Interfacing Low-Level C Device Drivers with Ada 95

Steven Doran
Litton Guidance & Control Systems
550 Canoga Ave
Woodland Hills CA 91367
(818)715-2348

dorans@litton.gcs.com

1. ABSTRACT

The personal computer hardware marketplace has grown rapidly in recent years. Many software projects, as a cost-cutting measure, are buying "off-the-self" items to meet heir hardware requirements. Almost all of the device drivers for these devices are written in the C programming language. However, the selection of the programming language for the project does not need to be confined to C. This paper details the powerful tools in Ada 95, such as pragmas to interface code written in other programming languages to Ada 95 applications. An example of a generic real-time Ada 95 application interfacing with a low-level C serial device driver is used to aid the reader in the concepts and idea's discussed in the paper.

1.1 Keywords

Ada 95, real-time, device drivers, C programming language

2. INTRODUCTION

The personal computer hardware marketplace has grown exponentially in recent years. As a consequence, many software projects, in order to save on their development costs, are buying "off-the-shelf" items to meet their hardware requirements. Many times a device driver is included with the hardware and almost all the time the device driver is written in the C programming language. Since the Department of Defense (DoD) dropped the Ada mandate, many project managers have been debating on which programming language to use on their project: Ada or C/C++. (Since the Ada vs. C/C++ debate is a complex one, this paper will only focus on one criteria.) The programming language selection might be biased towards C/C++ in the assumption that C/C++ would be the only language that can successfully interface with the given device driver. Another assumption might be that Ada 95 does not have the functionality to interface to the given device driver. Both assumptions are false. Ada 95 has several powerful features that give it the ability to interface with several programming languages, including C/C++[1].

This paper covers in detail the most important features in Ada 95 in order to interface with C device drivers. This paper also describes device drivers and presents an high level overview of the functionality most device drivers perform. The structure of this paper is as follows:

Section 3 will define device drivers. This section will give a high-level overview of what functions a device driver needs to perform in order to control a hardware device efficiently.

Section 4 will discuss pragmas in Ada 95 specific to interfacing with other computer languages. This section will define these pragmas and the rules on how to use them. Examples will be used to aid the reader in understanding the pragmas disused in the section. This section will also discuss the steps needed to build an Ada 95 executable with embedded pragmas calls to C subprograms.

Section 5 will discuss the Interfaces package in Ada 95. This section will give the semantics and declarations of the Interfaces packages. Examples will be used to aid the reader in understanding the package.

SIGAda'99 10/99 Redondo Beach, CA, USA
© 1999 ACM 1-58113-127-5/99/0010...$5.00

[1] This paper relates to Ada 95 only. Binding to foreign languages in Ada 83 was limited.

Section 6 will describe a fictitious real-time Ada 95 application called *Train_Monitor. Train_Monitor* receives several data inputs from sensors located on various locations on the train. These sensors collect data on the "state of health" of the train as it runs along the tracks. *Train_Monitor* constantly performs calculations on the data received from these sensors and sends messages via an 28K serial connection to a separate computer that displays the train status to the conductor.

3. DEVICE DRIVERS

A device driver is a software program that resides between a hardware device and the software applications. This code is specifically created to perform device control operations for the hardware device. Basic device control operations that every device driver needs to perform are:

- open
- close
- read
- write
- ioctrl

The open control operation initializes the hardware device. The normal sequence of events that is performed when the open operation is executed is: The driver will determine if there are any hardware errors (device is not ready, device does not exists, etc.) Next the driver will initialize the hardware including allocation of on-board memory if the device is so equipped. When the open operation is executed, the previous state of the device will be lost.

The close control operation closes the hardware device to software applications. Most device drivers will deallocate any resources that the open allocated.

The read control operation transfers data from the hardware device to the software application. Effective device drivers will perform checks to determine if all the data was successfully sent from the device. Normally this is done by counting the amount of bytes transferred. If the count is equal to the requested byte size passed to the read operation, then the read was successful. If the count is less than the requested byte size, then only part of the data was transferred. There can be a number of reasons that can cause this error to occur and is dependant on the hardware device being accessed by the driver. Most drivers will retry the read operation. If the count was greater than the requested byte size or the count is negative, then an error has occurred and the driver should log the error to the operating system. If the driver does not check the read operation for errors, the device driver will not be as robust and makes debugging a nightmare.

