Using preliminary Ada in a process control application

by M. E. GORDON and W. B. ROBINSON
The Foxboro Company
Foxboro, Massachusetts

INTRODUCTION

This section contains background information on the Ada language definition process, an introduction to features of Ada, and an overview of the Model Controller Operating System (MCOS), which was coded in Ada. More detailed information on Ada can be found in The Ada Language Reference Manual,1 The Ada Language Rationale,2 and Programming with Ada.3

An Ada introduction

Background

Ada is the programming language being developed under the auspices of the Department of Defense. The language development has extended over a period of several years, from requirements specification (Strawman, Woodenman, Tinman, Ironman, Steelman) to language definition. Accompanying the language development is the specification of a language support environment (Sandman, Pebbleman, Stoneman), which is progressing closely after Ada itself.

Ada is being considered as a standard language for process control by the Long Term Programming Languages (LTPL) Committee of the Purdue Workshop and the European LTPL Committee. Both ANSI and ISO standardization efforts are being initiated.

Currently in a formative stage, Ada is undergoing revisions in response to a Test and Evaluation (T&E) of the language. The programming project described in this paper was undertaken as part of the T&E review, and is based on the preliminary Ada language definition of June 1979. Since there are, as yet, no Ada compilers available, the evaluation is static. However, the Ada features described below are intrinsic to the conceptual model of the language and are unlikely to change.

This section will discuss how Ada language features support the modern language concepts of: (1) data abstraction, (2) modularity, (3) encapsulation, (4) concurrent programming, (5) machine independence, and (6) orthogonality and extensibility.

In later sections, it will be shown how such language features may be applied to a typical programming problem in a process control systems application.

Data abstraction

Data abstraction is supported in Ada by programmer-definable data types. Type declarations collect knowledge of common properties of objects in one place, thereby facilitating software maintenance. The principal advantage is greater software reliability, because the programmer's code is closer to the expression and solution of the applications problem.

Ada's strong typing is based on name equivalence, rather than structural equivalence. No implicit type conversions are allowed. Explicit conversions can be used in some cases, for instance, in converting numeric types. In other cases, Ada provides the UNSAFE_CONVERSION function as an escape hatch.

For programmers accustomed to creating variables on-the-fly (as in FORTRAN), Ada's requirements for declaration of variables and types may seem overly restrictive at first. With proper use of Ada's data facilities, however, the benefits far outweigh the constraints.

Modularity and program structure

Modularity is supported not only by traditional subprograms (procedures and functions), but also by Ada modules (packages and tasks). Although the overall structure of an Ada program follows the conventional block structure of ALGOL 60, it differs in that modules may be separately compiled and arbitrarily included at various levels of the program hierarchy. Ada offers control over visibility and scope through restricted clauses, which may override inheritance rules of module nesting. Importing is done by program unit name, not by object name.

Encapsulation

Package modules are the cornerstone of the language. Through them, encapsulation of data with their associated operations is possible. Other uses of packages are grouping related procedures together and forming a collection of related types and data objects.

Packages consist of a specification part and an optional package body. The specification part of a package contains

597
forces the realization of encapsulation via modularization. Above, allow the systems programmer and, later, the main­language such as Ada. Ada language features, as described like FORTRAN (developed in 1954). Actual machine-de­pends (analog) control diagrams. It was conceived as a typ­ical example of a small, real-time system providing these main features: (1) communications interface to host com­puters, (2) control algorithms and control database based on block diagrams, and (3) interface to process input/output components. Each of these is discussed in detail later.

The Model Controller Operating System (MCOS) supports these features with the multi-tasking facilities provided by Ada. Figure 1 shows a simplified sketch of the model controller.

One approach to reliability in digital control systems is redundancy. In the model controller, this is reflected in the dual-controller, dual-port, dual-link architecture. In the event of a failure of any main system component, the backup component is automatically activated. The process database is regularly (every control cycle) transferred to the backup, or tracking, controller via the shared data buffer. With ap­propriate checks against contaminating its database, the tracking controller is prepared to take control with a very recent database copy. The two controllers also share access to the process I/O bus. The hardware redundancy is com­plemented by a software redundancy: both controllers have their own copy of the MCOS software, which will be de­scribed in the following sections.

The communications link allows host computers to con­figure or to change the configuration of the control database, to read or write particular values in the database, and to monitor or change the status of each of the controllers.

