Australian National University

PREDICTABLE TOKEN RINGS

Week 4 Laboratory for Real-Time and Embedded Systems

Uwe R. Zimmer

Pre-Laboratory Checklist

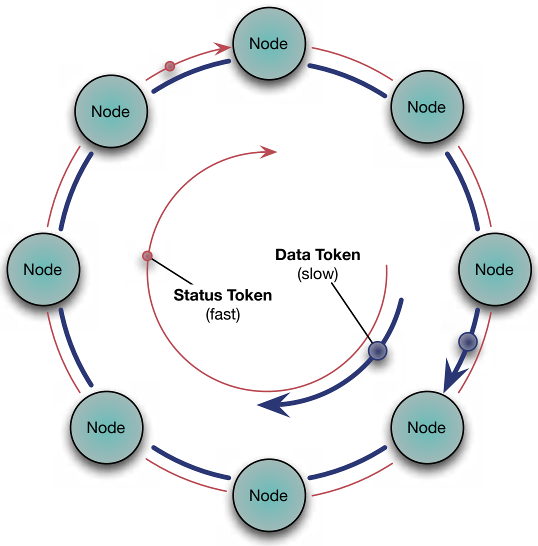
❑ **You have a working token ring structure with multiple tokens travelling independently and at different speeds from your previous lab work.**

Objectives

The objective of this lab is to familiarize you with to the concept of predictability. In order to move your previous structure closer to something which could qualify for a real-time system, we need to look into a few aspects of your precious coding in more detail.

Interlude: Predictable Token Rings

You already implemented a double token ring structure following the structure on the right. Token rings are often used in environments where predictability is essential. The first observation which you can make is that there is never a congestion on the network media (if you consider the links between the nodes as physical connections). This is different to what you possibly know from for instance Ethernet-based network systems with more than two computers on the same physical link, or wireless networks with many clients competing for access simultaneously. Yet not having to handle congestion is only one aspect of a predictable communication system. The next step is to guarantee that tokens will spend a predictable amount of time in each node. This is not a question of "how fast can you go" but a question of "how close to the predicted time-span will every node delay the token".



Exercise 1: Predictable Status Token

Assume that the computational effort to process the status token in each node is constant (and simulate this accordingly). Take measurements how long a status token stays inside a network node. You will need to explore the package `Ada.Real_Time` for this purpose. Your time measurements at the moment are still bound by the precision and time glitches of the desktop operating system on which you are running these experiments - so don't expect clock-cycle precise readings at the moment. The goal of this first exercise is though that the status token delay should not be influenced by the processing of the data token. Proof this by varying the processing times which you simulate for the data token and show that your time readings for the processing of the status token does not change.

Assigning priorities

You may also want to start experimenting with Priorities in your design. Priorities can be assigned statically by adding a `pragma Priority (<expression>)` inside a task definition (usually right after the line with the `task` name, but always before the according `end` statement). Thus a very simple priority assignment could look like this:

```
with System; use System;
(…)
task Not_so_important is
   pragma Priority (Priority'First);
   (…) -- entries for this task come here
end Not_so_important;
(…)
task Pretty_important is
   pragma Priority (Priority'Last);
   (…) -- entries for this task come here
end Pretty_important;
(…)
```

Priorities can also be dynamic (changed at run-time). This is achieved with the methods defined in `Ada.Dynamic_Priorities`. The package with its two, dead-giveaway-named procedures `Set_Priority` and `Get_Priority` speaks for itself. (If you have not yet found out how to look up standard packages, check Help -> GNAT Runtime inside your GPS environment.) Note the default task assignment, which means that if you do not specify which task's priority you want to chance, you will change the priority of the task this call is being executed in.

Don't be too worked up by the confusing concept that you can change priorities in a '*Fixed* Priority Scheduling' environment. In cases where priorities can change at runtime, the method will be referred to as 'Priority Scheduling'. Dynamic priorities can be used to emulate any scheduling method (e.g. if your environment has not already defined the scheduler which you need), as they give you full control over the scheduling process. Certification processes usually ban its usage though, and in fact dynamic priorities are forbidden in Ada-pre-defined, high-integrity language subsets. In your assignment you will work close-to-hardware and using the Ada Ravenscar language profile (targeting high-integrity, real-time systems) which bans any dynamic priorities.