The write control operation transfers data from the software application to the hardware device. The same situation applies to the write operation as the read operation. Effective drivers perform checks to determine if all the data was successfully sent to the device by counting the amount of bytes transferred. If the count is equal to the requested byte size passed to the write operation, then the write was successful. Again, most drivers will retry the operation. The same error conditions of the read operation also apply to the write operation.

The ioctrl control operation offers a device-specific entry point for the device driver to issue commands. In essence, ioctrl is for controlling the I/O channel. An example of ioctrl is the changing of the baud rate of a parallel port.

All control operations that were described return status flags. It is important that Ada 95 applications properly interface with the device driver to read these status flags for debugging purposes. The details on how to interfaces with the device driver will be discussed in Sections 4 and 5.

In order for the device driver to perform its given tasks, it needs to be linked into the operating systems kernel.

3.1 Operating System Kernel

A Kernel is the heart of every operating system. The kernel manages the system resources of the computer: CPU usage, memory management, etc. The kernel determines when a process should be created or destroyed. It also handles inter-process communication, and the priority scheduling of processes. However, the most important function the kernel provides, with regard to device drivers, is device control. A kernel must have a device driver for every physical device installed. This actually simplifies the device driver since the driver needs to be coded for only one specific device. This results in the ability to add or delete a device without effecting other device drivers or the operating system. When a software application needs to access the device driver, it needs "entry points" into the kernel. These entry points are called System Calls..

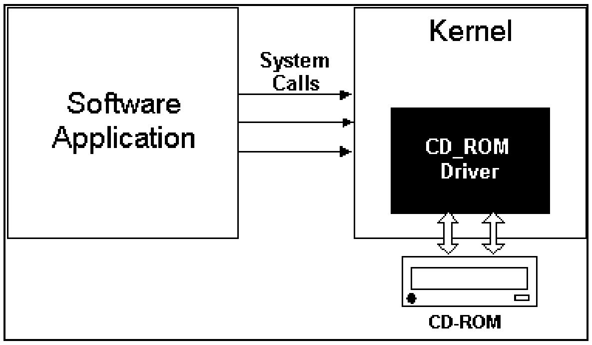

Figure 1

Interaction between an Software Application and an CD-ROM

There are two basic types or device drivers: character drivers and block drivers. Character drivers are different than block drivers in the fact they can manage I/O requests that are not fixed in size. This gives character drivers the ability to control a wide range of hardware devices. Serial device drivers are a example of a character device. Block device drivers can only transfer fixed-sized buffers. Usually the operating system, not the device driver, manages the fixed-sized buffers. The driver is called when the buffer requested from the operating system is not in the cache or the buffer has been changed. Block drivers also differ from character drivers in that the data sent to a block driver is addressable by a position. This position is usually determined by the software application, not the device driver. An driver for a IDE controller is an example of a block driver.

4. PRAGMAS

In order for Ada 95 to interface with foreign languages, the data being transferred from one language to another must be converted to the appropriate conventions. Ada 95 has three pragmas to perform this operation: pragma Import, pragma Export and pragma Convention. Pragmas Import and Convention are essential to bind Ada 95 applications to low-level C device drivers. The pragma Export should never be used[2].

4.1 Pragma Import

The pragma Import is used to import subprograms and data types defined in foreign languages to an Ada application.

[2] The non-use of pragma Export in binding Ada 95 applications to low-level C device drivers will be explained in Section 3.1.2.

The pragma Import syntax is defined in the *Ada 95 Reference Manual* as:

```
pragma Import(
        [Convention =>] convention_identifier,
        [Entity =>] local_name
        [,[External_Name =>] string_expression]
        [,[Link_Name =>] string_expression]);
```

The first parameter, convention_identifier, is the foreign language the object is defined in. The Import pragma support most high-level languages, C/C++, COBOL, FORTRAN and Pascal, and low level assembly languages. The second parameter, Entity, is the Ada name for the foreign language subprogram. The third parameter, External_Name is the name of the foreign language subprogram to be interfaced. The forth parameter, Link_Name, is the name of the object file to be sent to the Ada 95 Linker. For instance, suppose a C function called "C_Display" needs to be interfaced to Ada 95. C_Display is declared as:

```
int c_display (int num)
{
   printf("The Number Passed from Ada 95 to C
            is  => %d\n", num);
     return 0;
}
```

First an Ada subprogram needs to be mapped to the C_Display. Then the Pragma Import can be used. The syntax would look like the following:

```
procedure C_Display (Num : Integer);

pragma Import (C, C_Display);
```

The third parameters in pragma Import can be optional if the Ada 95 subprogram and the C subprogram are declared using the same name. Otherwise, the third option must be used:

```
pragma Import
    (C, Ada_Subprogram, "C_Subprogram");
```

In the above case, the parameter External_Name must be in quotes.