The process I/O bus, which can be accessed by only one controller at a time, allows the controllers to read or write analog and digital input/output devices.

In the following sections, it will be shown how Ada may be applied properly to an embedded operating system for process control. In particular, the remaining discussion will illustrate how Ada influences all stages of the software de­velopment process, from initial design to final implementa­tion.

The model controller operating system

The model controller is a system that allows digital im­plementation of multiloop control systems based on traditional (analog) control diagrams. It was conceived as a typ­ical example of a small, real-time system providing these main features: (1) communications interface to host com­puters, (2) control algorithms and control database based on block diagrams, and (3) interface to process input/output components. Each of these is discussed in detail later.

The Model Controller Operating System (MCOS) supports these features with the multi-tasking facilities provided by Ada. Figure 1 shows a simplified sketch of the model controller.

One approach to reliability in digital control systems is redundancy. In the model controller, this is reflected in the dual-controller, dual-port, dual-link architecture. In the event of a failure of any main system component, the backup component is automatically activated. The process database is regularly (every control cycle) transferred to the backup, or tracking, controller via the shared data buffer. With ap­propriate checks against contaminating its database, the tracking controller is prepared to take control with a very recent database copy. The two controllers also share access to the process I/O bus. The hardware redundancy is com­plemented by a software redundancy: both controllers have their own copy of the MCOS software, which will be de­scribed in the following sections.

The communications link allows host computers to con­figure or to change the configuration of the control database, to read or write particular values in the database, and to monitor or change the status of each of the controllers.

The process I/O bus, which can be accessed by only one controller at a time, allows the controllers to read or write analog and digital input/output devices.

Orthogonality and extensibility

Ada offers language primitives and rules for combining them (orthogonality*) to build specialized structures and functions. This principle of orthogonality and the related principle of extensibility are demonstrated by the I/O pack­ages, which are written in Ada. Although lacking some key features (e.g., variable-length strings, bit types), Ada pro­vides the tools to create them. This is a critical requirement for a systems implementation language.

Such freedom in a language exacts a price, that price being the responsibility for a programming discipline. When properly used, Ada can be a powerful tool for both systems and applications software development. However, without the appropriate programming methodology, the benefits of Ada are easily lost. For example, arbitrary use of the exit state­ment to create unstructured loop constructs would be a mis­use of the language.

Concurrent programming

Tasks, similar to packages in format, are the means for implementing concurrent processes. Unique to the Ada task­ing mechanism is the rendezvous concept, which serves the dual functions of synchronization and communication in a parallel processing environment. Three controversial fea­tures related to Ada's tasking are scheduling, interrupt han­dling, and access to shared data. They are likely to change in the final language definition.

Machine independence

Machine independence is the ultimate goal of a high-level language such as Ada. Ada language features, as described above, allow the systems programmer and, later, the main­tenance programmer to operate at a level which is closer to the system specification than is possible with older languages. Therefore, it is not necessary to wait until link time to discover most interface errors. This check­ing will be accomplished through a library management sys­tem, which is a requirement of the Ada support environment.

In the following sections, it will be shown how Ada may be applied properly to an embedded operating system for process control. In particular, the remaining discussion will illustrate how Ada influences all stages of the software de­velopment process, from initial design to final implementa­tion.

The model controller operating system

The model controller is a system that allows digital im­plementation of multiloop control systems based on traditional (analog) control diagrams. It was conceived as a typ­ical example of a small, real-time system providing these main features: (1) communications interface to host com­puters, (2) control algorithms and control database based on block diagrams, and (3) interface to process input/output components. Each of these is discussed in detail later.

The Model Controller Operating System (MCOS) supports these features with the multi-tasking facilities provided by Ada. Figure 1 shows a simplified sketch of the model controller.

One approach to reliability in digital control systems is redundancy. In the model controller, this is reflected in the dual-controller, dual-port, dual-link architecture. In the event of a failure of any main system component, the backup component is automatically activated. The process database is regularly (every control cycle) transferred to the backup, or tracking, controller via the shared data buffer. With ap­propriate checks against contaminating its database, the tracking controller is prepared to take control with a very recent database copy. The two controllers also share access to the process I/O bus. The hardware redundancy is com­plemented by a software redundancy: both controllers have their own copy of the MCOS software, which will be de­scribed in the following sections.

The communications link allows host computers to con­figure or to change the configuration of the control database, to read or write particular values in the database, and to monitor or change the status of each of the controllers.

