# Experience in programming device drivers with the Ravenscar profile

***Jorge López, Ángel Esquinas, Juan Zamorano, Juan Antonio de la Puente***
*Universidad Politécnica de Madrid. ETSIT UPM. E 28040 Madrid. Spain*

## Abstract

*The Ravenscar profile defines a subset of Ada tasking that can be statically analysable for real-time properties. The implications of the Ravenscar profile and other commonly used high-integrity restrictions for developing device drivers are analysed in the paper, and some guidelines are provided based on the analysis. The technical content of the paper is based on the authors' experience in developing communication drivers for the Open Ravenscar real-time Kernel (ORK) that are well suited for space on-board applications. A reference architecture for device drivers is proposed, and two instances of drivers based on it are described.*

*Keywords: Ada 2005, real-time systems, Ravenscar profile, device drivers, low-level programming.*

## 1 Introduction

The Ravenscar profile [5] defines a subset of Ada tasking that can be used to develop real-time systems with predictable, analysable temporal behaviour. It is aimed at high-integrity applications that can eventually undergo a certification process with respect to some domain-specific standard. The profile has been widely accepted in academy and industry, and a number of industrial-grade implementations are available which can be used to develop highly critical systems. All of them include cross-compilation chain and a runtime system supporting the static tasking model defined by the profile.

A Ravenscar runtime system typically includes a tasking kernel, as well as some basic device drivers, e.g. for one or more system clocks and a serial line for debugging purposes. However, embedded real-time systems usually include specific hardware devices for which appropriate drivers have to be developed, often as part of an application development process. Driver programming requires accessing device controller registers, even at the bit level, and synchronizing the I/O operations with the CPU, usually by means of interrupts [7]. The Ada language [12] includes a number of low-level constructs for this purpose, among which the main ones are:

- *Representation clauses* can be used to represent hardware registers, including bit-wise structures and the addresses where they are located, by means of data types, data objects, and type and object attributes.
- *Protected procedures* can be used as interrupt handlers. The enclosing protected objects may include data objects and other protected operations that can be used by the application tasks to interact with the device.

Other useful low-level elements of the Ada language include storage address handling, machine code insertions, and shared variable control pragmas.

The Ravenscar profile explicitly allows most of the above features, provided they are used in such a way that the structure of the program remains static (e.g. dynamic handler attachment is forbidden). However, the profile excludes some useful elements that are part of common programming patterns for drivers, such as multiple entries in protected entries and requeue statements (see e.g. [5]). Furthermore, since the profile only addresses the tasking aspects of the language, additional restrictions are usually set on sequential constructs in order to enforce temporal predictability and support different kinds of static analysis in high-integrity applications [14]. Such restrictions may limit the use of some elements that are commonly used in device drivers (e.g. access types). Therefore, developing device drivers for high-integrity real-time systems may require additional effort from the programmer in order to overcome the restrictions in the expressive power of the language that are imposed in order to comply with the predictability and reliability properties required from such systems.

In the next section the overall difficulties in developing device drivers for high-integrity systems are analysed, and some general guidelines are proposed. Section 3 describes the authors' experience in developing drivers for a family of on-board embedded computers in the space domain, and a software architecture for device drivers is proposed. Two instances of communication drivers derived from this architecture are described in section 4. Finally, conclusions of the work and future work plans are explained in section 5.

## 2 Device drivers and high-integrity restrictions

### 2.1 Ravenscar restrictions

The Ravenscar profile is defined by three pragmas and a set of restrictions [12]. The pragmas specify the dispatching and locking policies to be FIFO_Within_Priorities and Ceiling_Locking, respectively, and require potentially blocking operations within protected operations to be detected. The restrictions define a static, analysable tasking model [13].

The profile explicitly allows protected procedure interrupt handlers to be declared using pragma Attach_Handler, forbidding only the use of the dynamic attachment features defined in the Ada.Interrupts package. Therefore, the basic language elements for programming device drivers are available in Ravenscar programs, and most of the Ravenscar restrictions raise no problems in this respect.

However, some common programming patterns for drivers, such as the simple two-step pattern shown in listing 1 are not allowed by the profile. This pattern uses two features forbidden by the profile, multiple protected entries and requeue, to perform an input-output operation in two steps: first Start_IO is called to setup the device registers as needed. Then the call is requeued to End_IO, awaiting the completion of the operation to be signalled by an interrupt. When the interrupt arrives, the handler opens the End_IO barrier and the operation completes. Notice that End_IO is private as it is only invoked by the requeue statement in Start_IO and thus cannot be called by other program units.