The object file created by the C compiler must be passed to the Ada 95 Linker. The Ada 95 linker will search through all object files passed to it, so in most cases, the fourth parameter, Link_Name, can be ignored. Notice the integer that was returned from C_Display was ignored. This will be explained further in Section 5.

4.2 Pragma Export

The pragma Export is used to export Ada 95 subprograms to C. The syntax is very similar to pragma Import and is defined in the Ada 95 Reference Manual as the following:

```
pragma Export(
    [Convention =>] convention_identifier,
    [Entity =>] local_name
    [,[External_Name =>]string_expression]
    [, [Link_Name =>] string_expression]);
```

Refer to Pragma Import for an explanation of the parameters of pragma Export. Below is an example pragma Export routine:

```
function Ada_Function return Integer;
pragma Export (C, Ada_Function, "callada");

function Ada_Function
  return Integer is

   Value : Integer := 4;

begin   --| Ada_Function

   return Value;

end Ada_Function;
```

In order for the C program to call an Ada 95 subprogram, that subprogram needs to be declared as an external function. The declaration for "Ada_Function" in the C program would be:

```
extern Ada_Function;
```

The rest of the C code is the following:

```
#include <stdio.h>
extern ada_function;

int main()
{
    int val;
    val = ada_function;
    printf("The value passed from Ada is %d\n",val);

    return 0;
}
```

The pragma Export should not be used in the binding of Ada 95 applications to low-level C drivers. Device drivers should never call subprograms in the software application. If a device driver depended on an application subprogram, a device driver would have to be created for every application. Even if only one application is running on dedicated hardware, the device drivers should still be independent from the application. Otherwise, expanding either the hardware or software would be more difficult, expensive and time consuming. This paper described the syntax of the pragma Export only as an reference to help the reader better understand the tools and abilities Ada 95 has to offer when interfacing foreign programming languages.

4.3 Pragma Convention

The pragma Convention is used to specify that an Ada 95 object should use the conventions of the foreign language. The syntax of pragma Conventions is very similar to pragma Import and Export. It is defined in the Ada 95 Reference Manual as the following:

```
pragma Convention
    ([Convention =>] convention_identifier,
    [Entity =>] local_name);
```

For instance, suppose an Ada 95 subprogram was declared as the following:

```
procedure Ada_Call (Num : in Integer)
    is separate;
```

pragma Convention (C, Ada_Call);

This informs the compiler, and the reader of the code, that the subprogram was written in Ada 95, but it is intended to be called from a C program. This will affect how the C program will reference the parameters of Ada_Call. The C programming language does not have the functionality of protecting parameters (in, out, in out.) All parameters are considered "in out" in C. So if an Ada 95 function being passed through the pragma Convention has an "out" parameter, the parameter will be translated as an "in out." C is comparable to Ada 95 in how it passes parameters, by value, by pointer and by reference. As all Ada programmers know, passing by reference is the default option in Ada. In the Ada_Call example, Num would be passed as "in out" even though it is declared as "in."

Another aspect of the pragma Convention is in types and objects. Just as the pragma Convention indicated to the compiler to use C convention on subprograms, the pragma can to the same functionality to declared types. Below is an example:

pragma Convention(C, Ada_Type);

This will instruct the Ada 95 compiler to use C conventions on Ada_Type.

4.4 Building an Ada 95 executable with embedded pragmas linking C subprograms

In order to build an Ada 95 executable with embedded C subprogram calls, the object file that was created by a C compiler must be passed to the Ada 95 linker. Most hardware manufactures, especially in the Unix environment, supply the source code of the device driver. This makes binding your Ada 95 application easy, since the only required step is to compile the device drivers source code. If the hardware manufacturer does not supply the device drivers source code, then ask for the object files and documentation on the driver. Without the object files, it is impossible to bind your Ada 95 application to the device driver. Some companies will require you to complete a non-disclosure agreement. If the manufacturer is unable, or unwilling, to accommodate you needs, then you might want to consider another supplier.

