Real-time Garbage Collection in A_2

Ulrike Glavitsch, Institute of Computer Systems, ETH Zurich, ulrike.glavitsch@inf.ethz.ch

Abstract

This document describes the design and the implementation of the real-time garbage collector for the A_2 operating system. The heap data structures were redesigned and made suitable for an interruptible garbage collector. The presented real-time garbage collector can be preempted by real-time tasks such time constraints are met. Real-time tasks must obey certain rules – they are not allowed to modify the object graph and must not use high-level locks nor wait statements. The programming language Active Oberon was extended for real-time tasks such that the aforementioned rules can be enforced.

1. Introduction

One of the main goals of the KTI project A2 was the design and implementation of a garbage collector that can be interrupted by real-time tasks. The garbage collector of the previous Active Object System (AOS) was a stop-and-go collector, i.e. the running processes on all processors were stopped, a dedicated processor performed the collection and finally, the stopped processes continued. A stop-and-go garbage collector is not suitable for general industrial applications where fast response times are required.

The implementation of the real-time garbage collector was not performed at once. The heap data structures had to be redesigned in a first step. The reason was the fact that the mark bit in the old AOS was encoded in the address field for the type descriptor. Accessing the type descriptor during garbage collection would have result in masking out the mark bit. Masking out the mark bit when accessing the type descriptor would be too costly. The new heap data structures are expressed in high-level language and also the garbage collector that accesses these data only uses very few low-level statements. By this, the whole module is easier to understand and to maintain.

The approach taken for the interruptible garbage collector is in accordance with the philosophy of the group “make it as simple as possible but not simpler” (A. Einstein). This means that real-time tasks must obey certain rules. In particular, they are not allowed to modify the object graph. Compliance with the rules is enforced by the compiler. The fact that both the compiler and the operating system is under control of the group is the key factor for such an approach. This way, the effort to implement the interruptible garbage collector was limited and it was possible to implement it within the scope of the project.

2. New heap data structures

The new heap data structures consist of a typed header block and a user block that lie adjacently to one another in memory. Such a heap block is aligned to a 32 byte boundary. Both the header and the

user block are structured similarly, i.e. they contain two fields at the beginning of the block. The first field is an address that points to its corresponding header block. This field is only set for the user block and is NIL for the header block itself. The second field is a reference to the type descriptor. The header block always has a valid type descriptor. The type descriptor of the user block can be NIL in some cases. The user entry point for the whole heap block is the address of the first user data field. The layout of a general heap data structure is shown in Fig. 1.