This pattern can easily be transformed into one that does not make use of requeue, as shown in listing 2, provided that the driver is only used by one application task. In this case the task can call Start_IO as a procedure and consequently exit the protected object. A second explicit call to entry End_IO has to be made in order to await the arrival of the interrupt, which is handled as before.

Notice that allowing only one application task to use a device driver is quite natural in Ravenscar programs. Otherwise, two tasks might be queuing on the protected entry of the drive, which is forbidden by the profile.

## 2.2 Other high-integrity restrictions

In addition to the Ravenscar tasking restrictions, restrictions on the sequential part of the language are often enforced on high-integrity systems, in order to enhance their robustness and predictability and enable advanced verification techniques to be used [14]. Some common restrictions are:

- No_Allocators
- No_Unchecked_Access
- No_Dispatch
- No_Recursion
- No_IO
- No_Exceptions
- No_Access_Subprograms

Most of the restrictions in the above list do not raise any special problem for programming drivers. The last two ones, however, deserve some further attention. No_Exceptions prevents exceptions to be used to handle

**Listing 1 Two-step driver**

```
protected Driver is
  entry Start_IO;
private
  entry End_IO;
  procedure Handler;
  pragma Attach_Handler(Int_ID, Handler);
  Ready : Boolean := True;
  Finished : Boolean := False;
end Driver;
protected body Driver is
   entry Start_IO when Ready is
   begin
      ...
      Ready := False; Finished := False;
      requeue Complete_IO;
   end Start_IO;
  entry End_IO when Finished is
  begin
    ...
    Ready := True;
  end End_IO;
  procedure Handler is
  begin
    ...
    Finished := True;
   end Handler;
end Driver;
```

**Listing 2 Ravenscar-compliant two-step driver**

```
protected Driver is
  procedure Start_IO;
  entry End_IO;
private
  procedure Handler;
  pragma Attach_Handler(Int_ID, Handler);
  Finished : Boolean := False;
end Driver;
protected body Driver is
   procedure Start_IO is
   begin
     ...
     Finished := False;
   end Start_IO;
  entry End_IO when Finished is
  begin
   ...
  end End_IO;
  procedure Handler is
  begin
    ...
    Finished := True;
  end Handler;
end Driver;
```

hardware errors in I/O operations. Since error detection and signalling is a key element of most device drivers, lower-level mechanisms such as error status variables or error parameters in subprograms must be used. Some implementations (see e.g. [1]) allow a fine-grain control of exceptions by specifically restricting the use of exception handlers or exception propagation, but the results are roughly the same. As for No_Access_Subprograms, its implications are not obvious for simple drivers, but this restriction may cause problems for drivers with initialization-time configuration, as discussed in section 4.

# 3 A generic driver architecture

Computers are built up with a set of modules of three basic types: processors, memories and I/O devices, the latter being in charge of communicating with the computer environment through the so-called peripheral devices. Nowadays, the common way to interconnect computer components is by means of a computer bus or, more often, a computer bus hierarchy. The usual arrangement is to interconnect components on the same board with a local bus, and communicate different boards by means of a backplane bus.

The I/O device interface from the processor side usually consists of several registers that can be classified as:

- **Status registers**, which are used to store the status of the attached device. The processor can check the status of a device by reading its status registers.
- **Control registers**, which accept commands from the processor that are decoded by the I/O module in order to issue the corresponding request to the peripheral device.
- **Data registers**, which perform data buffering in order to decouple the different transfer rates of the main memory and the peripheral device.

A device driver is a software module that provides application code with access to a peripheral device. The application code invokes driver operations in order to interact with the device by means of commands that are sent to the device. When the device sends data or control information back to the driver, it completes the operation at the application level by returning from the call or by invoking other routines in the application. Device drivers also provide for interrupt handling and other synchronization operations. Due to their strong interaction with the device, device drivers are hardware-dependent and operating systems-specific in nature.

In order to enable the driver to interact with the device, the I/O registers must be allocated a unique address in an address space that can be made accessible to the processor. For port-based addressing architectures, the I/O registers can be accessed by subprograms including assembly code instead.

