Implementation of Data Distribution Service Listeners on Top of FlexRay Driver

Rim Bouhouch, Wafa Najjar, Houda Jaouani, Salem Hasnaoui
SYSCOM Laboratory
National Engineering School of Tunis
Tunis, Tunisia
{rim.bouhouch@yahoo.fr, wafa_najjar@yahoo.fr, jouani_houda@yahoo.fr, salem.hasnaoui@enit.rnu.tn}

***Abstract*-In this paper, we present a way to use Data Distribution Service Listeners implemented in the C language over FlexRay driver under µC-OSII. Our method is based on implementing all the DDS Listeners as callback operations and storing each kind of Listeners in a vector of linked list. The main goal is to create an interaction between the FlexRay Driver's Read and Write operations and the DDS Listeners to define a default communication behavior for the Interrupt Service Routine. Since every real-time network works as basis software for regular middleware such as AUTOSAR, we propose the use of the real-time middleware DDS, which has a wide range of qualities of service. Specifically, we explain in this paper our approach to implement DDS on top of FlexRay driver and its related state manager using the DDS Listeners.**

Keywords-DDS; FlexRay; Listeners; callback functions; ISR; µC-OSII.

I. INTRODUCTION

Real-Time Networks are the main field of communication systems studies since all new generations of applications not only have distribution requirements, but also are subject to deadlines. The uses of these networks vary from military to vehicular networking.

This kind of networks needs a driver to specify and control its functionality according to the normalized protocol and releases. The aim of the driver is to manage the communication system in the low communication layers regardless of the application layer. On the other hand, the OMG (Object Management Group) Data Distribution Service (DDS) provides a real-time Middleware that ensures the interaction between the physical layer and the application layer providing a communication pattern. In this paper, we will see how DDS Entities Listeners are implemented in the C language, and how to use them with the FlexRay driver according to the occurred interrupt service routine. Listeners allow communication between the middleware entities, FlexRay driver's tasks and real-time applications. We chose the C language because it is not only an embeddable language, but it is also compatible with FlexRay driver API description language under µC-OSII, the real-time OS (operating system). Also, the C language allows us to easily create the blocks Simulink of the communication model.

II. DDS OVERVIEW

DDS is a real time middleware specified by the OMG based on Subscriber/Publisher communication model [1].

The OMG DDS specification defines a data-centric communication standard for a wide variety of computing environments, ranging from small networked embedded systems up to large-scale information backbones. DDS provides a scalable, platform-independent, and location-independent middleware infrastructure to connect information producers (Publishers) with consumers (Subscribers). DDS also supports many quality-of-services (QoS) properties, such as asynchronous, loosely-coupled, time-sensitive and reliable data distribution at multiple layers (e.g., middleware, operating system, and network).

At the core of DDS is the Data-Centric Publish-Subscribe (DCPS) model, which defines standard interfaces that enable applications running on heterogeneous platforms to write/read data to/from a global data space in a net-centric system. Applications can use this global data space to share information with other applications by declaring their intent to publish data that are categorized into one or more topics of interest to participants. Similarly, applications can use this data space to access topics of interest by declaring their intent to become subscribers.

The underlying DCPS middleware propagates data samples written by publishers into the global data space, where it is disseminated to interested subscribers. The DCPS model decouples the declaration of information access intent from the information access, thereby enabling the DDS middleware to support and optimize QoS-enabled communication.

The following DDS entities (also shown in Fig. 1) are involved in creating and using a DCPS-based application:

• **Domain** – DDS applications send and receive data within a domain, which provides a virtual communication environment for participants having the same domain id. This environment also isolates participants associated with different domains, i.e., only participants within the same domain can communicate, which is useful for isolating and optimizing communication within a community that shares common interests.

• **Domain Participant** – A domain participant is an entity that represents a DDS application's participation in a domain. It serves as factory, container, and manager for the DDS entities described below.

• **Data Writer and Publisher** – Applications use data writers to publish data values to the global data space of a domain. A publisher is created by a domain participant and used as a factory to create and manage a group of data writers that publish their data in the same logical partition within the global data space. Data writers and publishers have related QoS policies that drive their behavior as DDS entities.

