

The challenges of mixed language device software development

by Paul Parkinson, Wind River

THIS ARTICLE DISCUSSES TOOLS THAT CAN HELP DEVELOPERS VISUALIZE, ANALYZE AND DEBUG DEVICE SOFTWARE THAT MIXES GNAT ADA, C AND C++. IT PROVIDES PRACTICAL EXAMPLES BASED ON RTI SCOPE TOOLS AND WIND RIVER'S VxWORKS RTOS.

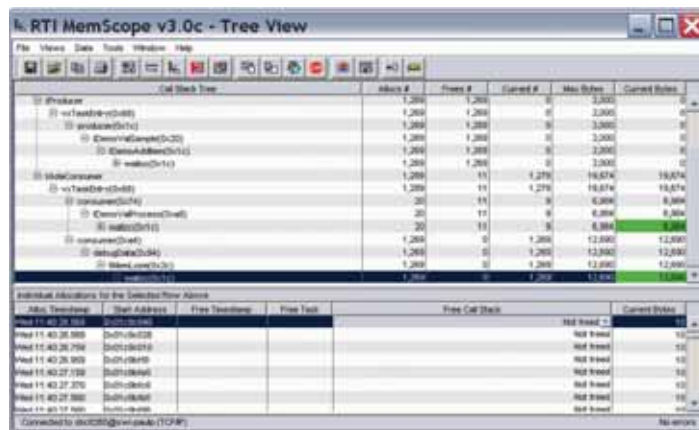


Figure 1. RTI MemScope tree view of mixed language device software

■ Implementations that mix Ada, C, and C++ provide the high-reliability and safety-critical benefits of Ada plus the advantages of using standard building blocks such as real-time operating systems, networking stacks and advanced display systems, all typically implemented in C. Mixed language programming, which has become more common in A&D and safety-critical applications since the US DoD relaxed its Ada-only mandate in 1997, brings integration and debugging challenges.

Historically, the selection of a programming language for an aerospace and defense project has been fairly clear-cut. In the US, Ada was mandated for use on DoD projects until 1997, when the required use of Ada was relaxed in favour of “an engineering approach to selection of the language to be used” (OSAD memorandum, 29th April 1997). Despite the change, many high-integrity and safety-related projects in the US continue to choose Ada for their development language, and the use of Ada is actively encouraged on safety-related projects in a number of other countries. Only recently, though, has there been a marked increase in the use of multiple language device software development. The most frequent pairing is Ada with C. Several factors may contribute to this situation. First, existing safety-critical systems implemented in Ada are being enhanced with additional functionality that requires the use of

an RTOS, usually written in C. Second, distributed applications are becoming more prevalent, through the adoption of TCP/IP and related networking technologies, which are often implemented in C. This adoption is being driven in NATO defence systems by strategies such as Network Centric Warfare (NCW) and Network Enabled Connectivity (NEC), and also in the US through the DoD Memorandum (“Internet Protocol Version 6 (IPv6)”, 9th June 2003) in relation to the use of IPv6. Third, the advances in display systems are now open to exploitation by safety-critical and safety-related device software implemented in Ada, which need to interface with OpenGL and other graphical libraries implemented in C. Finally, code reuse is a growing trend in software development, one that increases the likelihood of a system that mixes Ada and C. In aerospace systems, all these factors can occur together. A case in point is Integrated Modular Avionics (IMA) systems, where Ada and C device software can operate side by side on the same processor, driving graphical displays, and communicate over an Ethernet or AFDX network.

In order to efficiently develop and optimize mixed language device software, developers need to be able to perform the following: to call Ada procedures from C functions and vice versa; to perform mixed language source-level debugging; to understand the memory utilization

of the mixed language application and to understand the system-level behavior of the mixed language application. These requirements are explored in the following subsections, along with practical implementation examples.