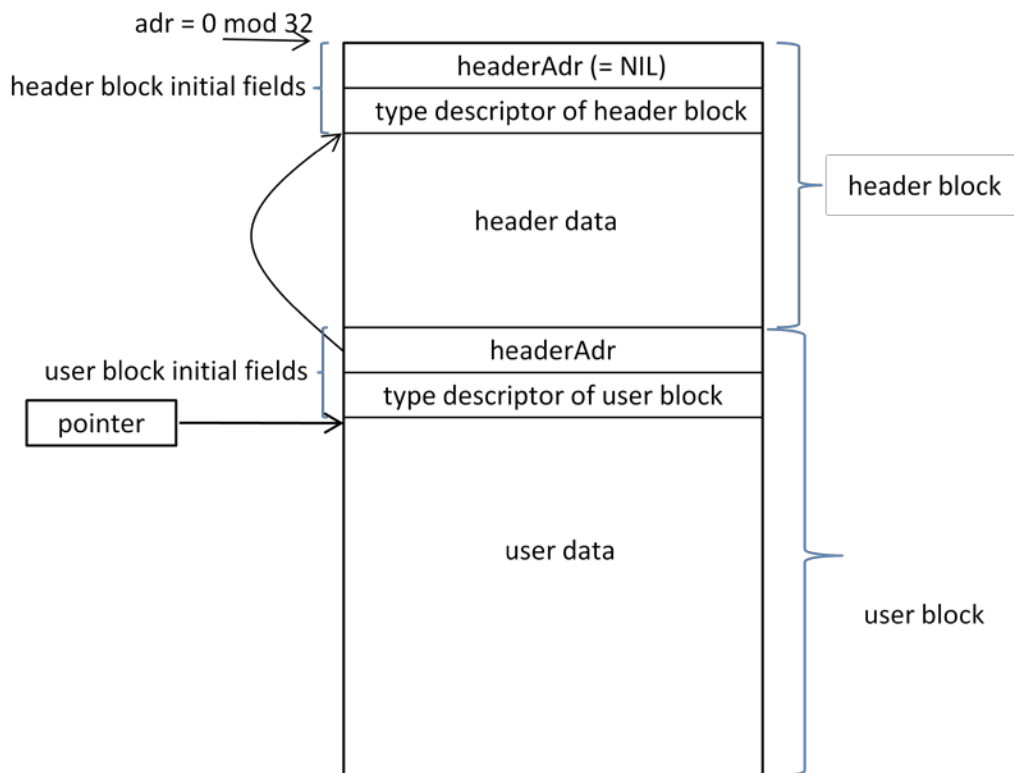

Fig. 1: Layout of general heap data structure

There is a base header block type and several extended header block types. The base header block contains the mark field, a size entry and the address of the user block. The definition of the base header block is given below.

```
TYPE
      HeapBlock = POINTER TO HeapBlockDesc;
      HeapBlockDesc = RECORD
        mark: BOOLEAN;
        size: SYSTEM.SIZE;
        dataAdr: SYSTEM.ADDRESS
      END;
```

The *mark* field is used by the garbage collector to identify a heap block as still being in use. This flag is cleared when the garbage collector is finished. The *size* field gives the size of the whole heap block. The size is rounded up to the next multiple of 32 bytes. It may happen that there are padding bytes between adjacent heap data structures. The *dataAdr* field points to the beginning of the user data. It

is not always possible to compute the beginning of the user data from the size of the header block. There can be some unused fields (padding) between the header data and the beginning of the user block in because of alignment requirements. The *dataAdr* field was introduced to allow an efficient access to the user data for all heap blocks.

There are extensions for the free block, a non-shared record block, dynamic arrays and shared objects. Dynamic arrays are distinguished into arrays with and without pointers. The declaration of these types is given in the following.

```
TYPE
      FreeBlock = POINTER TO FreeBlockDesc;
      FreeBlockDesc = RECORD (HeapBlockDesc)
        next {UNTRACED}, prev {UNTRACED}: FreeBlock
      END;

      (* non-shared object *)
      RecordBlock = POINTER TO RecordBlockDesc;
      RecordBlockDesc = RECORD (HeapBlockDesc)
      END;

      (* array without pointers *)
      SystemBlock = POINTER TO SystemBlockDesc;
      SystemBlockDesc = RECORD (HeapBlockDesc)
      END;

      (* array containing pointers *)
      ArrayBlock = POINTER TO ArrayBlockDesc;
      ArrayBlockDesc = RECORD (HeapBlockDesc)
      END;

      (* shared object *)
      ProtRecBlock = POINTER TO ProtRecBlockDesc;
      ProtRecBlock = RECORD (RecordBlockDesc)
            count: LONGINT;
            locked: BOOLEAN;
            awaitingLock: ProcessQueue;
            awaitingCond: ProcessQueue;
            lockedBy: ANY
      END;
```

The layouts of the various heap data structures are shown in Fig. 2 – 5. The layouts for non-shared and shared objects (Fig. 2 and Fig. 5) are less complicated than the layouts for arrays. For non-shared and shared objects no special alignment requirements exist. For arrays (Fig. 3 and Fig. 4), however, the first element of arrays is always aligned at an 8 byte boundary to make optimal use of IA-32 SIMD (single instruction multiple data) instructions. In addition, the pointer to an array object is aligned to an 8 byte boundary as well. Padding bytes are inserted to ensure the correct alignment whenever

needed. The type descriptor of an array that contains pointer is the type descriptor of the array element. For arrays that have no pointers the type descriptor field is empty. For arrays without pointers no type descriptor is needed since the array elements do not have any pointer offsets.