 ISBN: 978-1-61208-161-8

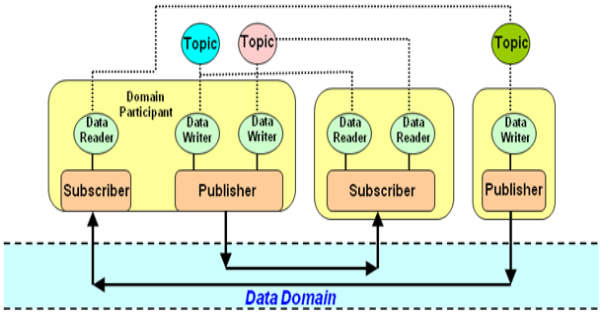

Figure 1. DDS Architecture

• **Subscriber and Data Reader** – Applications use data readers to receive data. A subscriber is created by a domain participant and used as a factory to create and manage data readers. A data reader can obtain its subscribed data via two approaches, as shown in Fig. 2: (1) listener-based, which provides an asynchronous mechanism to obtain data via callbacks in a separate thread that does not block the main application and (2) waitset-based, which provides a synchronous mechanism that blocks the application until a designated condition is met.

• **Topic** – A topic connects a data writer with a data reader, i.e., communication does not occur unless the topic published by a data writer matches a topic subscribed to by a data reader. Communication via topics is anonymous and transparent, i.e., publishers and subscribers need not be concerned with how topics are created or who is writing /-reading them since the DDS DCPS middleware manages these issues [3].

III. STATE OF ART

The OMG DDS is an API specification and an interoperability protocol that defines a data-centric publish-subscribe architecture. Since the DDS creation, the OMG has identified several implementations including RTI [6] (Real Time Innovations) Inc, which has developed the Java DDS, the PrismTech OpenSplice DDS [2] [7] that provides an academic and commercial API in the C and the C++ languages and a multitude of others companies [8] that provide a minimum profile of DDS. All these implementations are either too voluminous to be embedded or operate over a CORBA (Common Object Request Broker Architecture) platform [9]. Our approach is different, in that we associate DDS with a real-time network like CAN [10] or FlexRay.

The goal is to take advantage of the wide range of QoS available in the DDS specification and associate it with a real-time network to improve its performances, considering that DDS provides real-time QoS like Deadline or Latency Budget. But, the actual challenge of this association is how to ensure the interaction between the middleware and the real-time network, our approach is to integrate some DDS components and functionality into the network driver. The purpose is to guarantee that the driver Read and Write operations are taken in charge by the DDS Reader and Writer.

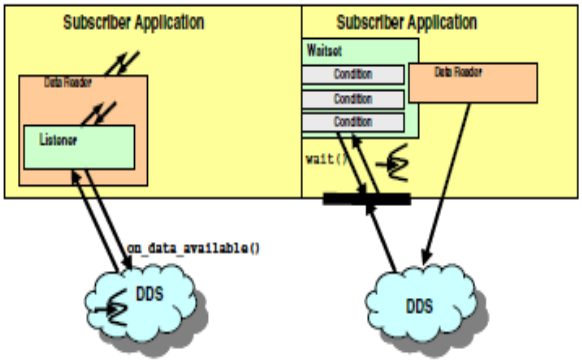

Figure 2. Listeners and wait-set Notifications[2]

Listeners and conditions (in conjunction with wait-sets) are two alternative mechanisms provided by DDS allowing the application to be made aware of changes in the communication status.

Since the condition mechanism involves the creation of a set of StatusCondition, ReadCondition and QueryCondition, we have chosen to use the Listeners mechanism to trigger the driver operations.

IV. DDS IDL TO C MAPPING RULES

In this section, we give some summarized rules to get DDS-DCPS API in the C language from its IDL specification. In fact, the OMG DDS specification is given as an idl (Interface Description Language) file, which is organized into Modules, Interfaces and operations. The Interface Description Language can be mapped into several programming languages including the C according to specific mapping rules. These rules can be applied to DDS-DCPS idl specification in order to get the correspondent C-API. The results obtained by this method have modified the naming of the DDS constants, types, entities and operations; thus each of these elements name is prefixed by the module name DDS_.

Example: the interface Entity in the idl description is mapped into DDS_Entity in the C API.

