depends on channels and observers. Therefore, it is necessary to determine its message and observers list during the channel definition. A message is a regular C struct; the observer canbeasubscriber(asynchronous), a message subscriber(asynchronous),oralistener (synchronous). The following code defines and initializes a regular channel and its dependencies. This channel exchanges accelerometer data, for example.

```
struct acc_msg {
 int x;
 int y;
 int z;
 };
 ZBUS_CHAN_DEFINE(acc_chan,
 struct acc_msg,
 NULL,
 NULL,
 ZBUS_OBSERVERS(my_listener, my_subscriber,
 my_msg_subscriber),
 ZBUS_MSG_INIT(.x = 0, .y = 0, .z = 0)
 );
 /* Name */
 /* Message type */
 /* Validator */
 /* User Data */
 /* observers */
 /* Initial value */
 void listener_callback_example(const struct zbus_channel *chan)
 {
 const struct acc_msg *acc;
 if (&acc_chan == chan) {
 acc = zbus_chan_const_msg(chan); // Direct message access
LOG_DBG("From listener-> Acc x=%d, y=%d, z=%d", acc->x, acc->y,
acc->z); } } ZBUS_LISTENER_DEFINE(my_listener,
listener_callback_example); ZBUS_LISTENER_DEFINE(my_listener2,
listener_callback_example); ZBUS_CHAN_ADD_OBS(acc_chan,
my_listener2,

3); ZBUS_SUBSCRIBER_DEFINE(my_subscriber,

4); void subscriber_task(void) { const struct zbus_channel *chan;
while (!zbus_sub_wait(&my_subscriber, &chan, K_FOREVER)) { struct
acc_msg acc = {0}; →acc.z); } } if (&acc_chan == chan) {

// Indirect message access zbus_chan_read(&acc_chan, &acc,
K_NO_WAIT); LOG_DBG("From subscriber-> Acc x=%d, y=%d, z=%d",
```

```
acc.x, acc.y,␣ } K_THREAD_DEFINE(subscriber_task_id, 512,
subscriber_task, NULL, NULL, NULL, 3, 0, 0);
ZBUS_MSG_SUBSCRIBER_DEFINE(my_msg_subscriber);

static void msg_subscriber_task(void *ptr1, void *ptr2, void *ptr3)
{ ARG_UNUSED(ptr1); ARG_UNUSED(ptr2); ARG_UNUSED(ptr3); const
struct zbus_channel *chan; struct acc_msg acc = {0};

while (!zbus_sub_wait_msg(&my_msg_subscriber, &chan, &acc,
K_FOREVER)) {

if (&acc_chan == chan) { LOG_INF("From msg subscriber-> Acc x=%d,
y=%d, z=%d", acc.x, acc.y, → acc.z); } } }
K_THREAD_DEFINE(msg_subscriber_task_id, 1024, msg_subscriber_task,
NULL, NULL, NULL, 3, 0,␣ →0);
```

```
 D: Channel list: D: 0- Channel acc_chan: D: Message size: 12 D: D:
D: Observers:- my_listener- my_subscriber D: 1- Channel
version_chan: D: Message size: 4 D: Observers: D: Observers list:
D: 0- Listener my_listener D: 1- Subscriber my_subscriber
```