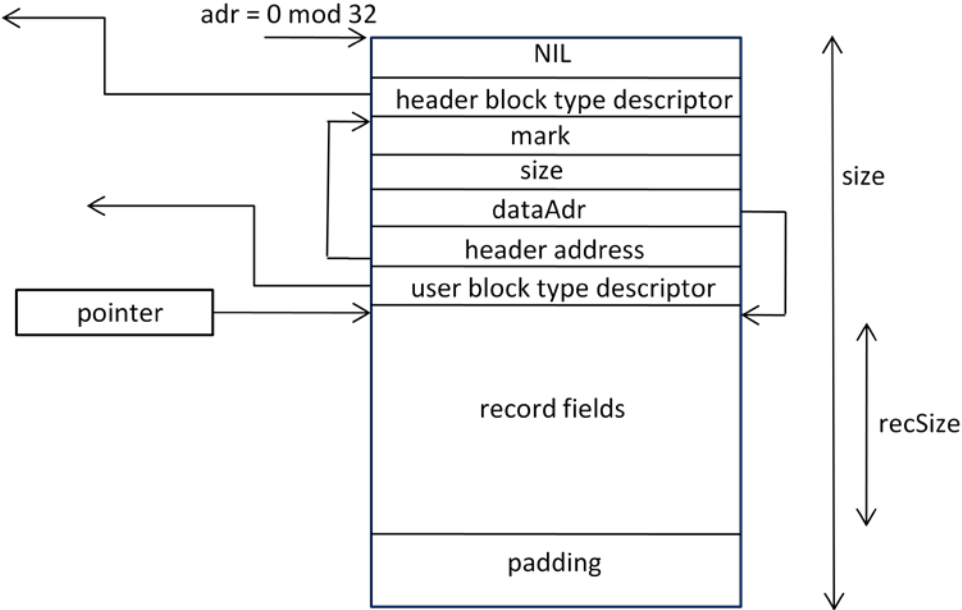

Fig. 2: Heap layout of a non-shared object

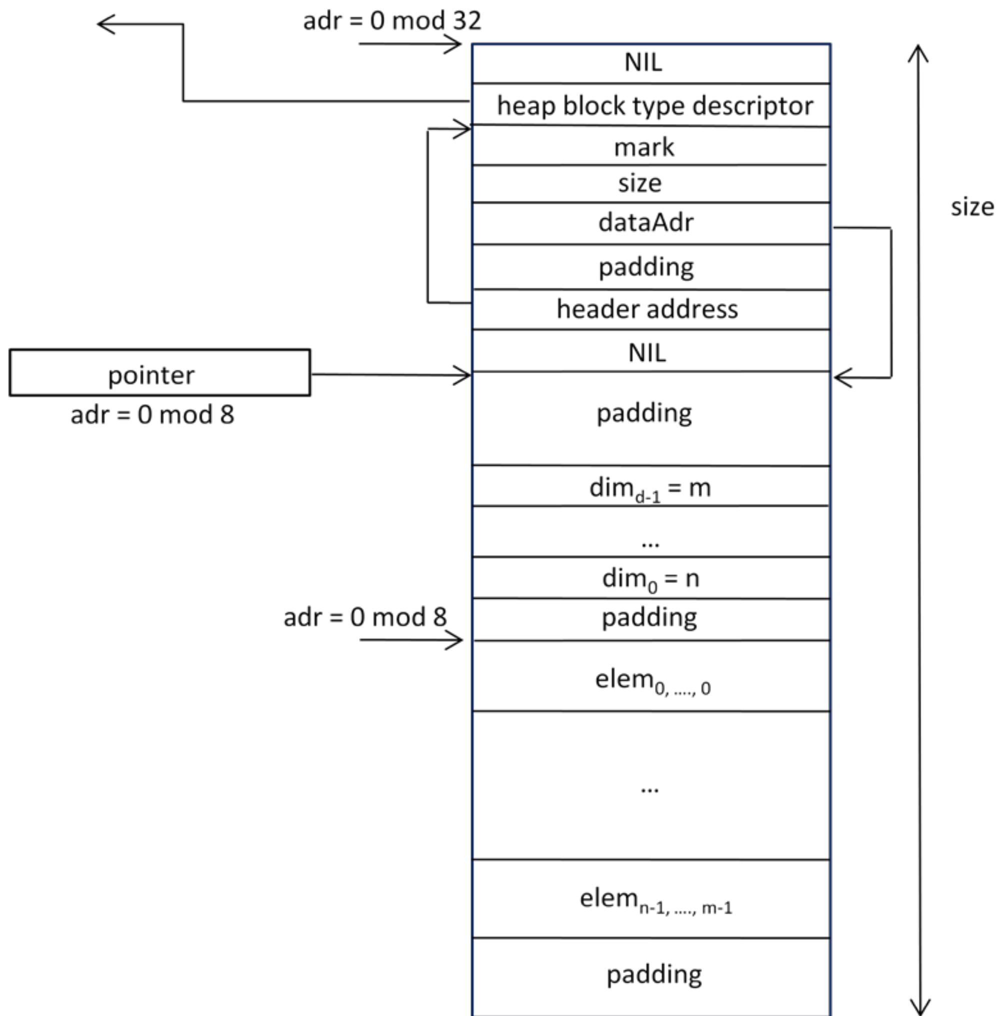

Fig. 3: Heap layout of an array that does not contain pointers

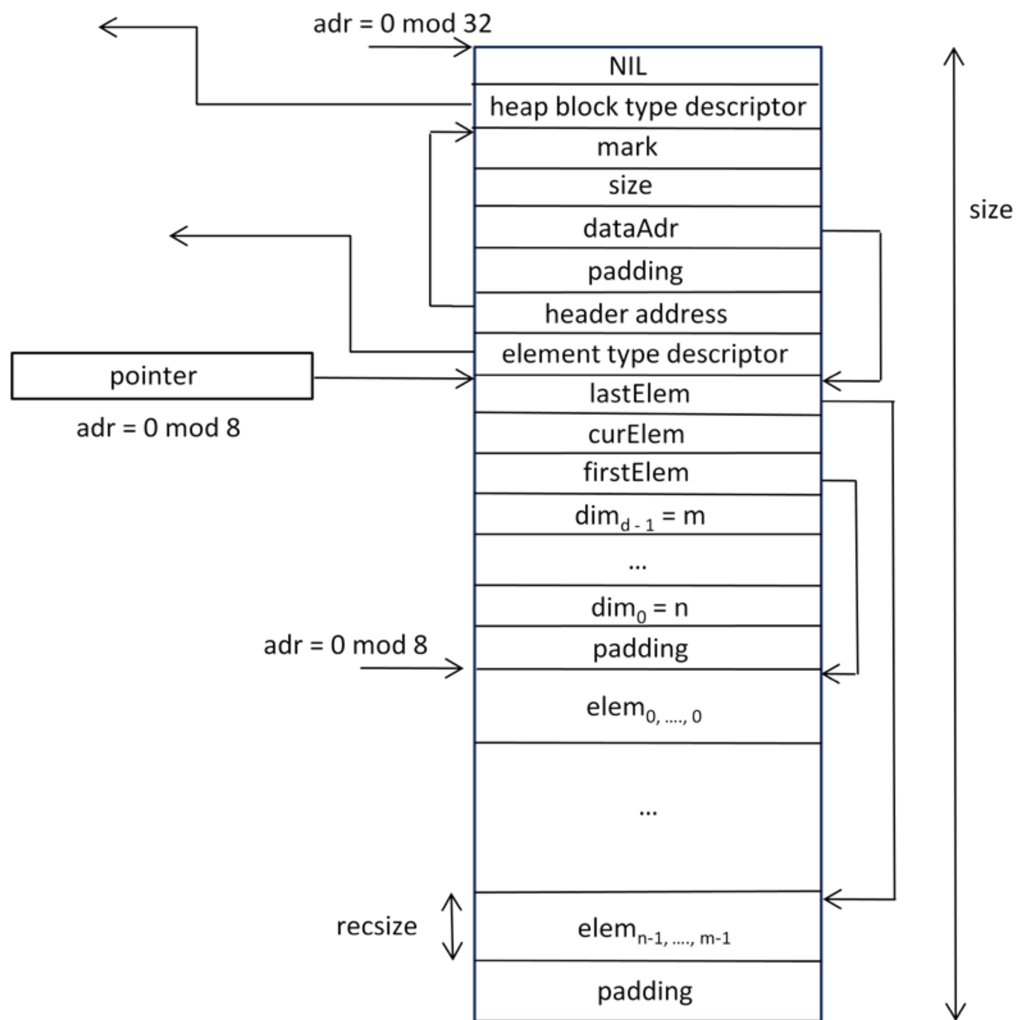

Fig. 4: Heap layout of an array that contains pointers

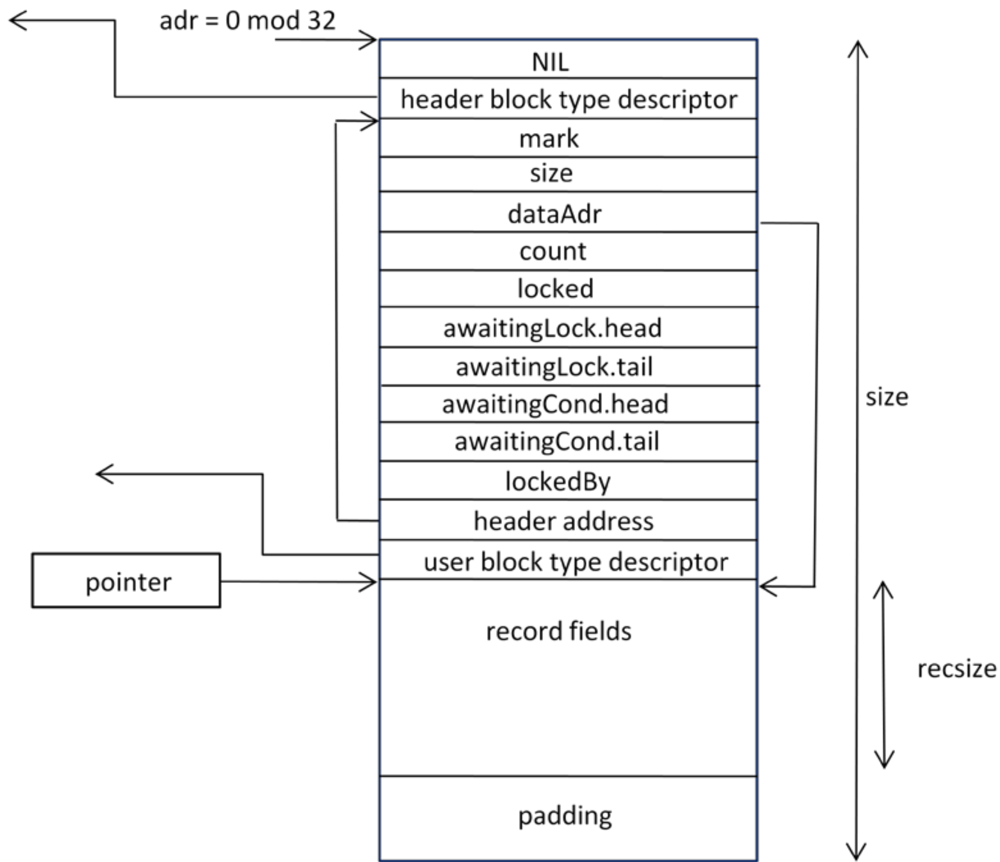

Fig. 5: Heap layout of a shared (protected) object

Type descriptors are no dynamic objects any longer. All type descriptors defined in the same module are stored in a type descriptor block that belongs to the module descriptor. The type descriptor block has no high-level language representation. It is not possible to find a high-level representation for a structure that contains a variable number of pointer offsets and a variable number of methods without compromising memory space or limiting the number of elements. Thus, a type descriptor is declared as sequence of bytes the size of which is known and that is initialized when the module is loaded. The structure of the type descriptor is shown in Fig. 6. There is a field *typeInfoAdr* in the type descriptor. It is a reference to an object that contains some additional information about the type, e.g. the module name and the type name. It is mainly used for debugging and not shown in the figure.