Passing C object files to the Ada linker varies greatly between Ada 95 vendors[3]. Below are some examples with popular Ada 95 compilers to help convey an understanding of the process. With GNAT Ada 95[4], the syntax is the following:

gnatlink *Ada_program.ali c_object_file*

It is possible to pass more than one object file to gnatlink. For instance:

gnatlink *Ada_program.ali*
c_object_file1 c_object_file2 ....

With the ObjectAda[5] Ada95 compiler, Click on Project, then Settings, then Link. Enter the path to the C object file in the "Pass to linker" dialog box.

5. INTERFACES PACKAGE

The package Interfaces is a child package of the Ada 95 library package *Standard*. It contains hardware-specific types and declarations useful for interfacing to foreign languages. The Interfaces package is also a parent package to several other child packages. In this paper, we will concentrate our study to the child packages related to the C programming language; however, the Interfaces package contains several other child packages to interface FORTRAN, COBOL. and assembly language

5.1 Interfaces.C

Interfaces.C is a child package of Interfaces and contains the basic types, constants and subprograms which allow Ada 95 applications to pass scalar types and strings to C functions[6]. This package also supports the Import, Export, and Convention pragmas. One important function the Interfaces.C package performs is the handling of the differences between the two languages. For instance, the C programming language does not implement procedures. An Ada 95 procedure would be interfaced by the Interfaces.C package as a C function returning a void. Ada 95 functions are similar to C functions so no interpretation is needed. The only major difference is Ada 95 functions cannot return a "void." Because C does not implement procedures, it does not mean that procedures cannot be used. An Ada procedure can correspond to a C function; however, the Interfaces.C package will ignore the returning value from the function. Some of the major differences between Ada

---

[3] Refer to the users manual supplied by your Ada 95 vendor.

[4] GNAT is a product of Ada Core Technologies, Inc. http://www.gnat.com/

[5] ObjectAda is a product of Aonix. http://www.aonix.com/

[6] Consult the Ada 95 Reference Manual for the complete declaration of Interfaces.C [Ada95a]

95 and C are in strings and pointers. The Interfaces.C package has two child packages to manage these differences.

5.2 Interfaces.C.Strings

The package Interfaces.C.Strings has declarations of types and subprograms which allow Ada 95 applications to allocate, reference and update C-style strings. Strings in the C programming language are simply character arrays. Commonly, a C program will declare a string as a pointer to an character array. In C, the syntax is char **variable_name*. The type Chars_Ptr declared in Interfaces.C.Strings is equivalent to char **variable_name*. With Chars_Ptr, an Ada 95 application can create a C string and pass it to a C function. This is a valuable functionality in interfacing to low-level C device drivers

since a majority character device drivers pass data to-and-from the hardware device using character pointers. Below is a example of an C subprogram with an character array as one of its parameters:

```
int block_output (char *buf, size_t length);
```

An Ada 95 equivalent to char *buf would be:

```
Character_Buffer :
        Interfaces.C.Strings.Chars_Ptr;
```

To further the block_output example, suppose a Ada 95 subprogram needed to be written to test the output of the device. For simplicity, we will assume the hardware device has already been initialized. The requirements of this test subprogram state a set of zeros needs to be outputted from the device. First the character array needs to be declared. In this example, a constant Ada 95 string will be declared. Then the string will be converted to a chars_ptr using the function New_String defined in Interfaces.C.Strings:

```
Test_String : constant String := "0000000000";

Character_Buffer : Chars_Ptr;
```

Once the variables are declared, Test_String needs to be converted to a chars_ptr:

```
Character_Buffer:=
   New_String (Str => Test_String);
```

Before the C function block_output is called, the second parameter needs to be addressed. This second parameter is the length of the chars_ptr. The type of this parameter is size_t, which is always used in low-level C drivers. Fortunately, Interfaces.C.Strings has two tools to help us obtain the size of the variable Character_Buffer. First, size_t is defined in Interfaces.C.Strings. Second, a common function in the C programming language, called strlen, is also defined in Interfaces.C.Strings. As its name applies, strlen returns the length of a string (chars_ptr) in size_t:

```
Character_Length : Size_T;
Character_Length:=
   Strlen(Item => Character_Buffer);
```