In a similar way, interrupt and DMA (Direct Memory Access) request lines have to be properly set and identified. Real-time modular computers are commonly based on

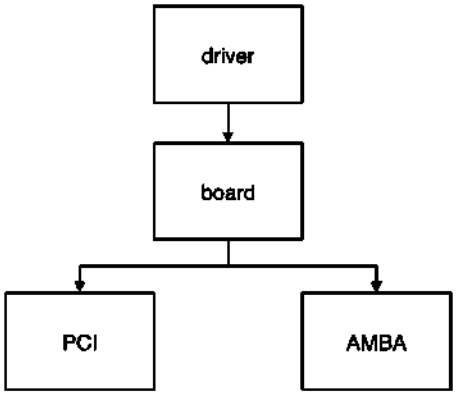


**Figure 1 Generic driver architecture**

standard backplane buses, such as VME, EISA or PCI, where processors, memory modules, and I/O boards are plugged. Some backplane buses provide jumpers or micro-switches for manual configuration of the addresses in each board. Other modular buses do not provide such low-level mechanisms, and the board configuration is done by reading board parameters and writing the settings on board registers, using a separate configuration address space. In this case, an initialization routine has to be developed as part of the device driver, which is commonly called a "plug and play" routine. Such a routine typically includes locating I/O devices by exploring boards that are connected to the system, calculating proper settings for the device, and writing them onto the configuration registers.

Once initialized, the main functions of a device driver are translating higher-level commands into device-specific commands, reading device states, synchronizing the operation of the device with the processors, and transferring data transfer to or from main memory.

The software architecture should ideally reflect this organization, by providing separate components for handling peripheral devices and buses. Figure 1 shows a generic architecture for a device driver that uses an AMBA bus [2] as a local bus, and a PCI bus [18] as a backplane bus. It is modelled after other software architectures that have been successfully implemented for other devices [4], [15], [17].

# 4 Communications drivers for ORK+ and LEON computers

## 4.1 Introduction

Two communication drivers for a particular platform used in space systems are described in this section to illustrate some concepts that must be taken into account for developing device drivers for high-integrity systems. The hardware platform is based on a LEON2 computer [3], a radiation-hardened implementation of the SPARC V8 architecture [19]. The software platform is based on the GNATforLEON compiler[1] and ORK+, the current version of the Open Ravenscar real-time Kernel [8][20].

