N89-16363

# Interfacing Ada* and Other Languages

Paul Baffes and Brian West

Intermetrics Inc.

## INTRODUCTION

The Department Of Defence has mandated the use of Ada on upcoming projects involving embedded system software. NASA has also indicated that Ada has been baselined for the Space Station project. Both of these decisions will require the contractor community to transition from their current non-Ada programming environments. Existing software, that is proven and validated, will most likely continue to be used during the transition period. During this period new Ada programs and existing programs in other languages may need to be interfaced.

A myriad of possibilities exits for the solution of the transition problem. One set of solutions deals with translating the source code of the other language into Ada source code or the intermediate langauge used by the chosen Ada compiler. Another solution involves a special interface subroutine that switches from the Ada run time environment to the run time environment of the other language. The latter solution will be examined.

The above mentioned non-Ada programming environments consist of many different programming languages like FORTRAN, PASCAL and HAL/S. While each of these languages is unique, they are all members of the ALGOL family of programming languages and share many implementation characteristics. Therefore, an interface subroutine can be analyzed for any one of these language and the results can then be extended to the remaining languages. HAL/S was chosen for this examination.

The HAL/S 360 compiler which runs under the IBM MVS Operating System and the Ada compiler which runs under the IBM VM/SL Operating System were selected for this study. The primary criteria for the selection of the HAL/S and Ada compilers was that they were hosted on the same machine architecture. Both compilers were developed by Intermetrics.

## GENERAL OVERVIEW OF THE PROBLEM

In this section, the general issues involved in interfacing any two different high-level languages will be explored. This explanation will outline the direction taken by the following sections, and should help in providing an overview of the intricacies involved in such an integration.

### The Language Environment

Along with any Programming Language comes a set of assumptions under which that language is run. This set of assumptions can be called the environment of that language, and consists of both algorithms and data structures. While these may cover a variety of subject matter, most Algol-like language environments can be understood through a few basic ideas.

The first of these basic ideas is known as a run-time stack. In general, the run-time stack is used to keep track of local data and register contents across procedure calls as the program is being executed. In this way, the integrity of a procedure can be maintained while control is passed to a subprocedure, and then restored when the subprocedure returns.

Another of these basic ideas concerns the internal representation of data. For example, one language environment might use a signed magnitude representation for integers while another may use twos complement form. Naturally such details are not an interfacing concern when a calling procedure and the called subprocedure are written in the same language. However, care must be taken to ensure that these internal representations are identical when two different languages are involved.

A final basic concept involves the managing of run-time errors, commonly known as exception handling. Most often, a language environment will provide a large collection of procedures called a run-time

• Ada is a Registered Trademark of the U.S. Department of Defence (Ada Joint Program Office) library which contains the mechanisms for dealing with these errors. However, since two different languages will use two separate run-time libraries, an error occurring in a subprocedure of one language would probably not be understood by the calling procedure of the other language. This would prevent the calling procedure from responding to the error in a proper manner.

### The Basic Interface

In light of the previous discussion, the interface between two distinct languages becomes a matter of switching environments. To accomplish this at run-time, a special linking subroutine would need to be invoked from the run-time library of the calling procedure. This subroutine would provide the mechanism for saving the present environment of the calling procedure and initiating the new environment. In turn, upon termination of the called subprocedure, this subroutine would regain control and reinstate the old environment.

Ada provides an instrument for interfacing with other languages called the PRAGMA INTERFACE directive. The sections that follow take into consideration the details necessary for implementing this mechanism.

## COMPARISON OF HAL/S AND ADA ENVIRONMENTS

### Parameter Passing

Almost every subroutine makes use of parameter passing, whether it accepts some value or values as input, or produces some output, or both. This process of exchanging information between procedures is part of a language environment and thus will most likely vary from one language to another. In regards to the HAL/S and Ada compilers cited, the discrepancies are dramatic.

### The Procedure Call

In most cases, all parameters are passed through registers to the called subroutine. However, when there are not enough registers for all of the parameters, another method is pursued. This method usually involves placing the parameter in temporary storage and passing the address of this location instead to the called subprocedure.

Both HAL/S and Ada comply with the conventions outlined above. However, the specific registers used by the two languages to accomplish these standards are not the same. For example HAL/S and Ada use a different register for addressing the temporary storage area where the overflow parameters are stored. In addition, Ada may store its parameters in more that one place, depending on whether or not they were dynamically allocated. Any interfacing subroutine would have to map one set of register conventions to the other and also be aware of the different locations where the overflow parameters are stored.

### The Function Call

Functions, unlike procedures, return a value to the calling procedure. This value is returned via the use of a register. As was true with parameter passing, this register may contain either the actual value or a reference to the location where the value is saved. Again, each language will use different conventions for returning this value. In fact, the HAL/S and Ada compilers cited utilize different registers for this purpose.

### Data Representation

A problem related to parameter passing arises from how each language chooses to represent its data types. There are a variety of factors involved in data representation including the number of bytes used, indexing schemes, value restrictions, and the algorithm employed for packing the representations to save space. Since each language will differ in its methods of representation, some scheme for converting data between representations would have to be implemented before any interfacing would be possible.

### The Run-Time Stack

The objective of the run-time stack is to keep track of the flow of a program during its execution; namely, to record the dynamic nesting of the called procedures. To accomplish this, the run-time stack contains the information necessary to describe the state of the program at any point during its execution. The particulars of the run-time stack are also implementation dependent.

### The HAL/S Run-Time Stack

HAL/S has a very straightforward approach to its run-time stack design. Its run-time stack is divided into "stack frames," one for each procedure currently being executed. These stack frames are further divided into two sections. The first of these is of a constant size and contains the following: a register save area, an area for the current code base, and a workspace for exception handling. The second section is of variable size and is used to store the procedure's local and temporary variables. The uses of these two sections are explained below.

