21st Century Ada: Faster, Stronger…and Higher

Still delivering powerful benefits from its original DoD development as Ada 83, as well as extended advantages developed for Ada 95, Ada continues to propel demanding safety-critical, real-time embedded-system programs.

For Reprint Orders Call (949)226-2000 / ©2003 The RTC Group

On August 12, 2002, the past and future of the Ada programming language lifted off in unison from Cape Canaveral's Space Launch Complex 41, guiding a massive Atlas V launch vehicle on its maiden voyage to the stars. Despite being regarded for years as a "defense-only" language due to deep roots in the Department of Defense (DoD), Atlas V and commercial programs worldwide continue to demonstrate the fiscal and technical advantages of Ada over alternative languages when developing safety-critical, real-time embedded system software.

Incorporating numerous systems originally developed for the proven Atlas III, development of the Atlas V began in 1998 to meet the growing needs of the U.S. Air Force Evolved Expendable Launch Vehicle (EELV) program built for International Launch Services (ILS) commercial and government satellite customers worldwide (Figure 1). "What we did moving from Atlas II to Atlas III was to restructure and layer the OS in a more object-oriented fashion. This affected much of the Atlas III software," says Michael Bethancourt, Flight Software Development Lab Manager and Team Leader for the Atlas Flight Software Operating System (OS) at Lockheed Martin Space Systems.

When his team started on Atlas V, they brought forward proven, sophisticated control algorithms and implemented them in a fully object-oriented software structure. The Atlas V relies on primary flight software—as well as other unique, in-house software—all coded in Ada 95. According to Bethancourt, Ada is still the best development language for much of what they do. Modern features critical to building reliable and robust flight software include strong typing, dynamic binding, polymorphism and a secure development environment, which together produce fewer interface errors and easy enforcement of proven development standards.

His 34 years of OS and software design experience and 26 years working with real-time embedded systems form the basis of a firm belief that Ada is still the right language for mission-critical, safety-critical, real-time embedded systems. Bethancourt's assessment correlates with statistics showing Ada use continuing to rise, despite the 1997 lifting of a DoD mandate to develop large military

(Copyright Lockheed-Martin Corporation.)

Figure 1
Ada is at the heart of the software management system on the Atlas V rocket.

