An Ada Library for Runtime Monitoring Support in Hard Real-Time Systems

Geoffrey Nelissen, David Pereira and Andr Pedro
CISTER Research Center

September 20, 2018

1 Overview

In this document we describe the implementation of a library that provides architectural support for Runtime Monitoring (RM) in hard real-time embedded systems. The library was designed in order to be simple to understand, analyse and be instrumented into some existing Ada software development.

2 User Packages

2.1 Buffers

In the library, events are saved into buffers which have a circular behaviour. The actual buffers – implemented as fixed size arrays – is hidden in the `Buffers` package. This package is defined to be generic, with arguments being the datatype that captures the information the user is interested to associate with event, and the maximum size that we expect that the buffer has. No dynamic memory manipulation is considered in the library.

```
1 generic
2    type Event_Info is private;
3    Buffer_Length : Positive;
4 package Buffer is
5  (...)
6 end Buffer;
```

The public part of package `Buffer` declares the type of valid indexes for the underlying buffer and also a procedure that safely returns the next index where an event is to be recorded. This procedure, named `Next_Buffer_Index` is inlined for efficiency purposes. Besides this procedure, the package also provides two functions, one to add and another to subtract two value of type `Buffer_Index` with the underlying modulo arithmetic to keep the result in `Buffer_Index`'s bounds.

```
1  subtype Buffer_Index is Natural range 0..(Buffer_Length-1);
2
3  procedure Next_Buffer_Index(Index : in out Buffer_Index);
4  pragma Inline(Next_Buffer_Index);
5  function "+"(Index_1,Index_2 : in Buffer_Index) return Buffer_Index;
6  function "-"(Index_1,Index_2 : in Buffer_Index) return Buffer_Index;
```

2.2 Buffer Writers

The package `Buffer.Writer` declares only the procedure `Write`. This procedure is responsible for writing events into buffers, in the position indicated by an hidden internal state variable. The implementation of the procedure `Write` ensures exclusive access for writing into the buffer and, in this way, enforces data consistency due to the absence of deadlocks. The specification of the buffer writer package is as simple as follows:

```
1  with Ada.Real_Time; use Ada.Real_Time;
2
3  generic
4  package Buffer.Writer is
5     procedure Write(Data :in Event_Info; TimeStamp :in Time);
6  end Buffer.Writer;
```

2.3 Buffer Readers

In the library, we also provide a package that implements the operations for reading data from the buffers. Although they should not be used directly but only when implementing the monitoring function, the package is available for the user and provides three procedures that read the event that is in a given index in the buffer, gets the last written event's index and timestamp, and gets the last overridden event's index and timestamp, respectively. The specification of a reader is as described in the code listing below.

```
1  with Ada.Real_Time; use Ada.Real_Time;
2
3  generic
4  package Buffer.Reader is
5   Event_Unavailable : exception;
6
7   procedure Read_Position(Index :in Buffer_Index;
8                           Data :out Event_Info;
9                           TimeStamp :out Time);
10
11  procedure Get_Last_Written(Index :out Buffer_Index;
12                             TimeStamp :out Time);
13
14  procedure Get_Last_Overridden(Index :out Buffer_Index;
15                                TimeStamp :out Time);
16 end Buffer.Reader;
```

2.4 Monitors

We now describe the package that users of the library must use in order to write their own monitors. This package, named `Monitor`, is made of two sub-packages: the sub-package `Monitor_Instance` that specifies a monitor instance, and the sub-package `Event_Reader` that provides procedures to read data from buffers which instances of `Monitor_Instance` can use.

```
1  package Monitor is
2  private
```

```
3     Monitor_Exception : exception;
4
5     generic
6        with procedure Monitoring_Function;
7        Period : Integer;
8        Priority : Integer;
9     package Monitor_Instance is
10    private
11    (...)
12    end Monitor_Instance;
```

The second sub-package, named `Event_Reader`, declares two public procedures, namely the procedures `Pop` and `Get` that operate by obtaining an event from the circular array underlying the `Buffer` package using and instance of a reader for that array, that is the private package `Reader`. The difference between these procedures remains on the fact that `Pop` performs an update on the next timestamp that must be used to get the next event, whereas `Pop` gets also an event, but performs updates on the timestamp only in the case where the implementation detects that some error in getting the next event occurs. We will detail both these procedures soon.

```
1     generic
2        with package InputBuffer is new Buffer(<>);
3     package Event_Reader is
4
5        procedure Pop(Data :out InputBuffer.Event_Info;
6                      TimeStamp :out Time;
7                      isEmpty :out Boolean;
8                      hasGap :out Boolean);
9
10       procedure Get(Data :out InputBuffer.Event_Info;
11                     TimeStamp :out Time;
12                     isEmpty :out Boolean;
13                     hasGap :out Boolean);
14    private
15     (...)
16    end Event_Reader;
```

To use this package, the user must write a specification where the various buffers are declares. Then, for each of these declared buffers, one can build an `Event_Reader` instance to read events from those buffers, with a correct update of the the timestamp that dictates the next event to be read. The private state variables `Current_Index` maintains the index in the circular array from where the reader is reading from, and `Time_Current_Index_Save` is used to maintain (...).

We now give a detailed description on how the procedures `Pop` and `Get` work internally. In a nutshell, their goal is to return the next useful event that is saved on the target buffer, and reporting an error if such event does not exist, or that the event has some time drift according to what would be expected. Their difference resides on the fact that `Get` does not advance the reading index of the buffer, unless an error is detected (in order to avoid future redundant searches for it).

The procedures `Pop` and `Get` can alert two one of two types of errors:

- `isEmpty`, when there is no event in the buffer;
- `hasGap`, when the event that was selected is not the more precise because previous events were overwritten in the meantime.

Since these procedures are one of the most important components of the library, in the following we present the code of `Pop` and describe it in detail, for the sake of using these functionalities the most clearly possible. We start by recalling the specification of the `Pop` procedure:

```
1 procedure Pop(Data :out InputBuffer.Event_Info;
2               TimeStamp :out Time;
3               isEmpty : out Boolean;
4               hasGap : out Boolean) is
```

The procedure `Pop` takes only output parameters, namely the data in the event read, its timestamp, and two errors variables that are set to `True` if there is no event to read, or that a gat in the buffer was detected. In order to assign values to these parameters, the procedure will search for the next event following some ordering constraints that depend of the timestamps of the last overridden event, the last written one, and their indexes in the buffer. This information is declared in the declaration section of `Pop`.

Now follows the implementation of the behaviour of `Get`. First, it starts by checking if there are any events in the buffer. If there is none, the reading index `Current_Index` is updated to the position at the end of the buffer (lines 12-18). The output variable `isEmpty` is updated accordingly. This behaviour is implemented also in the `Get` procedure in order to prevent further redundant searches if the used wants to call `Get` just to consult an event, and then call `Pop` to actually retrieve the event and advance in the internal reader's pointer.

If there are events available to be retrieved from the buffer, the next step is to compare the timestamps of the last read event and the timestamp of the event being currently pointed to. If the former is smaller or equal to the latter, then we search for the first event that has timestamp larger than the last read event's timestamp.

Next, the procedure checks if the timestamp of the current index is different from the timestamp of the index of the reading position at the beginning of the the procedure's execution. If they are different, then some of the existing writers has overwritten some events, and therefore the procedure must start searching for the first event that has a timestamp larger than the one of we are searching for.

If none of the previous cases hold, then we simply get the event being currently pointer by the `Current_Index` variable, and update it to the next position. Now, the searching for the next event process is finished. We are left with updating information regarding the case when, during the search, the procedure detected that a gap in the buffer occurred.