Now that the two parameters of block_output have been determined, the function can be called:

```
block_output
   (Character_Buffer, Character_Length);
```

Figure 2 below is the complete example. Notice the Integer that is returned from block_output is used for error checking. A standard convention in low-level C device drivers is to return a -1 if the function fails. This can happen for a number of reasons; however, checking for error conditions helps the Ada 95 application become more user friendly.

```
with Text_Io;
with Interfaces.C;
use  Interfaces.C;

with Interfaces.C.Strings;
use  Interfaces.C.Strings;

procedure Test_Device is

  function Block_Output
    (Item : Chars_Ptr;
     Size : Size_T) return Integer;

  pragma Import (C, Block_Output);
```

```
    --| Variable Declarations
    Test_String : constant String :=
                                   "0000000000";

    Character_Buffer : Chars_Ptr;
    Character_Length : Size_T;
    Result : Integer;

begin   --| Test_Device

    Character_Buffer:=
        New_String (Str => Test_String);

    Character_Length:=
        Strlen(Item => Character_Buffer);

    --| Send Test Pattern to Hardware Device
    Result := Block_Output
                  (Item => Character_Buffer,
                   Size => Character_Length);

        if Result = -1
          then
            Text_Io.Put_Line("Device Failure!!");
        end if;

end Test_Device;
```

Figure 2 *- Test Device Example*

5.3 Interfaces.C.Pointers

Interfaces.C.Pointers is an generic package with declarations of types and subprograms which allows the Ada 95 applications to perform C-style operations on pointers. This includes arithmetic operations, increment and decrement of pointers, and copying data from the pointer. This functionality is needed because pointers are treated differently between the two programming languages. In the C programming language the value of a pointer is the real memory address. In Ada, a type used to access the data of a pointer. Ada access types are safer and easier to use. It is very difficult to have "lost pointer" in Ada while it is almost inevitable in C. Ada Deallocation of a pointer is much more efficient than the free command in C. There are no "memory leaks" in Ada!

The instantiation of the generic Interfaces.C.Pointers looks like the following:

```
package C renames Interfaces.C[7];
package Character_Array_Pointer
    is new C.Pointers
        (Index               => C.size_t,
         Element             => C.char,
         Element_Array       => C.char_array,
         Default_Terminator  => C.nul);
```

Interfaces.C.Pointers is not exclusive to character arrays. Any type of arrays can be used. The parameter Default_Terminator has two options. Either the Default_Terminator with a special terminator element (such as a C.nul as in the example) is used or the programmer tracks the length of the array.

General "house keeping" of pointers still apply to Interfaces.C.Pointers. All pointers should be deallocated when no longer needed. The C command free is not supported in Interfaces.C.Pointers.

6. CASE STUDY - FICTITIOUS REAL-TIME TRAIN APLLICATION

This is an example of a Fictitious Real-time Train Application called Train_Monitor using the Ada tools and concepts described in this paper. Train monitor calculates real-time data from sensors throughout the train. Once the data is obtained, it performs real-time calculations and sends messages to another computer via a serial connection. These messages are then displayed to the conductor. Lets assume the reason for two computers is to distribute processing: one computer dedicated to the manipulation of data, one computer dedicated for the human interface display. For simplicity, how Train_Monitor calculates its data is not discussed. What is discussed is how Train_Monitor sends the messages to the second computer. Since the train is a fast train and the messages need to be sent to the conductor swiftly, a task is used to send the data to the device driver. Below is the components of our example:

[7] The rename is not essential to the instantiation, it was used to clarify the example

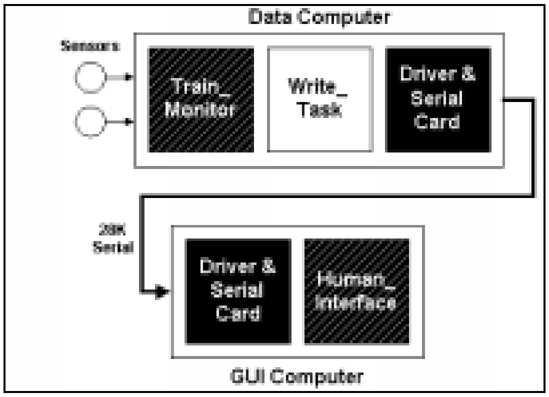

Figure 3 *- Components of Case Study*

In order to obey the rules of an Object Oriented Design, all Ada subprograms in Train_Monitor are defined in a package called Train_Monitor_Interface. For clarity, all Ada subprograms have the same names as their C counterparts except for Send_Error_Message. The C counterpart to Send_Error_Message is write_dev and is declared as the following:

```
int write_dev (char *buf, int count);
```