V. DDS LISTENERS PATTERN

The Listener provides a generic mechanism for the Data Distribution Service to notify the application of relevant asynchronous status change events, such as a missed deadline, violation of a DDS_QosPolicy setting... The Listener is related to changes in communication status.

Each DDS_Entity can be associated with a listener, but the implementation of these Interfaces must be done by the application. Therefore, the following Listeners are available:

- *DDS_DomainParticipantListener*
- *DDS_TopicListener*
- *DDS_PublisherListener*
- *DDS_DataWriterListener*
- *DDS_SubscriberListener*
- *DDS_DataReaderListener*

All the operations associated with each Listener must be implemented, but it is up to the application whether an operation is empty or contains some functionality [2].

 ISBN: 978-1-61208-161-8

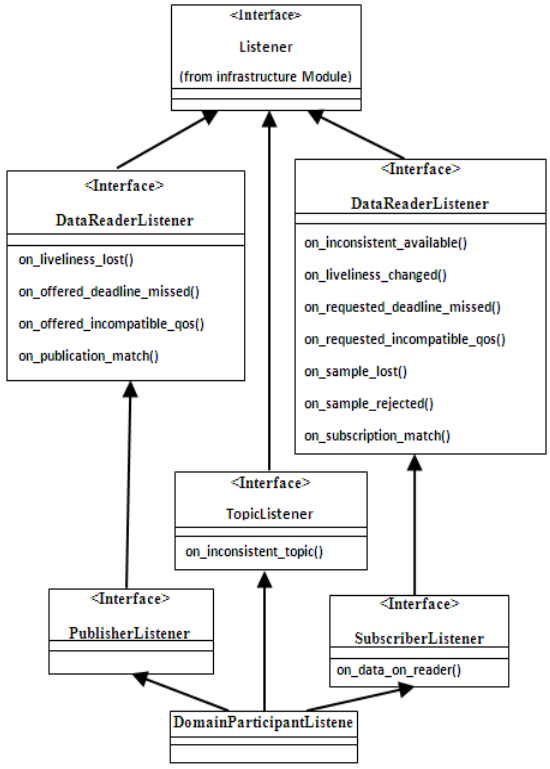

Figure 3. DDS Listeners Inheritance [4]

The structure DDS_<Entity>Listener represents the implementation of the Listener for an <Entity>. Since a Listener is implemented as a structure of pointers, the application must allocate this structure and initialize these pointers; all the function pointer attributes within the structure must be assigned to a function.

The entities listeners can inherit from each other those inheritances, shown in Fig. 3, are conformed to the classes inheritances presented in the DDS specification PIM model.

Example: *the DDS_DataReaderListener structure of Pointers:*

```
#include <dds_dcps.h>
Typedef     struct     DDS_DataReaderListener     *
DDS_DataReaderListener;
struct DDS_DataReaderListener
{
void *listener_data;
DDS_DataReaderListener_RequestedDeadlineMissedList
ener on_requested_deadline_missed;
DDS_DataReaderListener_RequestedIncompatibleQosLis
tener on_requested_incompatible_qos;
DDS_DataReaderListener_SampleRejectedListener
on_sample_rejected;
DDS_DataReaderListener_LivelinessChangedListener
on_liveliness_changed;
DDS_DataReaderListener_DataAvailableListener
on_data_available;
DDS_DataReaderListener_SubscriptionMatchListener
on_subscription_match;
DDS_DataReaderListener_SampleLostListener
on_sample_lost;
};
```

VI. USE OF CALL BACK FUNCTION TO IMPLEMENT LISTENERS

A call back function represents the storage of the address of a sequence of instructions that the execution will trigger later on a precise event.

In fact, it involves handing over to a third routine, passing to it as an argument the address of one of our duties, so it can then call it when it needs it. In the implementation, we perform the routine that we want, but what is certain is that to save the address and call a function, it takes a function pointer that is used to perform an action, which is not known at the time of writing the code (system, library…).

All the entities listeners in DDS-DCPS are implemented as callback functions so that the notification process can be event triggered. Each listener is a structure of pointers that refers to a specific communication status change. Modifying a pointer, results in the execution of the associated function representing the default behavior of DDS regarding the happening event, as shown in Fig. 4.