[1] www.adacore.com

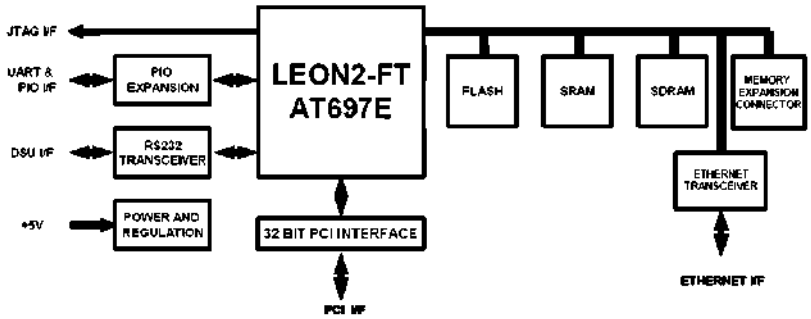


**Figure 2 GR-CPCI-AT697 CPU board block diagram (reproduced from [10]).**

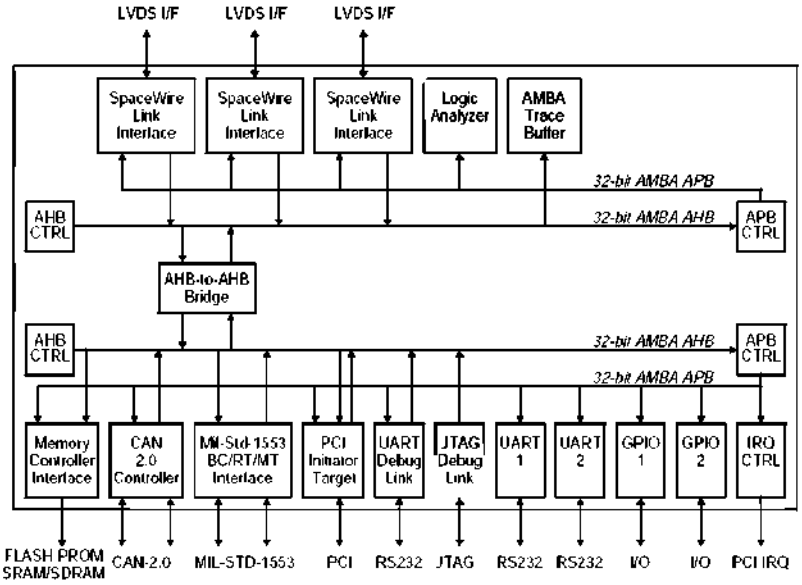


**Figure 3 RASTA interface board block diagram (reproduced from [11]).**

## 4.2 Hardware platform

GR-RASTA is a development and evaluation platform for LEON2 and LEON3-based spacecraft avionics built on a Compact PCI (cPCI) backplane bus. The computer has two cPCI boards:

- *GR-CPCI-AT697*: this is the processor board. It includes a LEON2 processor and memory. Its structure is shown in figure 2. The board has a PCI bridge to access the cPCI backplane bus.
- *GR-CPCI-XC4V*: this is an interface board based on a FPGA which has several I/O modules, including three SpaceWire links. Its design is based on an AMBA[2] bus to which the units are connected. It also has a PCI bridge to access the cPCI backplane bus. The structure of this board is shown in figure 3.

System software is usually unaware of the bus hierarchy, aside from the startup configuration of the plug-and-play feature. However, it is important to take into account the "endianness" of the different buses of the hierarchy as it has a strong influence on the definition of device registers. The SPARC v8 architecture, and therefore LEON, is big-endian. This is also the byte ordering of the AMBA buses in LEON processors. However, the PCI bus is little-endian, as it was mainly developed for Intel x86 processors. In this way, I/O device multibyte registers will suffer byte twisting as shown in figure 4. This issue must be taken into account for PCI I/O device multibyte registers as well as for DMA transfers. Accordingly, PCI hosts and PCI DMA I/O devices must be properly initialized.

[2] The Advanced Microcontroller Bus Architecture (AMBA) is a system and peripheral bus widely used in System-on-a-chip (SoC) designs.

## 4.2 The SpaceWire Device

The Gaisler SpaceWire (GRSPW) core handles the lower-level layers of the SpaceWire protocol [9]. It is an intelligent I/O device with Direct Memory Access (DMA) and interrupt-based synchronization with the CPU. The interrupt service routine (ISR) is expected to read the status registers so as to check if the operation has been successfully completed.

The GRSPW core has three main parts:

- The link interface, which handles the communication on the SpaceWire network and consists of a transmitter, receiver, and FIFO interfaces. FIFO interfaces are provided to the DMA engines and are

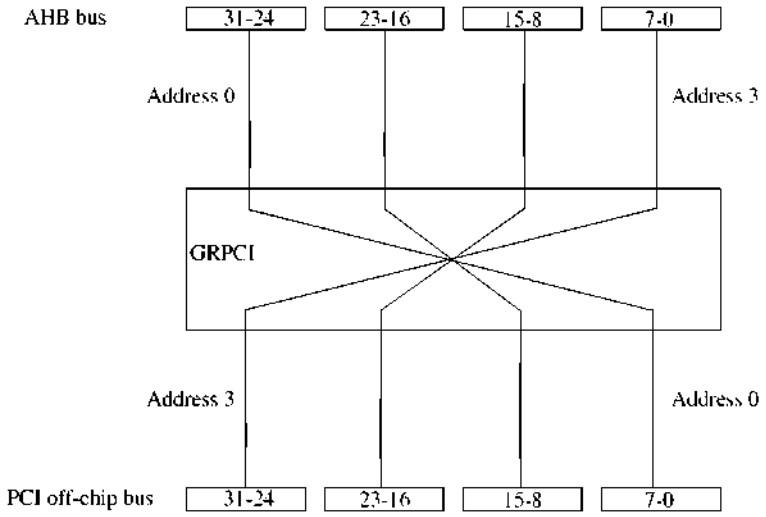


**Figure 4 AMBA to PCI bus byte twisting.**

used to transfer a number of characters (N-Chars in the following) between the AMBA and SpaceWire domains during reception and transmission.

N-Chars are sent when they are available from the transmitter FIFO and there are credits available. The credit counter is automatically increased when flow control tokens (FCT) are received and decreased when N-Chars are transmitted. Received N-Chars are stored to the receiver N-Char FIFO for further handling by the DMA interface.