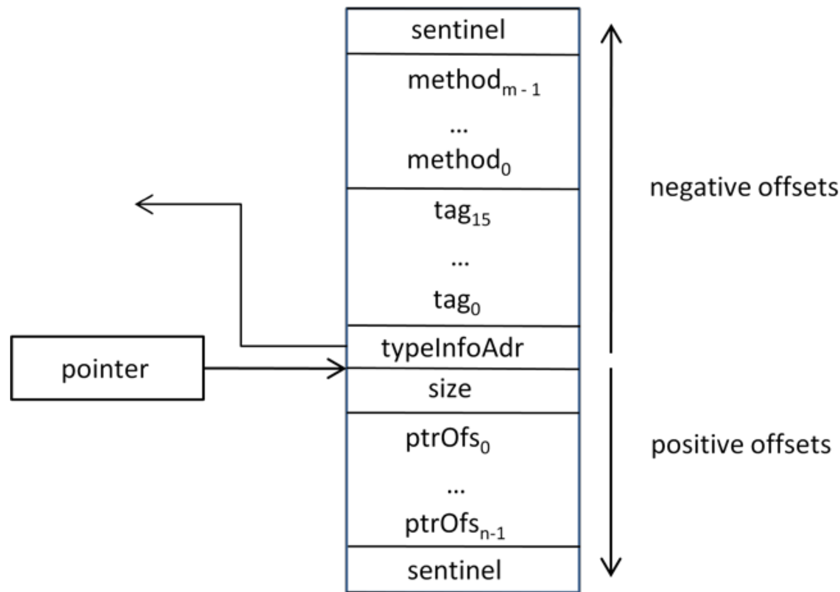

Fig. 6: Layout of type descriptor

3. Real-time tasks

Real-time tasks are a new class of active objects. They have the highest priority in the system and they may interrupt the garbage collector at any time. For a list of all the priorities in A_2, see module Objects.Mod. There are some restrictions on real-time tasks. Allocation of memory is not allowed both in the body of the active object nor in the procedures that are called in the body. This rule makes sure that the object graph is not modified when real-time tasks are running. Furthermore, sections that are marked with the EXCLUSIVE keyword and AWAIT statements are forbidden, too. The reason for this restriction is less obvious. Real-time processes that would wait on a lock or a condition could be delayed infinitely long by processes of lower priority. This is undesirable for real-time tasks. To reduce the waiting time for high-priority processes a priority inversion handling algorithm can be used. However, severe problems may occur when priority inversion handling is enabled in such a situation. The low-priority processes temporarily may get real-time priority to finish their task quickly such that a waiting real-time process can run. However, there is no guarantee that the originally low-priority process does not modify the object graph. Thus, in order to let real-time tasks run efficiently they are not allowed to use exclusive sections nor AWAIT statements.

The aforementioned restrictions are enforced by the compiler. There exists the new keyword REALTIME that must be included in the block modifiers of the active body to mark an active object as a real-time task. All procedures called in a real-time body must also include the keyword REALTIME in the declaration of the procedure. However, active bodies or procedures that are not marked as real-time may call real-time procedures. The following example code illustrates the use of the keyword REALTIME.

```
TYPE
  RTTask = OBJECT

    PROCEDURE {REALTIME} ReadSensorData();
    BEGIN
      …
    END ReadSensorData;

    PROCEDURE {REALTIME} Process();
    BEGIN
      …
    END Process;

    PROCEDURE {REALTIME} Propagate();
    BEGIN
      …
    END Propagate;

  BEGIN {ACTIVE, REALTIME}
    LOOP
      ReadSensorData();
      Process();
      Propagate();
      …
    END
  END RTTask;
```

Delegates can also be defined as real-time delegates. In this case, the keyword REALTIME has to be included into the TypeModifier. Only real-time methods can be assigned to a variable of a real-time delegate type. However, a real-time method can be assigned to a non-real-time delegate variable. The declaration and the use of real-time delegates is shown in the following.

```
MODULE M;

  TYPE
    NonRealtimeDelegate = PROCEDURE {DELEGATE};
    RealtimeDelegate = PROCEDURE {DELEGATE, REALTIME};

    O = OBJECT

      PROCEDURE P; (* no real-time method *)
      BEGIN …
      END P;

      PROCEDURE {REALTIME} Q; (* real-time method *)
      BEGIN …
      END Q;

    END O;
```

```
  VAR
    nonRealtimeDelegate: NonRealtimeDelegate;
    realtimeDelegate: RealtimeDelegate;
    o: O;

BEGIN (* module body *)
  NEW(o);
  nonRealtimeDelegate := o.P;
  nonRealtimeDelegate := o.Q;
  realtimeDelegate := o.Q;
  (* The following assignment is not allowed and causes a
     compiler error:
     realtimeDelegate := o.P
  *)
END M;
```