The Listener is invoked on the changes of communication statuses. A change of a communication status sets a status flag. The status flag is only reset when the status is being read. The Listener's operations will only be invoked on the communication statuses for which they are enabled by the mask. This invocation is based on the listener own status changes and/or on the status changes of the Listeners inherited from. Each bit in the bit-mask represents one of the statuses that can trigger the response of the Listener to the specified status change.

To access the Listeners all the entities define a generic operation and a specific subclass operation to access the class listener.

Example: *the DDS_DataReaderListener structure of Pointers implement as call back functions:*

```
#include "dds_dcps.h"
static struct DDS_DataReaderListener msgListener;
DDS_FooDataReader FooDR;
/* at this point, it is not important how to
create the FooDR*/
DataWriterListenerData UserDefined_ListenerData;
/* at this point, it is not important how
UserDefined_ListenerData is implemented. This
parameter can be used for Listener identification.
If not used, the parameter may be NULL. */
/* Prepare a listener for the Foo DataReader. */
msgListener = DDS_DataReaderListener__alloc();
msgListener.listener_data                    =
UserDefined_ListenerData;
msgListener.on_requested_deadline_missed = NULL;
msgListener.on_requested_incompatible_qos = NULL;
msgListener.on_sample_rejected = NULL;
msgListener.on_liveliness_changed = (void (*)(void
*, DDS_DataReader)) on_live_change;
msgListener.on_data_available = NULL;
msgListener.on_subscription_match = NULL;
msgListener.on_sample_lost = NULL;
/* Set the Listener with a mask only to trigger on
on_liveliness_changed. */
status   =   DDS_DataReader_set_listener   (FooDR,
&msgListener, DDS_LIVELINESS_CHANGED_STATUS);
```

This example presents the allocation and initialization of a DDS_DataReaderListener which is only enabled for the status on_liveliness_changed. The Listener msgListener will be attached to the created DDS_DataReader named FooDR. As we can see, we have associated to the pointer msgListener.on_liveliness_changed a call back function named on_live_change that will be triggered if the status is matched.

VII. FLEXRAY DRIVER UNDER µC-OSII

For our research studies, we developed a FlexRay driver under the µC-OSII Real-Time Operating System and some

Phycore PCM023 cards. We added for each a daughter card containing the Fujitsu MB88121C component. The driver behavior, as shown on Fig. 5, is based on the communication between the ISR (Interrupt Service Routine) and *FlexRayTx/ FlexRayRx* Tasks that manage the communication process. Note that the *FlexRayTx* task has the publisher as type where the *FlexRayRx* has the subscriber as type.

- **FlexRayReceiveTask ~Subscribtion task**: the *FlexRayReceiveTask* pends (reads from the mailbox) for the received message in its mailbox posted by the corresponding Interrupt Service Routine (ISR). When a data sample FlexRay Frame arrives, this task is responsible for the deserialization (extracting Frame-ID) and for storing the data in the receive queue of the *DataReader*.
- **FlexRayPublishingTask (only one by publisher):** *Every* user task calling Write () operation may use a semaphore that will lock the task when the *DataWriter*'s send queue is full. The Frames are transmitted using one of the two following modes.

Synchronous Publishing Mode: the user task invokes the *DataWriter*'s Write() operation which puts the samples (FlexRay Frame) on a separate "queue" and then calls the Write() operation within the FlexRay API Driver to put the frame in the FlexRay controller's transmit buffer before returning to the user task.

Asynchronous Publishing Mode: In this mode the Write () operation returns immediately to user task leaving the corresponding ISR to transfer frame from the queue to the controller transmit buffer. However, a flow controller (a separate task) is needed to reorganize the transmit queue depending of the FlexRay frame ID.

The TX/RX tasks react to the messages posted by the ISRs according to the related mailbox; in fact each mailbox is associated with an interruption event. Since FlexRay has two buses (channel) that can either send or receive the data frame, four mailboxes are needed to represent these communication events.

For example, if the *FlexRayRx* task receives an interruption on MailBox 2 (MB2) it knows that the related event is that the reception buffer is full and so calls the Read () function to read from the reception buffer.

The communication process using Mail Boxes under µC-OSII is driven by the OS_MBPend() and OS_MBPost() operations, as shown in Fig. 6, The ISR posts the messages on the mailboxes and the *FlexRayRx/Tx* task pends them.