Since our Ada 95 application only calls write_dev to send error messages, the name Send_Error_Message was used for readability. Write_Dev has an integer parameter that counts the size of the message sent to the function. The write_dev function compares the count value to the amount of data actually sent to the device. If a discrepancy occurs, the driver re-sends the data to the serial device.

6.1 Initialize (Open) Serial Card

To initialize the serial device, Train_Monitor will call the C function init_device via the Ada procedure Initialize_Device. The init_device has one parameter, the speed of the serial connection. The requirements of Train_Monitor states the connection between the two computers is to be 28 kilobytes a second. Initialize_Device looks like the following:

```
with Text_Io;
use  Text_Io;
with Train_Monitor_Interface;
with Interfaces.C.Strings;
use  Interfaces.C.Strings;
procedure Initialize_Device
     (Connection_Speed : in Float) is

   Status : Integer;
   Count  : Interfaces.C.Size_T;
   Error  : Interfaces.C.Char_Array (1..50);
   Error_Message : Chars_Ptr;

begin   --| Initialize_Device

    Status :=
       Train_Monitor_Interface.Init_Device
           (Rate => Connection_Speed);

       if Status = -1
         then
           --| Send an error message to the
           --| Conductor
           Error(1..35) := "Computer Error -
                           Don't Drive Train!";
           Error_Message :=
              New_Char_Array(Chars => Error);
           Count := Strlen(Error_Message);

           Train_Monitor_Interface.
              Send_Error_Message
                  (Count => Count,
                   Error => Error_Message);
       end if;

end Initialize_Device;
```

Figure 4 ***-*** *Procedure Initialize_Device*

Again, notice Initialize_Device stores the integer returned from init_device and uses the value to determine if an error message needs to be sent to the conductor. Char_Array is a C character array defined in Interfaces.C. New_Char_Array is a function defined in Interfaces.C.Strings that converts a C character array to chars_ptr.

6.2 Close Serial Card

Once the train has preformed its duties for the day, the train is shut-down. The serial hardware device should be closed. This will prevent the device from being in a strange state that might make it fail the next day. The Ada function Close_Device is called to performs this function:

```
with Train_Monitor_Interface;
with Interfaces.C.Strings;
use  Interfaces.C.Strings;

procedure Close_Device is

   Status : Integer;
   Count  : Interfaces.C.Size_T;
   Error  : Interfaces.C.Char_Array (1..50);
   Error_Message : Chars_Ptr;

begin   --| Close_Device
   Status :=
      Train_Monitor_Interface.Close_Device;

      if Status = -1
        then
          --| Send an error message to the
          --| Conductor
          Error(1..32) := "Computer Error -
                           Shutdown Error!";
          Error_Message :=
               New_Char_Array(Chars => Error);

          Count := Strlen(Error_Message);

           Train_Monitor_Interface.
              Send_Error_Message
                 (Count => Count,
                  Error => Error_Message);
      end if;

end Close_Device;
```

Figure 5 *- Procedure Close_Device*

6.3 Read to Serial Card

Since our example data's path is only in one direction, a read procedure is not needed

6.4 Send Messages to the Conductor (Write)

To initialize the serial device, Train_Monitor will call the C

When the Train_Monitor calculates a critical event, a message needs to be sent to the conductor immediately. To assure that this occurs, an asynchronous task is used[8]. The task is launched after Initialize_Device is called. Data is passed to the task using a pointer. When an event occurs, a flag is set. The Write Task detects the change and sends the message to the driver. The Write_Task is defined as the following:

```
task Write_Task is

  entry Initialize
       (Train_Status : Status_Ptr_Type);

end Write_Task;
```