The process I/O bus, which can be accessed by only one controller at a time, allows the controllers to read or write analog and digital input/output devices.
Overview of the operation sequence

At every quarter second clock interrupt, each controller begins by checking its mode. Typically one is on control, the other tracking. The lead controller processes all control blocks in the following manner: All process I/O components are read, conditioned, and stored in a table.

For each block: (1) appropriate parameters (such as measurement or set point) are updated; (2) the algorithm for each block is executed; (3) if needed, an output is sent from the block to the appropriate component.

After all blocks are processed, the database is written into the buffer.

The tracking controller copies the data buffer into its own database.

At any time, the system might receive an interrupt on the communications channel. Typical communications messages supported are: (1) secure/release, (2) read/write database, and (3) read/write controller status.

When neither the control processing nor the communications task is active, a security task executes, monitoring the health of the system.

SYSTEM DESIGN WITH ADA

This section contains discussions of some of the design decisions made and relates them to the facilities of Ada for modularization and for representation of data. The design issues are: (1) modularity and program structure, (2) data structures and representations, (3) exception handling, (4) scheduling, and (5) interrupt handling.

Modularity and program structure

The overall program structure of the Model Controller Operating System (MCOS) is illustrated in Figure 2a. The main procedure contains the module specifications of the three primary tasks: communications, executive, and security. The function of the main procedure is to initiate the three tasks. The code for the main procedure is straightforward, as shown in Figure 2b.

Each task body is a separate compilation unit. By separating the task body from its specification, implementation details may be changed without necessitating a recompilation of the main procedure (assuming the interface remains the same). Ada's separate compilation facility was used extensively in the design of MCOS, to take advantage of the logical interface—physical implementation separation. This facility supports top-down design in that the logical interfaces may be defined first, with the implementation stubs developed later. Another advantage of the separation principle is that it streamlines the code, improving the readability of Ada program units.

Within each task body, the specifications for its internal packages are local, while their bodies are separate. The tasks and their primary functions are: (1) communications task: handle port interrupts; (2) executive task: process control blocks (if main controller) or track the database (if backup controller); and (3) security task: perform various software checks during any idle time.

The communications and executive tasks are discussed more fully in later sections.

The final two main modules of MCOS are packages. The database package and status_manager package are separate compilation units in the MCOS program library, and are independent of the overall block structure. The purpose of the status_manager package is to encapsulate global type and data declarations with their associated operations. The database package, on the other hand, contains related types and objects for global use and a synchronizing semaphore task. The two packages are self-sufficient in that they see no other units in the program library. Other program units which require access to the global data make it visible by including the package name in a visibility list. Therefore, importing of global data is done on a module basis, rather than on a single variable basis. The global packages may be imported at any level of the block structure hierarchy.

As demonstrated in Figure 3c, good programming practice dictates explicit import lists of objects to be listed in comments. Without such information, the code is virtually unreadable, since it is not apparent which data objects are being referenced. Rather than burden the programmer (who may be inconsistent and/or error-prone), such information could be generated automatically by the compiler or text editor.

It must be noted that the MCOS program structure presented here was achieved through several iterations. Because of the novel interplay between traditional block structure and separately compiled modules, classic rules-of-thumb for structured design were not directly applicable. Ada is a language without a history and, consequently, with-
out a refined programming methodology. Proper use of Ada features was discovered partly through trial-and-error. A good program structure for MCOS was achieved by following certain guidelines: (1) control visibility as much as possible by using traditional block structure; (2) within the general block structure of the program, textually include only the module specifications; use stubs for module bodies, which should be developed and compiled separately; (3) reserve use of separately compiled modules for global library packages; (4) whenever feasible, compile the specification part and the implementation part separately to reduce re-compilation dependencies.

Data structures and representations

To quote Wirth,5 "The choice of (abstract) representation of data is often a fairly difficult one, and it is not uniquely determined by the facilities available. It must always be taken in light of the operations to be performed on the data." In our model system, there are several instances where the choice of abstract representation was complicated by the fact that two different tasks required access to the same data objects. In another instance, to be described later, the strong type restraints of Ada had to be circumvented to allow a more flexible approach to handling raw data for input and output.

The largest data object, the array of control blocks, is considered in detail below. The design of data structures proceeded in parallel with the overall module design.