The Write () and Read () operations prototypes are written as follow:

CR=Write (descriptor, @Buffer, size)
CR=Read (descriptor, @Buffer, size)

Where the descriptor indicates the channel that the event is associated to, the descriptor is known by task according to the Mailbox number (the OS_Event pointer) it refers to the occurred event that caused the interruption. The buffer address is the user buffer address where data should be written to and from. And finally, the size argument is an optional argument describing the user data size.

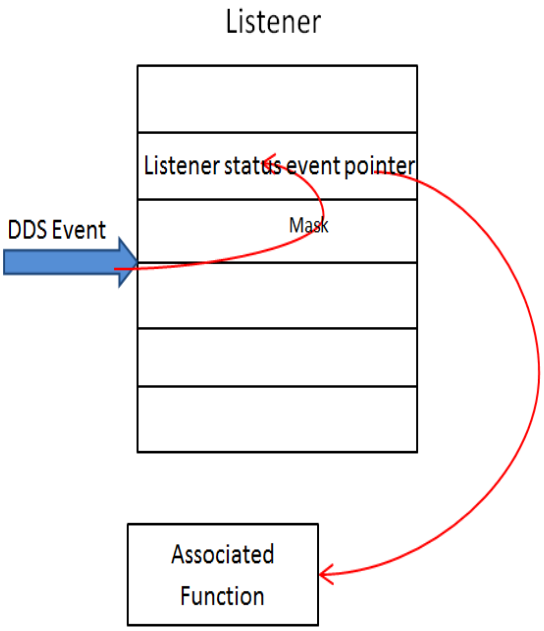

Figure 4. Listener implement as callback function

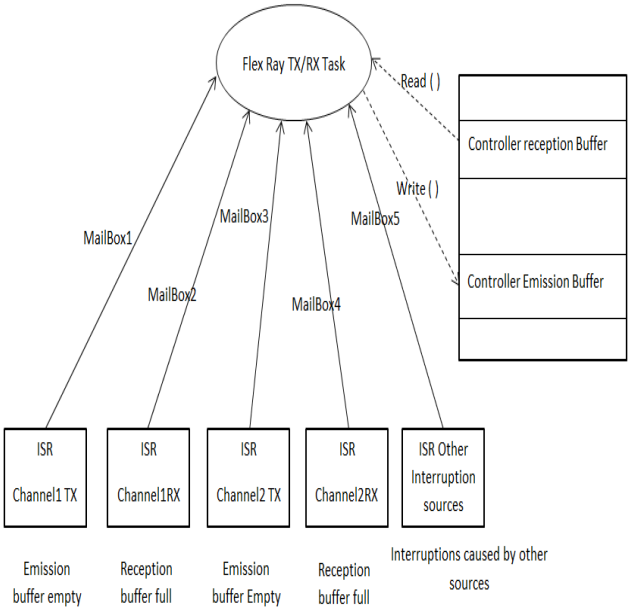

Figure 5. FlexRay Driver under µC-OSII

The returned value CR indicates whether the write/Read operation was successful or if an error occurred during the process and even the nature of the error.

If the ISR indicates the arrival of a frame, the *FlexRayRx* task will call the Read () operation, but if the ISR indicates that the emission buffer is empty the *FlexRayTx* task will call the Write () operation to write into the controller emission buffer.

According to the value of CR the *FlexRayRx* task will decide the next step to take.

 ISBN: 978-1-61208-161-8

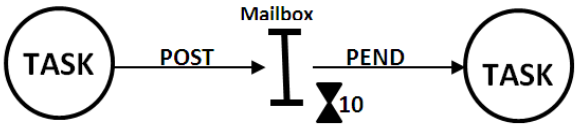

Figure 6. Message MailBox [5]

VIII. RELATIONSHIP BETWEEN DDS LISTENERS AND FLEXRAY DRIVER

After developing the FlexRay driver, we have noticed that for each kind of ISR the *FlexRayRx/Tx* task associates a default behavior, we will use DDS listeners to set default behavior for each ISR.
When an interruption occurs and the ISR sends the corresponding message, the *FlexRayRx/Tx* task will only get the ID from the frame. Since in FlexRay protocol an ID is usually associated to a data type we will assume that the flexRay ID is equivalent to the Topic Key in DDS. Therefore, getting the ID from the frame is the same as identifying the Topic.