Many programming languages provide limited support for interfacing to other programming languages; as a result, language compilers and debuggers have provided limited support as well. Developers have therefore needed to implement their own bindings between languages in assembler, an error-prone activity that can result in non-portable code. In the case of the Ada95 programming language, this provides well-defined interface to the C language (Ada 95 Reference Manual, Annexe B: Interface to Other Languages), enabling developers to easily and safely call C functions from Ada procedures and the converse. A federated command & control system implemented in Ada historically may have performed I/O over a bus or via serial devices directly from Ada routines. In order to upgrade this system to support IPv6 networking connectivity, though, it is likely that the application will need to interface with an IPv6 stack implemented in C.

For reasons of future maintainability, we may prefer not to embed IPv6-specific code within the Ada command and control application. Instead we may prefer to use an abstraction layer

to hide or minimize the interface with the network stack. This could be achieved through the use of the inter-process communication (IPC) capabilities provided by the underlying real-time operating system. In the case of a VxWorks-based implementation, VxWorks message queues could be used for IPC, and a GNAT Ada procedure would call the `msgQReceive()` API which is implemented in C as follows:

```
function Msg_Queue_Receive
(Msg_Queue_Id: Integer;
 Buffer: System.Address;
 Max_NBytes: Interfaces.C.unsigned;
 Timeout: Integer)
return Integer;
pragma Import (C, Msg_Queue_Receive,
"msgQReceive");
```

This enables calls from GNAT Ada to C in a straightforward manner, provided that the appropriate Ada data types are selected to interface to the data types used in the parameters to the C function. A blocking `msgQReceive()` call from the Ada application is shown below:

```
Return_Value := Msg_Queue_Receive
(Data_Queue_Id, Ada_Buffer'Address,
Result_Structure_Size, -1);
```

The reverse mapping can be achieved in a similar way through use of a `pragma Export`.

Once developers have used the ability to call C functions from Ada procedures, and vice versa, they will also want to perform source-level debugging of this mixed language application code. This presents its own challenges, particularly as compilers and debuggers have tended to use a variety of object module formats (OMF), and developers who have tried to bind C++ applications created by different compilers know this problem all too well. In recent years, industry has sought to standardize on executable and linking format (ELF) and debugging with arbitrary record format (DWARF). This has made the parsing of object files more straightforward, but these tools still need to be aware of their own naming conventions and also the naming conventions of other tools which have created object code that needs to be debugged.

In the case of the GNAT Ada & GNU C compilers, these potential obstacles were overcome by the Ada95 language and by the inherent compatibility of GNAT and GNU due to their common heritage. At the compiler level, GNU C and GNAT Pro Ada use the same GCC technology (they are two front ends to the same GCC backend). As a result, both compilers produce the same object code and debugging formats, thus enabling developers to freely mix Ada and C in their applications. In the case of the Ada command and control application, this

enables the debugger to step from an Ada procedure into a C function, following the application's flow of execution in the usual manner.

A mixed language debugger alone doesn't meet all the challenges posed by the development of mixed language applications. In the command and control system case, for example, memory may be dynamically allocated by C functions interfacing to the IPv6 stack for passing of data which may be subsequently freed by Ada procedures within the core of the command and control application. Here tools that are C-centric or Ada-centric alone will not be sufficient. Instead the developer will need

dynamic visualization tools to monitor the dynamic memory utilization of both languages in order to assess the memory utilization of the mixed language application overall.

This behavior presents the system designer with additional considerations if the Ada and C runtime systems perform dynamic memory allocation from different memory pools, or use different memory allocation schemes, and may force the designer to partition or assign memory ahead of time. Ideally, the Ada and C runtime systems would share a common underlying method. In fact, the GNAT Ada runtime library, which invokes the C runtime