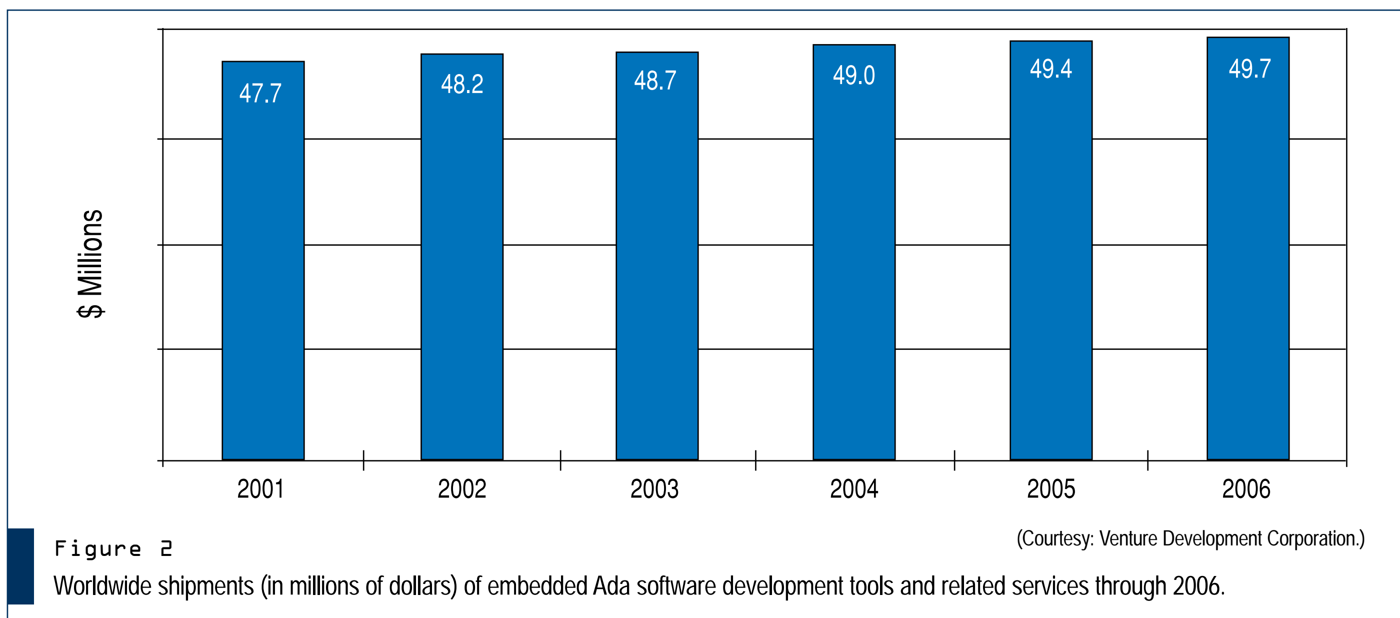

Figure 2

Worldwide shipments (in millions of dollars) of embedded Ada software development tools and related services through 2006.

embedded system software using Ada. In fact, according to the report *The Embedded Software Strategic Market Intelligence Program 2001-2002: Volume V: Ada in Embedded Systems* by Venture Development Corporation, Ada development tools sales continue to increase. (Figure 2).

Protected Typing in Ada 95

Vice President of Technology Joyce Tokar, of Arizona-based DDC-I, the tools provider for the Atlas V program, points out that significant advances in Ada 95 have increased the robust nature of the language, while ANDF-based multi-language tools like her company's SCORE product allow cross-compilation of C/Embedded C++ as well as Ada for the target.

"Ada 95 has extended the Ada 83 strong typing model with additional support for shared variable manipulation through the introduction of the protected type," Tokar explains. "This capability enables the shared use of a variable in a safe manner, even in a multi-process environment. Interrupt handlers are also closely associated with the process model through protected procedures for interrupt handlers. In applications like the Atlas V, shared data collected from a sensor input may be protected by encapsulating the data in a protected object including the appropriate operations."

What is a Protected Type?

The Ada 95 protected type offers programmers a synchronization mechanism to access shared data without using an additional task. The protected type is a data structure defining data components and operations on the defined components. Data shared between multiple tasks may be declared as components of a protected type. A protected type resembles an Ada package in that it has a distinct specification and body. The specification defines the user interface and data components of the protected type. The body provides the implementation.

The specification has a private and a visible part. The private part contains all of the shared data and may contain the specification of private protected operations. The visible part defines the interface through which clients may access, manipulate and modify the private shared data.

There are three protected operations: functions, procedures and entries. Protected functions provide read-only access to the protected data.

```
protected Share_Data is
     function Read return Data_Type;
     procedure Write( I : in Data_Type );
     entry Evaluate;
private
     Data_Item : Data_Type := Init_Val;
     Eval_Data : Boolean := False;
end Share_Data;

protected body Share_Data is
     function Read return Data_Type is
     begin
          return Data_Item;
     end Read;

     procedure Write( I : in Data_Type ) is
     begin
          Data_Item := I;
          Eval_Data := True;
     end Write;

     entry Evaluate when Eval_Data is
     begin
          Evaluate_Data( Data_Item );
          Eval_Data := False;
     end Evaluate;

end Share_Data;
```

Figure 3

The Ada definition of a protected object named Share_Data.

For Reprint Orders Call (949)226-2000 / ©2003 The RTC Group

Protected procedures and entries provide exclusive read-write access to the protected data. Protected entries also have an associated barrier condition that must be true prior to executing the entry body. Each entry has a queue for tasks that must wait until the corresponding entry's barrier condition becomes true.

Objects of a protected type are declared, as needed, throughout an Ada application. Each protected object has an associated lock that must be acquired prior to accessing the protected data through the protected operations. Hence, multiple tasks that share data components of a protected object must first acquire the protected object's lock; this mechanism synchronizes access to the protected data. Upon completion of a protected operation, prior to releasing the lock, entry barrier conditions are checked. If a barrier is now true this process continues until there are no true barriers or all entry queues are empty. At this point the protected object's lock is released.