Shared data

The critical issue here is the need to provide adequate protection of the database, which can be accessed by both the communications task and the control executive task.

In MCOS, total encapsulation of the database was rejected for two reasons: first and foremost, the number of transactions using the database is prohibitively large, and using specialized interfaces would degrade performance. In particular, the communications task has a 10 millisecond timeout period and thus cannot be delayed too long by control processing. Secondly, a small dedicated system does not require the level of protection encapsulation offers. The communications task performs sufficient validity checks to protect the database in transactions with host computers.

Although encapsulation of the database was rejected, mutually exclusive access to the database must be provided so that consistent data is used by the tasks sharing access. The simplest solution, and the one finally used, employs Ada's generic semaphore task to create a critical region, during which only one task can access the database. It may be that this solution is too inefficient, in which case an equivalent solution would have to be implemented in machine code.

By way of contrast, the package status_manager, described below, is an example of the use of the package structure for encapsulating data objects and the operations performed on them. The brief duration of the exclusive read/write access to the status registers makes encapsulation feasible.

Physical representations of data objects

Since the object machine for the Model Controller Operating System is predicated as having a small memory, space issues are important. Often a record has several components. To allow the compiler to assign the storage for these may use more storage than is desirable. Ada provides several representation facilities for records1 (Chapter 13): One can specify the number of bits to be used representing objects of a given type, specify that the compiler is to use packing, or specify actual word and bit layouts for components of a record. All of these facilities were used in the MCOS exercise.

Exception handling

In any real-time system, security and robustness are essential. Exceptional conditions, such as overflow during a calculation, should not cause a system crash, but must be handled in a meaningful way so the system can recover and continue processing. Ada provides an exception handling facility which appears to be adequate, although sometimes cumbersome11(11). Examples of the use of exception handlers are given in later sections.

Scheduling

The strategy for scheduling the tasks and procedures in a real-time system must be carefully thought out and carefully implemented. In the MCOS, the communications task must execute immediately upon receipt of an interrupt. Other scheduling requirements are somewhat "softer." However, it is in the area of scheduling that Ada seems to have the greatest weakness. According to the LRM1 (9.8), "The language does not specify when a scheduling decision is made; for example, a round-robin time-sliced strategy is acceptable." There is a language-defined priority attribute for tasks which can be used in scheduling decisions. However, there seems to be little in the language to facilitate the design and implementation of a scheduling algorithm.

MCOS requires a scheduler with the following properties: (1) the scheduler is to be invoked when a new task enters the ready queue, in particular when an entry call or interrupt occurs; (2) when the scheduler is invoked, ready queues are examined and tasks with the higher priorities are executed first.

Interrupt Handling

Interrupt handling is another area where Ada, as currently defined, falls far short of the mark. This is a critical area for real-time applications, such as process control, in which cer-
tain hardware interrupts demand immediate, and uninter-
rupt ed, service. In MCOS, for example, a communications
interrupt requires that the message be processed and a reply
sent within 10 msec, otherwise a timeout occurs, putting the
controller on standby. Unfortunately, there is no way to
guarantee dedicated resources to a high-priority interrupt in
Ada.
Ada does not distinguish between hardware interrupts and
software signaling between tasks. An interrupt is mapped
onto a rendezvous entry, via a representation specification.
Yet, there is no way to indicate the urgency of an interrupt
on the entry queue. Task priorities only determine which of
several tasks waiting on the ready queue will be serviced
next. However, entry queues are handled strictly on a first-
in, first-out basis.
The Ada language design team has recognized the inade-
quacy of the current interrupt handling mechanism. It is
expected that the problem will be rectified in the final lan-
guage definition (June 1980). Therefore, in the MCOS pro-
gramming exercise, the communications interrupt was coded
in Ada for illustration purposes, with no assertion of cor-
rectness.

ADA IMPLEMENTATION OF MCOS

This section provides a more detailed view of some of the
issues that were raised earlier. In particular, using Ada in
the actual implementation of MCOS is discussed with regard
to the database, control processing, and communications
processing.