The couple (ID, MB number) is now the unique identifier used by the *FlexRayRx/Tx* task to choose whether to call the Topic Publisher or the Subscriber. Actually, the MB number helps identifying if the event is a received data or an empty space in emission buffer, and the ID represents the Topic identifier.

A. Subscription Case

The subscription is related to the ISR event "controller's reception buffer full", in this case after identifying the event and the topic, the *FlexRayRx* task will call the appropriate *DDS_Subscriber* related to the identified Topic. In fact, it's the *DDS_SubscriberListener on_data_on_readers* operation that is called. This listener is identified by *FlexRayRx* task thanks to the pointer *listener_data*, attribute that can be used to supply the identity of the Listener.

This operation will then search for the linked list representing the *DDS_DataReaders* corresponding to the Topic. Since a *DDS_DataReaderListener* is attached to each *DDS_DataReader*, while browsing the linked list the *DDS_DataReaderListener's* operation *on_data_available* will be triggered on each listener object. The listener will have as parameter the related *DDS_DataReader* and so can call its operation *DDS_read* to get the data.

Note that if *on_data_on_readers* is called, then the middleware will not try to call *on_data_available*. However, in this case, the application will force this call and the *DDS_DataReader* objects will get data by the mean of the notification process.

Fig. 7 illustrates the whole subscription scheme representing the callback routine.

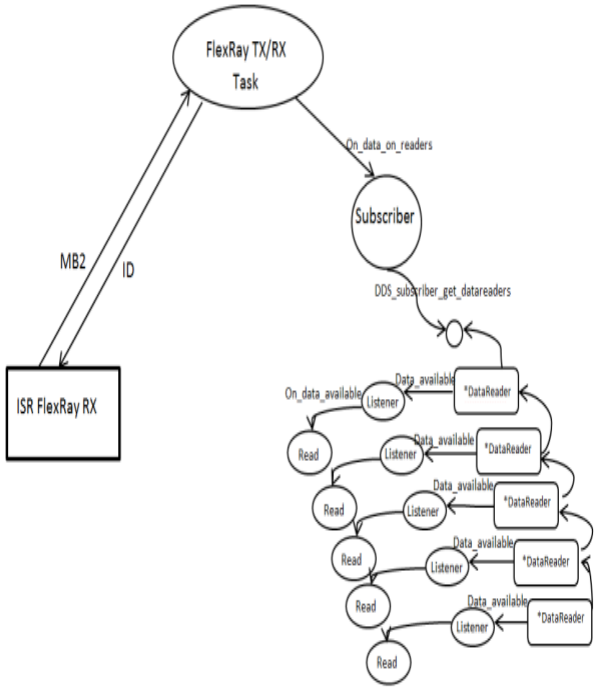

Figure 7. Subscription Routine

B. Publication Case

The publication is related to the ISR event "controller emission buffer empty", in this case after identifying the event and the topic, the *FlexRayTx* task will call the appropriate *DDS_Publisher* related to the identified Topic. In fact, it's the *DDS_PublisherListener on_publication_match* operation inherited from *DDS_DataWriterListener* that is called. This listener is identified by *FlexRayTx* task thanks to the pointer *listener_data*.

The Publisher will then search for the linked list representing the *DDS_DataWriters* corresponding to the Topic. Since a *DDS_DataWriterListener* is attached to each DDS_DataWriter, while browsing the linked list the DDS_DataWriterListener's operation *on_publication_match* will be triggered on each listener object. The listener will have as parameter the related *DDS_DataWriter* and so can call its operation *DDS_write* to write the data into the buffer.

Note the DDS specification does not set a default behavior in the Publication case, but since the use of Listener is a given option we have set our own default publication behavior matching the FlexRay Driver Needs.

Fig. 8 illustrates the whole publication scheme representing the callback routine.

IX. CONCLUSION

The Real-Time Middleware DDS offers a communication model between application level and physical layer. One of the rational uses of this middleware would be its association with a real-time network such as CAN (Control Area Network) or FlexRay Networks so that we increase the networks performances related to response time.