When a subprocedure is called, a new stack frame is created and placed onto the stack. The contents of all the calling procedure's registers are then stored in the register save area of this new stack frame. In turn, when the called subprocedure returns control to the calling procedure these stored register contents are replaced into their appropriate registers. In this way, the calling procedure's register contents are not violated by the called subprocedure. The remaining fixed portion of the stack frame provides the procedure with run-time control information. This information includes: the location of the first executable instruction for the current procedure, a temporary workspace, and a link to the error library.

The second section of the run-time stack is left for the local and temporary variables of the subprocedure being executed. The size of this section varies from procedure to procedure depending on each procedure's number of local and temporary variables. The size of each procedure stack frame, however, is determined at compile time. So while stack frame sizes may vary from procedure to procedure, each procedure's particular stack frame size is fixed at execution time.

### The Real Time Executive

Real time executives are used to synchronize and allow communication between two independently executing programs. Any program which depends upon some real world event will depend upon a real time executive for proper execution. The internal mechanisms which implement real time executives are nontrivial and vary widely among the languages that provide real time features. Although HAL/S and Ada both have a powerful set of real time executive tools, these tools are unalike and they require different approachs by the applications programmer for solving real time problems. Because their sets of real time executives are not the same, the HAL/S and Ada language environments will incorporate different implementation schemes. To interface these two sets of real time executives would pose an extremely involved challenge.

### The Run-Time Library

Every language has a set of primitive utilities which it uses repetitively. This set of utilities is commonly called the run-time library. The run-time library is automatically linked with the program's object module before execution. As a result, every procedure or subprocedure of the program can employ any routine provided by the run-time library.

Of course, each language will have a unique run-time library. One of the more significant problems arising from this concerns error handling. When an error occurs during the execution of a program, the problem is most often managed by a routine in the run-time library. If this were to happen in a called subprocedure of a different language, there would be no guarantee that the process used to handle the error would be understood by the calling procedure. This problem is important because some errors may require termination of the program. Thus, if the called subprocedure were to force termination before returning control, the calling procedure would not be able to exit in a graceful manner. This could result in a loss of pertinent information. Additionally, similar errors may be handled with different levels of severity by different language environments. In particular, what may cause a HAL/S program to terminate may only raise an exception in an Ada program. This presents a formidable problem for the interfacing subroutine.

## SUMMARY

### Overview of an Interface Subroutine

The interface subroutine would operate in a straightforward manner. The routine would first load the passed parameters into the registers. A parameter would be passed either by its actual value or by a pointer, a machine address. Verifying that the parameters were passed in the correct format would be the responsibility of the Ada applications programmer.

The next step in the interface subroutine would be to initialize a new HAL/S stack frame and branch to the entry point of the HAL/S executable code. During execution, calls to the HAL/S run-time library may be made. To guarantee proper execution, the Ada applications programmer would have to include all needed HAL/S run-time library routines in the load module. Upon finishing the normal execution of the HAL/S code, a branch would be made back to the linking subroutine and the old stack frame would be popped off the stack.

Finally, the interface subroutine would remove the passed parameters from the registers. Before assigning these values to their appropriate memory locations, constraint checking should be performed. Any constraint violation should raise an exception and the corresponding exception handler should be invoked at that time.

### Restrictions on the Interface

Restrictions, unfortunately, would have to be placed on the called HAL/S procedure. The interface subroutine would resolve as many of the differences between the two run time environments as possible. Those differences which could not be resolved would result in restrictions on the interface.

One restriction would involve the way

errors are handled. Run-time errors in the HAL/S executable code will not raise exceptions when they occur. Some of these exceptions could be raised by the interface subroutine when constraint checking is done. Other run-time errors in the HAL/S code would go unnoticed and the subsequent execution would be indeterminant. Note that the called HAL/S procedure would have to have an appropriate ON ERROR IGNORE statement or else the HAL/S code could make an unsupported operating system call.

Another restriction concerns the visibility of variables. At the point of the HAL/S procedure call in the Ada program, some of the declared variables may have visibility. While an Ada procedure called from the same point would be able to access these visible variables, the HAL/S procedure could not. Succintly, the only way the Ada program and the HAL/S procedure could communicate would be via the passed parameters.

Yet another restriction would be that the HAL/S procedure could not invoke real time executives. Additional restrictions may be to limit the use of Ada real time executives and to circumscribe the use of I/O in the HAL/S procedure. The above two proposed limitations need further investigation.

## CONCLUSION

Interfacing two separately developed compilers is a complex task. The complexity arises because very few design standards exist for compiler development. This, coupled with the many complicated design decisions inherent in compiler construction, virtually guarantees noncompatibility. The interface subroutine which would link the two different run time environments would resolve as many of the dissimilarities as possible. The differences that could not be resolved would be responsible for the restrictions placed on the interface. Albeit restrictions would exist, the resulting interface may be well worthwhile.

## BIBLIOGRAPHY

Aho, Alfred V. and Ullman, Jeffrey D., Principles of Compiler Design. Reading: Addison-Wesley Publishing Company, 1977.

Booch, Grady. Software Engineering with Ada. Menlo Park: The Benjamin/Cummings Publishing Co. Inc., 1983.

Ryer, Michael J. Programming in HAL/S. Cambridge: Intermetrics Inc., 1980.

Computer Program Development Specification for the Ada Integerated Environment. IR-MA-300. Cambridge: Intermetrics Inc., 1984.

HAL/S-360 Compiler System Specification. IR-60-07. Cambridge: Intermetrics Inc., 1981.