The database

Control blocks

The primary function of the model controller is performed
through the control blocks and their corresponding algo-
rithms. In a typical process control application, several types
of control algorithms, such as proportional-integral-derivati-
ve (pid), lead-lag (llag), etc., would be used. MCOS has
the pid, llag, nonlinear (nonl), and digital input (din) algo-
rithms as a suitable cross-section. Each system, however,
can have up to 32 control blocks, of which an arbitrary num-
ber can be pid, an arbitrary number can be llag, and so on.
In a particular control scheme, blocks can be intercon-
nected, can obtain inputs from process I/O devices, and can
generate outputs for process I/O devices. A sample config-
uration is shown in Figure 3a. Because of this intercon-
nectability, blocks are processed in sequence. In general, a
block will obtain inputs only from blocks that have been
processed before it. This ordering stems from the traditional
digital implementation of continuous analog control.
A standard assembly language or FORTRAN implementa-
tion of such a system would have to treat the 32 blocks
as a massive array of words of undifferentiated type, and
the layout of parameters within different block types would
be contained in an external document, presumably a system

specification. Hence, the meaning of a particular word in the
database would be obscured and opportunities for errors by
both original writers and later maintainers is increased.
With the tools of Ada or any other sufficiently typed lan-
guage, such a situation can be avoided, and the form and
meaning of the database items can be given explicitly in the
program itself. Variant records were chosen to represent the
control blocks. (A variant record is a record with choice of
alternative substructures based on the value of a discrimi-
nant component (3.7).). Two rules which affect the utility
of variant records are: the discriminant can be changed only
during a complete record assignment; and the same com-
ponent name cannot be used in different variant parts. Both
rules caused some difficulties, as will be discussed later.
An enumeration list of block types is the discriminant com-
ponent for the variant record type. This has the declaration:

type block.name is (null.block, pid, nonl, din, llag);
The other components common to all blocks were the block
status word, the name fields, the options word, and the block
parameter. These required separate type declarations and
representation specifications as well. In specifying the var-
iants, it was necessary to identify common types that apply
to the components of the different blocks. The four major
types were value, logical, value-pointer, and logical-pointer.
Each is a record in its own right, with a representation spec-
ifying one word of storage. For instance,

type value is
record
from.pointer:BOOLEAN;
value.is.bad:BOOLEAN;
counts:normalized.counts;
end record;
for value use
record
from.pointer: at 0*WORD range 0..0;
value.is.bad: at 0*WORD range 2..2;
counts: at 0*WORD range 3..15;
end record;
Examples of components in a pid block which are of type
value are the measurement, set point, and output. On the
other hand, in the din block, the measurement and output
are of bit.pattern type, and there is no set point. Hence it
was necessary to place these components in the variant part
of the control block.
The final problem was to choose a naming scheme that allowed easy use of the control blocks in the algorithms. Separately named components would have led to tedious implementation of the algorithms. Instead, like parameters were grouped into arrays by type, with each array indexed by an appropriate enumeration type. For instance,

```plaintext
type pidlist is (meas, remote_sp, feedback, halim, lalim, hdlim, ldlim, holim, lolim, bias, pband, rate, integral, setpoint, output, absdb, devdb, outdb, kl, filtered_meas, integral_balance);
type llalist is (meas, dynamic_gain, timel, bias, output);
```

Note that these lists overload literals, such as meas, and hence, when ambiguities arise, care must be taken in using them, for instance by writing “pid_list(meas)” explicitly.

Finally the data object block is declared in the database package by

```plaintext
block: array (1..32) of control_block;
```

While the development of these types proceeded in top-down fashion, Ada has the unfortunate and annoying restriction that the type declarations must be presented in bottom-up order. This is a hindrance to both writers and readers of a program.

**Status manager package**

MCOS contains status registers to indicate certain conditions of the system hardware/software. The logical representation of the status flags and the specifications of the available operations are encapsulated in the visible part of the status_manager package. The physical representation of the registers (Boolean arrays), as well as the implementation of their corresponding access routines, are concealed in the package body. The status_manager package is global to MCOS. Its visible part provides a simple interface for accessing the status registers.

Within the package body, the implementation of the various access routines differ in the level of protection afforded to the status registers. Protection mechanisms are provided only as dictated by the functional requirements. For example, since writing to the controller status register is accomplished via hardware command registers, no extra protection is needed. On the other hand, since the unit status register is directly read/write accessible, a high degree of protection is desirable. A server task provides this protection. Here the rendezvous is used to prevent simultaneous access to the unit status register by parallel tasks:

```plaintext
package body status_manager is
  task body protect_status_reg is
    begin
      loop
        accept set_status (flag: unit_stat_list.change;
                           new_stat: Boolean) do
          unit_stat_reg (flag) := new_stat;
      end loop;
      end set_status;
      end loop;
      end protect_status_reg;
      begin
        initiate protect_status_reg;
      end status_manager;
```

Note that the initiate statement of the server task is placed in a begin block at the bottom of the package body. Task initiation occurs when the package body is elaborated at runtime.