 ISBN: 978-1-61208-161-8

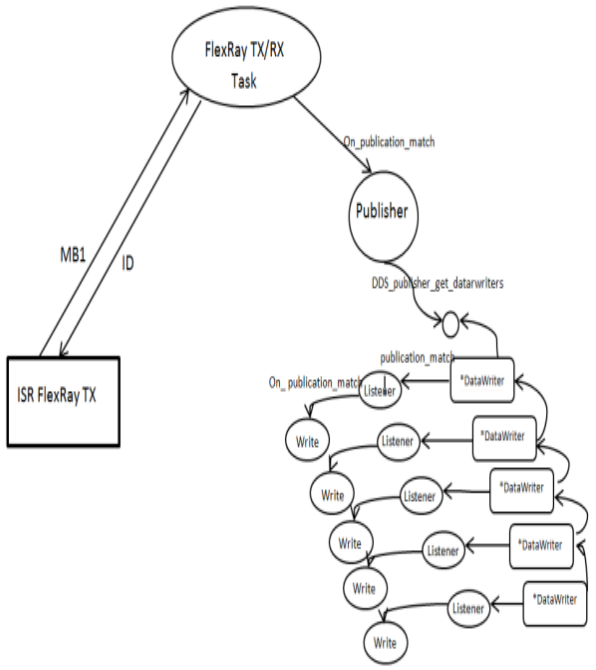

Figure 8. Publication Routine

The aim of this work is to use DDS Listeners to define a default response behavior for FlexRay ISR so that each time an interruption related to communication status occurs, a default routine is called. In this purpose we have used the DDS Listeners based on callback functions to link each ISR with the appropriate routine to replace the usual FlexRay driver's read and write operations.

The defined routine for the subscription process is indicated according to the DDS specification, but for the publication routine the DDS specification does not set a default behavior so we had to implement one of our own to match the FlexRay driver needs.

In the future works we will use this association (FlexRay Network and DDS middleware) to study its performances on a vehicle network based on the SAE (Society of Automotive Engineers) benchmark network model. This model will contain 13 nodes and applications reading and writing on the FlexRay buffers and using for the first time the DDS middleware instead of the usually used in the automobile field, the middleware AUTOSAR. We have already developed the SAE benchmark model and added the node number 13 to it, and its validation has been made using the Tasking compiler and PHYTEC XC167 cards [11].

ACKNOWLEDGMENT

The research presented in this paper would not have been possible without the support of our colleagues. We wish to express our gratitude to the SYSCOM ENIT members for their help and assistance.

REFERENCES

[1] Object Management Group- Manufacturing Domain Task, Data Acquisition from Industrial Systems specification, OMG document dtc/01-09-03, November 2002. http://www.omg.org/technology/documents/recent/omg_manufacturing.htm. 20.08.2011.

[2] Prism Tech, "Open splice C reference guide", version 2.2, Massachusetts: Burlington, 2006, pp. 22-25.

[3] N. Wang, D. Schmidt, H. Van't Hag and A. Corsaro, "Toward an adaptive data distribution service for dynamic large-scale network-centric operation and warfare (NCOW) systems", IEEE, pp. 2-3, August 2010.

[4] Object Management Group, "Data distribution service for real-time systems", version 1.2, Massachusetts: Needham, January 2007, pp. 129-130.

[5] JJ. Labrosse, "µcOS-II the real time kernel", Kansas: Lawrence, November 1998, pp. 6-7.

[6] RTI and R. Joshi, "Achitecting high performance distributed real-time applications with Java", April 2007.

[7] Prism Tech, "Open splice C++ reference guide", version 2.2, Massachusetts: Burlington, 2006.

[8] DDS vendors, http://portals.omg.org/dds/category/web-links/vendors. 20.08.2011.

[9] Open DDS, http://www.opendds.org/. 20.08.2011.

[10] T. Guesmi, R. Rekik, S. Hasnaoui, and H. Rezig, "Design and performance of DDS-based middleware for real-time control systems", IJCSNS Vol.7 No.12, December 2007, pp. 188-200.

[11] H. Jaouani, R. Bouhouch, W.Najjar, and S.Hasnaoui, "DDS on top of FlexRay vehicle network", IEEE- VCN 2011, unpublished.

 ISBN: 978-1-61208-161-8