- The AMBA interface, which consists of the receiver and transmitter DMA engines.

  The receiver DMA engine reads N-Chars from the N-Char FIFO and stores them on a DMA channel. Reception is based on descriptors located in a consecutive area in memory that hold pointers to buffers where packets should be stored. When a packet arrives it reads a descriptor from memory and stores the packet to the memory area pointed by the descriptor.

  Before reception can take place, a few registers need to be initialized, such as the node address register, which needs to be set to hold the address of this SpaceWire node. The link interface has to be put in the run state before any data can be sent. Also, the descriptor table and control register must be initialized.

  The transmitter DMA engine reads data from the AMBA bus and stores them in the transmitter FIFO for transmission on the SpaceWire network.

- The RMAP handler is an optional part of the GRSPW and handles incoming packets which are determined to be RMAP (Remote Memory Access Protocol) commands.

### 4.3 SpaceWire Driver Architecture

Figure 5 contains a diagram of the software organization of the GRSPW driver, which is an instance of the generic architecture described previously (see figure 1). The driver has four main components:

- The PCI driver component, which provides data type definitions and operations for reading and writing the PCI configuration registers.
- The AMBA driver component, which provides data type definitions and operations for scanning the AMBA configuration records.
- The RastaBoard driver component, which provides a common interface for drivers using the GR-RASTA board, as well as hooks for interrupt handlers to be called upon reception of the single hardware interrupt issued by the board.

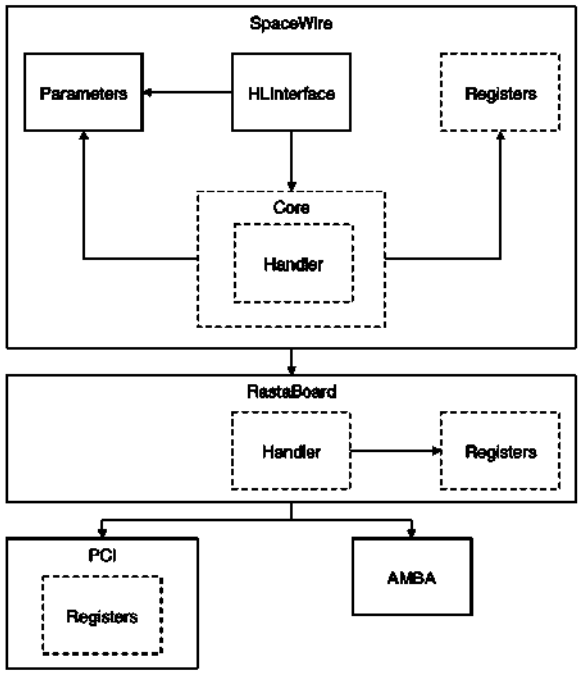


**Figure 5 SpaceWire driver architecture.**

- The SpaceWire driver component, which provides all the software items required by application programs to initialize and use the SpaceWire cores included in the GR-RASTA computer platform.

The components of the SpaceWire driver are:

- HLInterface: contains the higher-level interface for application programs, consisting of type definitions and operations initializing the SpaceWire devices, setting their node addresses, and sending and receiving data packets.
- Parameters: contains the definitions of all the parameters that can be configured by the application programmer.
- Core: contains all the code that interacts with the device registers in order to implement the I/O operations.

  This component exports a set of interface operations, which are used to implement the HLInterface operations. The component implements all the device operations in terms of the device registers and other hardware characteristics.
- Handler contains the device interrupt handler, which is invoked on the completion of I/O operations. There is a single interrupt for all the three SpaceWire devices, and a synchronization object for each of the transmit and receive sections of each SpaceWire hardware device.[3] Each occurrence of the interrupt is signalled to the appropriate synchronization object by identifying the device and function that has caused the interrupt.
- Registers: contains register and bit field definitions, as well as other data definitions that may be required to interact with the device.

### 4.4 Serial driver

There are two UARTs (Universal Asynchronous Receiver-Transmitter) in the RASTA board that provide an interface between an APB bus and a RS-232 serial line. Each UART provides the functionality for asynchronous serial communications, supporting data frames with 8 data bits, one optional parity bit, and one stop bit. As usual, it is also possible to configure the baud rate and the flow control.