4. Interruptible garbage collector

The garbage collector is implemented as a separate active object that runs at priority *GCPriority* that is lower than the real-time priority. The garbage collector process is not immediately scheduled when the threshold for allocated memory is about to exceed. As a first step, a flag is set in the scheduler that garbage collection is necessary. The processes with priorities from *Low* to *High* running on the various processors are preempted. The garbage collector process is scheduled when all these processes are not running any more. Only real-time and idle processes may run when garbage collection is ongoing. In particular, there is a global variable called *lowestAllowedPriority* in module Objects.Mod. During garbage collection this variable is set to *GCPriority* indicating that only processes with priority *GCPriority* are scheduled. As soon as the garbage collector is finished the variable is set to its original value Low that indicates that processes with priorities *Low* to *Realtime* can be scheduled again.

Real-time and idle processes may be either running or are preempted waiting in the ready queues. This is valid for the time when the garbage collector runs or when it is not active. As a consequence, the ready queues cannot be marked by the new garbage collector the same way as before. An element of the ready queue may be inserted or deleted just when the garbage collector is about to mark the queue and since insertion and deletion of elements in a queue are not atomic operations a trap may occur. For this reason, the definition of the ready queue had to be adapted. The ready queue is a root object that is marked in a special way. Only the processes with priorities *Low* to *High* are marked in the ready queues by the garbage collector. Processes with priorities *MinPriority*, *GCPriority* and *Realtime* are rooted in special lists. A process of one of these priorities is inserted into the corresponding list when the process is created. These lists will not change during garbage collection and can be marked without problems.

The definition of a ready queue object is given in the following.

```
TYPE
  ReadyProcesses = OBJECT(Heaps.RootObject)
    VAR q {UNTRACED}: ARRAY NumPriorities OF ProcessQueue;

    PROCEDURE &Init;
    VAR i: LONGINT;
    BEGIN
      FOR i := 0 TO NumPriorities - 1 DO
        q[i].head := NIL; q[i].tail := NIL
      END
    END Init;

    PROCEDURE FindRoots; (* override *)
    VAR i: LONGINT;
    BEGIN
      (* only mark queues of user processes since these will not
         change during GC *)
      FOR i := Low TO High DO
        Heaps.Mark(q[i].head);
        Heaps.Mark(q[i].tail)
      END
    END FindRoots;

  END ReadyProcesses;
```

So far, there is only one real-time task in A_2. It is the real-time clock (see object *RealtimeClock* in module Objects.Mod). User can register events for this object by calling the new global procedures *SetRealtimeTimeout* and *SetRealtimeTimeoutAt* in Module Objects.Mod. Clearly, the event handlers for these events must have real-time capability. The real-time event queue is checked each time when a timer interrupt occurs and the real-time clock process is scheduled when the time for an event has expired. In contrast to the regular clock object (object Clock in Objects.Mod) the real-time clock object may run also when a garbage collection phase is ongoing. The normal clock object is only put in the ready state but not scheduled in case garbage collection is in process.

Interrupts are not switched off during garbage collection. First-level interrupt handlers must obey the restrictions for real-time tasks in order not to corrupt the system. All first-level interrupt handlers are turned into real-time procedures, i.e. they are not allowed to modify the object graph and they cannot use exclusive sections nor AWAIT statements. These restrictions on first-level interrupt handlers had consequences for some other modules. In particular, the trap handling had to be adapted. The trap handling used to write output to a trap window by using methods of objects in module Stream.Mod that are not real-time safe. The solution implemented was the introduction of new objects that have the required real-time capability and that can be used for the purpose of trap handling. Concretely, besides the objects *Writer* and *StringWriter* there are new objects *RealtimeWriter* and *RealtimeStringWriter* in module Streams. The objects *RealtimeWriter* is an extension of object *Writer* in the sense that the overridden methods of *RealtimeWriter* have real-time capability.

5. Conclusions

An implementation of a real-time garbage collector was achieved for A_2. This implementation was realized in two steps: First, the heap data structures were redesigned in order to fit an interruptible garbage collector and second, the former stop-and-go garbage collector was turned into a real-time collector. The heap data structures are expressed in high-level language and also the program code that accesses these data structures is written with only very few low-level statements. The work on the real-time garbage collector has shown that it is highly beneficial if both the compiler and the operating system can be targeted to a specific goal. An optimal solution can be strived for without making compromises in too many directions.