At the moment you control options over your system are still limited by the underlying desktop operating system and priorities are commonly disrespected there in two ways:

a. Priorities are often interpreted in "bands", which means that changing priorities by a single discrete step has often no effect.

b. Priorities are often taken as hints rather than a mandate, which means that higher priority tasks will usually gain access to the CPU more frequently, but cannot monopolize the CPU.

Both of these common effects vary widely between different operating systems and also between different versions of the same desktop operating system. So don't put your expectation up too high about the control which you have on your current setup. You will be provided with fully controllable hardware later in the course.

Hanging out for a bit

Tasks can also put themselves to sleep for a defined amount of time or until a specific absolute time. In high-integrity systems only the latter will be allowed, so don't become too attached to the former. The Ada syntax for those statements is `delay <relative-time-expression>` and `delay until <absolute-time-expression>` respectively. As you can judge from the course title, you will be using those time related statements a lot later and in all possible contexts, so don't spend too much time on those here, but rather focus on basic scheduling first. If you are already curious, then have a look at the package `Ada.Real_Time`. The `<absolute-time-expression>` for the `delay until` statement is of type `Time` as defined there. Note that the `<relative-time-expression>` is of type `Duration` (a subtype of `Float`) and is not defined in `Ada.Real_Time`, yet conversion routines between the real-time-type `Time_Span` and the general type `Duration` are provided there. Again, do not get too attached to the relative delay statement as it is also banned in the Ravenscar language profile which you will be using in your assignment. As we will learn, relative delay statements introduce drift-effect which are usually highly unwelcome in real-time systems.

Switching between tasks on the same priority

Preempting tasks is a complex operation which should only be applied on a necessity basis in a real-time system. In other words: the concept of "provide all tasks with a fair share of CPU time just because if seems more fair" does not apply. Tasks will be only preempted based on priorities.

Yet sometimes tasks themselves will know that they reached a lesser critical part of their computation, and that it would be "all right" now to release the CPU if it could be made use of elsewhere in the system. This leads ultimately to "cooperative scheduling" (a scheduling method used in some high-integrity systems) which we will discuss later in the course as well.

A task may indicate that it is willing to be suspended if the CPU can be deployed for another, same-priority task at the moment by the statement `delay 0.0`, or slightly less clumsy by calling `Yield` from the package `Ada.Dispatching`. If there are currently more (or an equal number of) CPU cores available then runnable, same-priority tasks, those calls have no effect. Otherwise the current CPU core will be handed over to the next (in a first-in-first-out fashion) runnable task on the same priority.

Experiment with the above

Remind yourself what the goal of this first exercise is and provide evidence to us that your status token stays in each Node for a close-to-constant amount of time. Do this by submitting your Token_Rings.zip file on the *SubmissionApp* under "Lab 4 Predictable Status Tokens" for code inspection by your colleagues and by us.

Exercise 2: Fully predictable token ring

This is an advanced exercise which is only recommended for students who want to dig deeper and want to explore the full options of how to implement such a network interface under real-time constraints.

This exercise looks a little further ahead into the course and you will likely need to check out asynchronous transfer of control statements to achieve satisfactory results here.

So far we did not assume anything about the processing times of the data token. We will keep this assumption of a potentially varying and unpredictable time-span up, but require the processing in each node to stick to predicted values regardless. This can be achieved by monitoring the processing times for the data token which it is being processed and by potentially stopping computations which would be violating deadlines. At the same time you may also want to pad in additional delays for the case that the processing completed before the intended predictable handling time.

To make it more convincing simulate the heavy processing inside the data processing task with some CPU-cycle-consuming number crunching loop with some wide variations is duration instead of a delay statement. This will require your code to actually stop a running job in mid-air rather than shorten the delay-time of a blocked task.

This should result in your structure to become fully predictable even in the presence of a computational task which cannot be predicted.

Submit your Token_Rings.zip file on the *SubmissionApp* under "Lab 4 Predictable Token Ring" for code inspection by your colleagues and by us.

**MAKE SURE YOU LOGOUT
TO TERMINATE YOUR SESSION!**