UART devices are not message-oriented as SpaceWire devices, but character-oriented devices, i.e. an UART I/O operation involves just one character. Nevertheless, higher level software usually needs to send or receive a set of characters which builds up a message. In order to provide this functionality, the driver includes two separate memory buffers for storing messages:

- Transmit buffer: when higher level software sends a message, the corresponding data are pushed into this buffer and then the transmission starts until the buffer is empty. The interrupt service routine is in charge of transferring data from the buffer to the transmitter register.
- Receive buffer: when a data item is received, the interrupt service routine transfers it from the receiver register to this buffer. In this way, higher level software can receive messages by getting the data from this buffer.

[3] The GR-RASTA interface board has three SpaceWire devices.

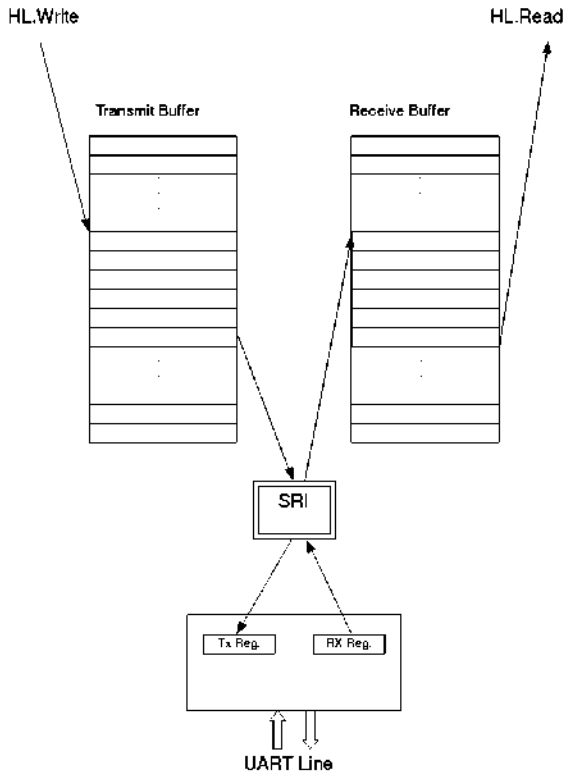


**Figure 6 UART buffers arrangement.**

These intermediate buffers are stored in main memory, and their sizes can be specified with the Buffer_Size parameters (declared in Uart.Parameters). If the value of these parameters is changed, the driver needs to be recompiled. Figure 6 shows the data flow between the UART registers and the intermediate buffers.

It must be noticed that the two-step driver pattern would have been useful when calling the driver's high level operations HL.Write and HL.Read. Both operations deal with messages and they try to read or write a set of characters from or to intermediate buffers. A call to HL.Read can be made with a message length greater than the currently stored in the receive buffer and thus the calling task may have to wait for the arrival of the rest of the message. As shown in section 2, this can be done in full Ada by using a requeue statement, but it has to be must be transformed into the alternate pattern shown in listing 2 in order to comply with the Ravenscar profile.

Figure 7 contains a diagram of the software architecture of the GRUART driver, which is an instance of the generic architecture described in section 3.

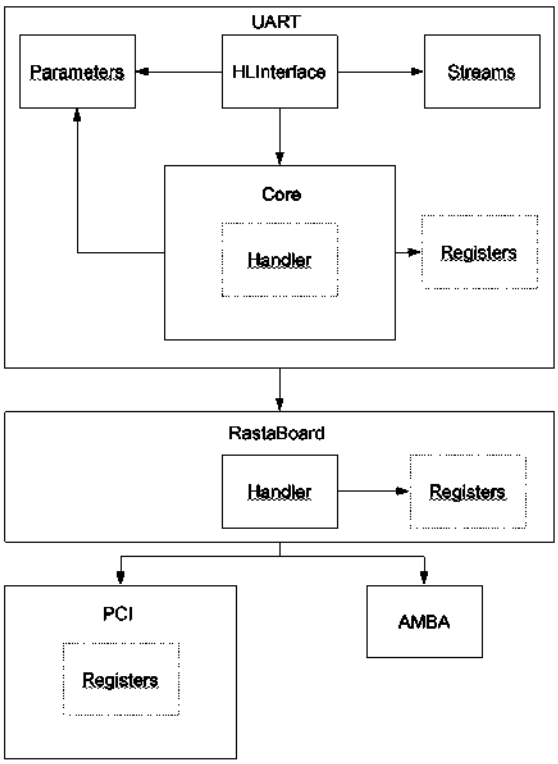


**Figure 7 UART driver architecture.**

## 5 Conclusions

The main issues related to writing device drives in Ravenscar Ada have been examined in the paper. A first conclusion is that the low-level mechanisms of the Ada language make it comparatively simple to develop device drivers in a high-level language. Features such as representation clauses and protected interrupt handlers allow the designer to build high-level abstractions of the hardware and greatly simplify writing the functional code of the drivers. Using record fields to name register bit groups improves the code readability compared to the lower-level bit mask approach used by other languages.