For example, in Figure 3 the protected object Share_Data provides the user with three operations, Read, Write and Evaluate. These operations provide access to the shared data defined in the private part of the Share_Data specification. When a task needs to read the value of Data_Item it will call the protected function Read. These functions work as follows:

Read: This task must acquire the protected object's lock prior to reading the data. To update the value stored in Data_Item, a task will call the protected procedure Write.

Write: After acquiring the protected object's lock, the task will write the data and set the flag to indicate that there is new data that needs to be evaluated. If a task wants to evaluate the data, it will call the protected entry Evaluate.

Evaluate: If there is new data to evaluate, the task will proceed with the evaluation and then it will clear the evaluate data flag. If the data has already been evaluated, the calling task will wait in the entry queue until new data arrives.

Protected Procedures as Interrupt Handlers

The interrupt handling model in Ada 95 builds upon the protected type by utilizing protected procedures without parameters to implement interrupt handlers. For example, the protected object defined in Figure 3 may be used in conjunction with a protected object that contains an interrupt handler to interface to the sensor providing the data to be shared as defined in Figure 4.

The pragma Attach_Handler associates the procedure Data_Ready with the interrupt defined as Sensor_Interrupt_Id. When this interrupt occurs, the protected procedure calls on the private protected

For Reprint Orders Call (949)226-2000 / ©2003 The RTC Group

procedure Read_Sensor_Data to get the data from the sensor. By defining this operation as a private protected operation, it may not be called by tasks that do not have access rights to the protected object and its data. This ensures that the sensor data will be read without being interrupted by another task trying to access the data or the protected object.

When the sensor data has been retrieved from the sensor, it is written into the shared data area using the protected operation Share_Data.Write. The interrupt handler is complete and ready to receive another interrupt while the new sensor data is available for evaluation and processing by tasks accessing the Share_Data object.

A small, efficient feature of Ada 95 that can solve many synchronization and communication problems on both single and multi-processor architectures, the protected type was designed to serve as a mechanism to support data-oriented synchronization. In addition to being used to define critical regions or implement common real-time paradigms such as semaphores, mutexes and events, use of the protected type replaces instances where tasks are used in Ada 83 with protected objects. Shared data and associated protected operations may be defined within a protected type to create a module that encapsulates the desired real-time abstraction.

Solid Investments in Ada

The DoD maintains an enormous investment in Ada, and major safety-critical embedded systems written in Ada like those in the Atlas will be in service well into the next century. Ada isn't going away in the foreseeable future.

Also providing tools to Sikorsky's Black Hawk helicopter program, Tokar's company is currently supporting Sikorsky's Control Display Unit (CDU) redesign project—for every military and civilian variation in the Black Hawk family—where strategic migration of legacy code and increased hardware standardization are dramatically reducing development costs. "For many years, like the rest of the industry, when a customer requested modifications we would invest two or three years redesigning specific systems for their application. Then another customer would say, 'I want this,' and we'd repeat the process," says Bret Stockreef, a Senior Systems Software Engineer at Sikorsky Aircraft's Stratford, Connecticut headquarters.

The program reached a point where adding new functions to the old system was impractical. The boxes were too heavy, too expensive and short on processor horsepower. The proposal: a new breed of CDU: a modular unit alongside an entirely new Multi-Function Display (MFD) from Rockwell-Collins, who is also writing the primary flight display software while Stockreef's team serves as the overall project integrator and lead software developer.

For Reprint Orders Call (949)226-2000 / ©2003 The RTC Group

Built around an 80486-based processor, which Sikorsky uses for other programs, the most practical and cost-effective approach has been to reuse available legacy code and port it using their existing DDC-I Ada Compiler System (DACS) as the software development environment.

The results have been as solid as predicted, reducing the number of system boxes from twelve to six, generating major real estate and weight savings. The program's success is also spurring increased movement toward system standardization across much of the Sikorsky product line.