Train_Status points to the state of the train. The trains state data is defined in the record Status_Type:

```
type Status_Type is
  record
   Processing         : Boolean;
   Error_Occured      : Boolean;
   Error_Message_Size : Size_T;
   Error_Message      : Chars_Ptr;
end record;
```

The Processing field indicates that the application Train_Monitor is processing data. This field will always be true while the train is moving. The Error_Occured field is set to true when Train_Monitor calculates a error condition. The Error_Message field is the message that needs to be sent to the conductor. Before the Write_Task is launched, the pointer to the train state has to be assigned and allocated to Status_Type:

```
type Status_Ptr_Type is access Status_Type;
Status_Ptr: Status_Ptr_Type;
   .......
Train_Status  : Status_Ptr_Type;
   .......
Train_Status := new Status_Type;
```

[8] Tasking is not discussed in detail in this paper. Consult any Ada 95 test book on tasking.

The Write_Task is launched after the procedure Initialize_Device is called by the following statement.

```
Write_Task.Initialize
     (Train_Status => Status_Ptr);
```

Below is the task body:

```
task body Write_Task is

     Status : Status_Ptr_Type;

 begin   --| Write_Task
    loop
      select
        accept Initialize
             (Train_Status : Status_Ptr_Type) do
          Status := Train_Status;
        end Initialize;

          while Status.Processing = True loop

             if Status.Error_Occured
               then
                 --| Send an error message to the
                 --| Conductor
                 Train_Monitor_Interface.
                   Send_Error_Message
                       (Error =>
                          Status.Error_Message);
                   Status.Error_Occured :=False;
             end if;
         end loop;
      or
        terminate;
      end select;
     end loop;
   end Write_Task;
```

Figure 6 *- Write Task Body*

A select statement is used in order to give the task more than one rendezvous. Below is an example of a error condition in Train_Monitor:

```
if Train_Speed > 432.0
      then
         --| Send An Error Message To The
         --| Conductor
         Error(1..28) := "TRAIN TOO FAST!!
                          SLOW DOWN!!";
         Error_Message :=
         New_Char_Array(Chars => Error);

         Train_Status.Error_Occured :=True;
         Train_Status.Error_Message_Size  :=
                Strlen(Error_Message);

         Train_Status.Error_Message :=
                Error_Message;
end if;
```

6.5 Ioctrl Statements to the Serial Card

In this case study, the only ioctrl function in the device driver is to clear the on-board memory on the serial card. The procedure, Clear_Memory, handles this task. Below is a call to Clear_Memory:

```
Status :=
     Train_Monitor_Interface.Clear_Memory;

if Status = -1
  then
    --| Send an error message to the conductor
    Error(1..30) := "Computer Error - Device
                         Error!";
    Error_Message :=
       New_Char_Array(Chars => Error);

    Count := Strlen(Error_Message);
    Train_Monitor_Interface.
       Send_Error_Message
         (Count => Count,
          Error => Error_Message);
```

```
end if;
```

7. CONCLUSION

Ada 95 has a number of tools available to interface to other languages. This ability is one of many reasons Ada 95 can thrive as a programming language. Many Ada programmers who have not transitioned to Ada 95 might not be aware of the potential Ada 95 possesses. This paper has demonstrated that interfacing to existing low-level C device drivers is not difficult and ungainly as some might have suspected. The sample of the train monitor presented as a case study has presented to highlight this point.

The author would like to make the following recommendations:

1)Ada vendors should not create "proprietary" interfaces. This would "splinter" the Ada community and limit Ada's growth potential. Java, a programming language that was created to be "THE STANDARDIZED" language, was "splintered" in several different versions. Now incompatibility is a major problem in the Java community.

2)Child packages that interface to "new" programming languages should be added to the Interfaces package in the next Ada Standard. (ex. Interfaces.Java, Interfaces.C++, Interfaces.TclTk)

8. REFERENCES

[1] Reference Manual for the Ada Programming Language Copyright (C) 1992,1993,1994,1995 Intermetrics, Inc.

[2] Ada 95 Rationale - The Language - The Standard Libraries. Copyright (C) 1994 Intermetrics, Inc.

[3] Programming in Ada 95 - 2nd Edition Written by John Barnes Copyright 1998 Addison Wesley Longman Limited

[4] Linux Device Drivers. Written by Alessandro Rubini Copyright 1998 O'Reily & Associates, Inc