The good news is that these useful mechanisms are compatible with the Ravenscar profile, and thus can be used to build device drivers for high-integrity embedded real-time systems. The only potential problem that has been identified is the inability to use the requeue statement to write interrupt drivers using the well-known two-step synchronization pattern. However, a simple workaround has been proposed that only requires the restriction that a protected entry can only be called by one task.

A software architecture that can be used to develop device drivers for LEON computers has been introduced in the paper. Two driver instances for communication devices have been built based on the architecture. The authors' experience has been very positive, and is currently being continued with the developing of additional device drivers for the ORK+ real-time kernel and the GR-RASTA LEON computer boards within ESTEC, the European Space Research and Technology Centre of ESA.

## Acknowledgments

This work has been funded in part by the Spanish Ministry of Science, project TIN2008-06766-C03-01 (RT-MODEL), and by the European Space Agency, ESTEC/Contract No. 21392/08/NL/JK.

## References

[1] AdaCore (2009). *GNAT Reference Manual.*

[2] ARM (2003). *AMBA 3.0 Specification.*

[3] Atmel (2005). *Rad-Hard 32 bit SPARC V8 Processor —AT697E.*

[4] D. Berjón (2005). *Desarrollo de un subsistema fiable de comunicación para sistemas de tiempo real.* Master's thesis, ETSIT- UPM. In Spanish.

[5] A. Burns, B. Dobbing and G. Romanski (1998). *The Ravenscar tasking profile for high integrity real-time programs.* In L. Asplund (ed) Reliable Software Technologies—Ada-Europe'98. LNCS 1411, Springer-Verlag, pp 236-275.

[6] A. Burns and A. Wellings (2007). *Concurrent and Real-Time Programming in Ada.* Cambridge University Press.

[7] A. Burns and A. Wellings (2009). *Real-Time Systems and Programming Languages.* 4th edn. Addison-Wesley.

[8] J.A. de la Puente, J.F. Ruiz and J. Zamorano (2000). *An open Ravenscar real-time kernel for GNAT.* In H.B. Keller and E. Plödereder (eds), *Reliable Software Technologies—Ada-Europe 2000.* LNCS 1845, Springer-Verlag, pp 5–15.

[9] ECSS (2008). ECSS-E-ST-50-12C: *Space engineering — SpaceWire — Links, nodes, routers and networks.*

[10] Gaisler Research (2005). *LEON2 Processor User's Manual.*

[11] Gaisler Research (2006). *RASTA Interface Board* FPGA User's Manual.

[12] ISO/IEC: Std. 8652:1995/Amd 1:2007. *Ada 2005 Reference Manual. Language and Standard Libraries.* LNCS 4348, Springer-Verlag.

[13] ISO/IEC: TR 24718:2005. *Guide for the use of the Ada Ravenscar Profile in high integrity systems.* Based on the University of York Technical Report YCS-2003-348 (2003).

[14] ISO/IEC: TR 15942:2000. *Guide for the use of the Ada programming language in high integrity systems.*

[15] D.S. Morilla (1995). *Programación en Ada del LANCE Am7990.* Master's thesis, FI-UPM. In Spanish.

[16] PRAXIS Ltd (2008) *The SPARK Ravenscar Profile.*

[17] J.E. Salazar, J.E.: *Desarrollo de un driver para un sistema espacial de alta integridad*. Master's thesis, FI- UPM. In Spanish.

[18] T. Shanley and D. Anderson (1999). *PCI System Architecture*. 4th edn. Mindshare Inc.

[19] SPARC International (1992). *The SPARC architecture manual: Version 8*. Prentice-Hall.

[20] S. Urueña, J.A. Pulido, J. Redondo and J. Zamorano (2007). *Implementing the new Ada 2005 real-time features on a bare board kernel*. Ada Letters, vol XXVII no 2, p 61–66.