While some C++ is being used, the bulk of the embedded systems code is in Ada, which he sees as a large advantage when it comes to porting code to different processors. Sikorsky's choice to continue with Ada for the redesigned Black Hawk CDU sends a clear signal that Ada remains the language of choice for large-scale, safety-critical embedded system development. Despite predictions of its

```
protected Sensor_Data is
          procedure Data_Ready;
          pragma Attach_Handler( Data_Ready, Sensor_Interrupt_Id);
     private
          procedure Read_Sensor_Data;
          Data_Item : Data_Type := 0;
     end Sensor_Data;

     protected body Sensor_Data is
          procedure Data_Ready is
          begin
               Read_Sensor_Data;
               Share_Data.Write( Data_Item );
          end Data_Ready;

          procedure Read_Sensor_Data is
          begin
               Get_Data_From_Sensor( Data_Item );
          end Read_Sensor_Data;
     end Sensor_Data;
```

Figure 4

Incorporating interrupts into a protected object. Procedure Data_ready is designated as the interrupt handler for the sensor interrupt Sensor_Interrupt_Id.

For Reprint Orders Call (949)226-2000 / ©2003 The RTC Group

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions(
        Max_Entry_Queue_Length => 1,
        Max_Protected_Entries => 1,
        Max_Task_Entries => 0,
        No_Abort_Statements,
        No_Asynchronous_Control,
        No_Calendar,
        No_Dynamic_Attachment,
        No_Dynamic_Priorities,
        No_Implicit_Heap_Allocations,
        No_Local_Protected_Objects,
        No_Protected_Type_Allocators,
        No_Relative_Delay,
        No_Requeue_Statements,
        No_Select_Statements,
        No_Task_Allocators,
        No_Task_Attributes_Package,
        No_Task_Hierarchy,
        No_Task_Termination,
        Simple_Barriers);
```

Where **pragma** Detect_Blocking is defined as follows:

The bounded error that is the invocation of one of the following potentially blocking operations during a protected action shall be detected:

- a protected entry_call_statement
- a delay_until_statement
- a call to a language-defined subprogram that is potentially blocking, for example Ada.Synchronous_Task_Control.Suspend_Until_True

Note the detection of these bounded error cases results in Program_Error being raised ([RM] 9.5.1 (17)). Potentially blocking operations that occur in a foreign language domain need not be detected.

Figure 5a

This is the explanation of the behavior of pragma profile for the Ravenscar execution time profile.

For Reprint Orders Call (949)226-2000 / ©2003 The RTC Group

demise after the mandate for defense contracts was lifted, Ada usage is increasing, as is industry recognition of Ada advantages in code reliability, reusability, readability and portability.

Extensions to Ada 95

While Ada 83 remains deeply embedded in many applications, Ada 95 was designed to address the needs of a number of different application domains. The annexes attached to the language definition categorize additional functionality into several domains: Systems Programming, Real-Time Programming, Distributed Programming, Information Systems, Numerics, and Safety and Security.

The Real-Time and Safety-Critical community embraced these enhancements and began to take the language to the next stage of functionality. The International Real-Time Ada Working Group (IRTAW) evaluated the restrictions on programming execution environments available through the use of pragma restrictions, working to define an execution environment allowing users to take advantage of the Ada's tasking capabilities, while at the same time imposing appropriate restrictions to define a deterministic scheduling model usable for implementing real-time, safety-critical environments.

The IRTAW consensus on the model came at a group meeting in Ravenscar, UK, which gave the profile its name: Ravenscar. The next step was to add the profile to the language definition and make it available to all users. Over a few years and many meetings, the language lawyers and the real-time experts agreed on the likelihood that there will be more than one execution time profile that users may want to use in their application developments.

The team agreed to introduce a new pragma in the Real-Time Systems Annex called pragma Profile, which may be viewed as a short-hand representation of the restrictions and scheduling characteristics associated with an execution time profile. The Ravenscar profile is specified in a program using the configuration pragma as follows:

```
pragma Profile( Ravenscar );
```

This has the effect of telling the compilation system the configuration to use for compilation and execution of this application has the characteristics listed in Figures 5a and 5b. These new features have been proposed to ISO for inclusion in the language definition as an addendum.

Building on the concept of execution environment definition, DDC-I has introduced a new parameter to the **pragma** Profile called ARINC-653-SAFE. This profile restricts the Ada execution environment for compatibility with an ARINC-653 operating system. Using this configuration, the Ada application may not include tasks, exceptions, interrupts, or automatic dynamic storage allocation as all of these facilities are available in the

operating system. A user may build an application using this pragma to develop safety-critical applications taking advantage of the acknowledged safety characteristics of the ARINC 653 definition.

Note that there are several new restrictions introduced in this definition. These include the following:

No_Calendar -- There are no semantic dependencies on package Ada.Calendar.
No_Dynamic_Attachment—There is no call to any of the operations defined in package Ada.Interrupts (Is_Reserved,Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, Reference).
No_Local_Protected_Objects—Protected objects shall be declared only at library-level.
No_Protected_Type_Allocators—There are no allocators for protected types or types containing protected type components.
No_Relative_Delay—There are no delay_relative_statements.
No_Requeue_Statements—There are no requeue_statements.
No_Select_Statements -- There are no select_statements.
No_Task_Attributes_Package—There are no semantic dependencies on package Ada.Task_Attributes.
No_Task_Termination—All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate.
Simple_Barriers—The Boolean expression in an entry barrier shall be either a static Boolean expression or a Boolean component of the enclosing protected object.
Max_Entry_Queue_Length—Max_Entry_Queue_Length defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error at the point of the call. For the Ravenscar Profile, the value of this restriction is 1. Note that the effect of this restriction applies only to protected entry queues due to the accompanying restriction of Max_Task_Entries => 0.

Figure 5b
An explanation of the pragma restrictions found in Figure 5a.

Embedded Systems, Embedded Costs

The most significant benefit Ada proffers quite often isn't even considered when the choice between development languages is made. When embedded computer systems are evaluated from a total lifecycle perspective, research has determined that sixty to eighty percent of the program costs occur after development and implementation, which means the maintenance phase accounts for the lion's share of costs.

Since procurement decisions are typically driven by development costs and maintenance is often handled as a separate contract on many projects, this creates an artificial division in the lifecycle making it far too easy to overlook maintenance. A decision based on the first phase costs is only considering twenty to forty percent of the total possible project expenses.

Good software engineering practices were a founding principle when Ada was developed, and it remains the only internationally standardized programming language (ISO) designed to address large project applications and still deliver similar performance efficiencies to smaller scale, unit cost-driven programs like the Raytheon-built Joint Stand-Off Weapon (JSOW). Using software originally developed by Texas Instruments, engineers at Raytheon currently maintain the Guidance Electronics Unit (GEU) of the JSOW, one of the most valuable medium-range tactical weapons in the United States' high-tech arsenal.

"The GEU is the operational core of the JSOW, responsible for guidance, navigation and control of the munition from launch to strike. Every system on the weapon in some way, shape or form communicates with the GEU," explains Guy Yeager, a principal software engineer and software team lead for the JSOW/GEU program at Raytheon.

Required mission capabilities for the JSOW created significant challenges, particularly with respect to aggressive cost goals. Unlike some DoD programs using an "affordability" label to satisfy critics, strict JSOW goals drove a number of key design and production decisions to produce components like the GEU at low cost.

Main software development for the GEU was originally done at TI Systems. Yeager's team is currently porting old assembly code to Ada for a '486 processor. A strong advocate of Ada, especially for safety-critical applications, Yeager knows first-hand how robust exception handling, multi-tasking capability and built-in run-time contribute to cost-effective software programming for real-time embedded systems.

"Ada's strong typing and other characteristics seem very limiting to programmers accustomed to languages like C++, but if it's code for a safety-critical system, I don't want creativity. Ada is much more self-regulating in that respect. Even if you know what you're doing, it has intelligent checks integrated into the language that ultimately produce fewer mistakes in the final code," Yeager concludes. ▮▮

DDC-I
Phoenix, AZ.
(602) 275-7172.
[www.ddci.com].

Venture Development Corporation
Natick, MA.
(508)653-9000.
[www.vdc-corp.com].

For Reprint Orders Call (949)226-2000 / ©2003 The RTC Group