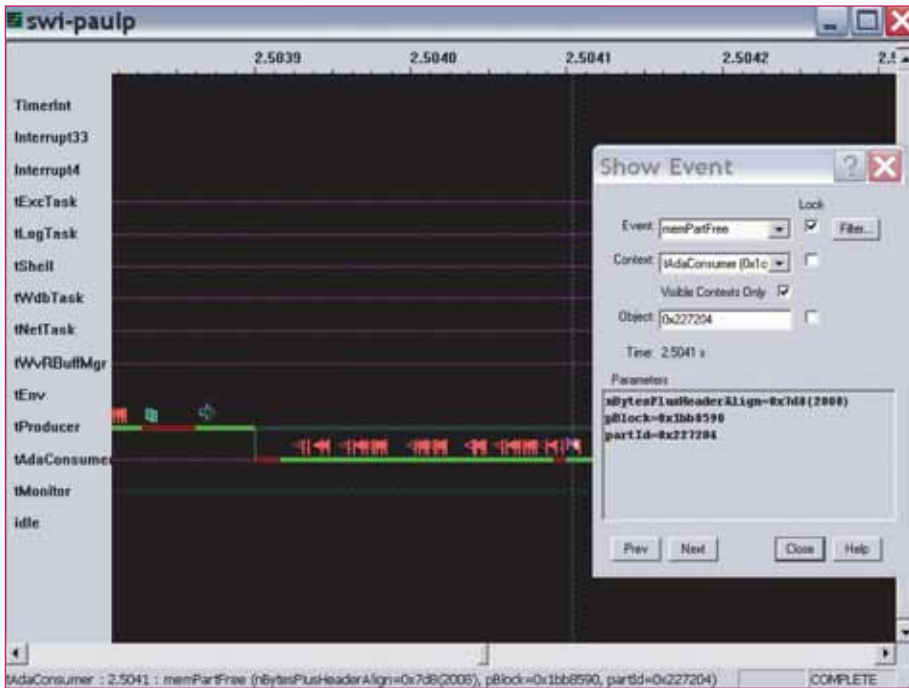


Figure 2. Wind River System Viewer showing Ada procedure call to VxWorks C API

library's dynamic memory allocation routines, including malloc() and free(), uses this approach. Thus, only accesses to the C run-time library's routines need to be traced in order to trace the dynamic allocation or deallocation of memory by either language.

In the case of the Ada command and control application, the RTI MemScope tool could be used to trace all dynamic memory allocations and deallocations by either Ada code, C code, and any dynamically allocated memory buffers passed between them on the VxWorks system. This is illustrated by figure 1, showing the VxWorks task tProducer which is implemented in C, and an Ada task tAdaConsumer, where the Ada procedure consumer() calls the C function tDemoValProcess(), which in turn calls the VxWorks API malloc().

In a mixed language environment it is also important to understand the performance characteristics of the C and Ada application components in the context of the entire system. This can determine if the task scheduling and interaction of the application is correct, and if it is meeting its performance requirements. In the case of the Ada command and control application, the developer would want to be able to analyze the correct behavior of the Ada application on receipt of data packets from the C-based TCP/IP network stack using the IPC mechanisms provided by the RTOS, and also the throughput latency. This would require the interactions of both C and Ada code with the RTOS to be traceable, which is generally achieved through instrumentation. Because GNAT Ada tasks map directly to VxWorks tasks, the Wind River System Viewer can display

the flow of data packets at system level (as shown in figure 2), and measure the transfer time between the C-based network stack and Ada command and control application.

We have considered some of the driving factors for the development of mixed language device software, the challenges they present and ways they can be overcome. The growth in distributed networked applications may also require the development of applications not simply in mixed languages, but in a mixed OS environment. In the case of the command and control system, a mixed OS configuration, one using VxWorks and Linux, for example, might be utilized to provide hard real-time performance and I/O throughput capabilities respectively. This would require the integration achieved by Wind River, ACT and RTI on VxWorks to be replicated on Linux. The common IDE for both VxWorks and Linux, would boost developers' productivity, as they would not need to master a separate toolset for each OS. ■

Click-for-More

Interested in more information about Wind River's tools, RTOSs and platforms for Defence and Aerospace?

Visit our specific website with links to:

- ▶ Technical Details about VxWorks 6.0
- ▶ Technical Details about RTI Scope Tools
- ▶ Aerospace & Defense platforms from Wind River
- ▶ Wind River Worldwide User Conference

Simply type-in Reader Service #: **585** at Embedded-Control-Europe.com/know-how