Since Ada does not prevent simultaneous access to shared data by parallel tasks, it is the programmer’s responsibility to ensure the proper level of protection by controlling access via the rendezvous or semaphore. Ada provides protective mechanisms, but does not enforce their use. For system reliability in a parallel processing environment, therefore, good programming discipline is required. Without it, system security is threatened.

**Control processing**

**The controller executive**

The controller executive is a task that executes in parallel with the communications task. The executive accepts a clock interrupt to begin the control cycle and determines the controller’s current mode (e.g., standby, control, etc.). A controller that is tracking reads the database buffer. A controller in standby simply exits. When the controller is in control mode or initializing mode, it runs a sample control algorithm and compares the output to a known result. If this checks, it proceeds to control block processing. Otherwise it takes itself off control and exits. The executive code is given in Figure 3b.

**The control package**

The procedure do_control_processing is in the module control_package and is called from the task executive. It handles block initialization and regular control processing in a uniform manner. Since blocks can be taken off control by a host computer, this must be checked by reading the appropriate block status bit. The control package is hierarchical in organization since no parallel processing occurs within it.

Two difficulties encountered in control processing involved type conversions and exception handling. We discuss these in detail below.

**Process I/O components**

In the MCOS there are 100 input/output components, each of which can be one of several types. The first attempt at representing the components used an array of variant records. However, physical limitations required using the iden-
Using Preliminary Ada in a Process Control Application

603

Now the id field can be revised independently of the twelve bit value. Moreover, in the case of the digital components, slice assignments can be used to get the information into the appropriate block components. But what happens when the twelve bits must be treated as an integer? Here the package UNSAFE_PROGRAMMING comes into play. It provides a generic facility for converting between otherwise incompatible types; for instance, for converting an object of type pio_data to an INTEGER occupying a 16-bit word.

The following representation ensures that each pio_data value occupies one word.

```ada
for type pio_data use
record
  id: at 0 WORD range 0..3;
  twelve_bits: at 0 WORD range 4..15;
end record;
```

Now UNSAFE_CONVERSION can be used to translate the single word of pio_data type to a single word of INTEGER type. This requires the instantiations function data_to_int is new UNSAFE_CONVERSION (pio_data; INTEGER); temp_data: pio_data; followed by the conversion statements temp_data := (no_data, pio_data. twelve_bits); raw_count := data_to_int (temp_data);

### Numeric computation and error handling

During the processing of the control blocks, some of the algorithms require numeric computations. Because of the real-time nature of the controller, any error conditions that could arise must be handled in such a way that the system does not halt. This section contains a discussion of the error handlers used in designing the MCOS algorithm set.

Suppose we have three variables given by the declaration

\[
X, Y, Z: \text{INTEGER range 0..4000};
\]

followed by three assignment statements, where it is assumed that the expressions on the right-hand side yield INTEGER values:

```ada
begin;
  X := expression_1;
  Y := expression_2;
  Z := expression_3;
end;
```

If one of the expressions has a valid INTEGER value outside the range 0..4000, the RANGE_ERROR condition is raised. At this point the program checks whether a handler has been included within the block. If so, the action specified in the handler is taken. If not, the search for a handler continues in the next outer scope. In the scope given even if we include an exception handler, the program will resume execution not at the next statement following the one raising the exception, but at the statement following the end of the block. Thus, in general, it is not possible to resume execution from the point of error.

In some problems, such as signal conditioning, one wishes
to clamp the value that is out of range. In that case, the following seems cleanest for eliminating the RANGE_ERROR exception.

\[
X := \max (\min (\text{expression}_1, 4000), 0);
Y := \max (\min (\text{expression}_2, 4000), 0);
Z := \max (\min (\text{expression}_3, 4000), 0);
\]

However, this will be valid only so long as the expressions on the right are valid INTEGER values. If one of them is not, an OVERFLOW or a DIVIDE_ERROR exception occurs in the expression evaluation. DIVIDE_ERROR can be avoided by testing the denominator beforehand. This leaves but two possibilities for OVERFLOW: (1) write the expression as a function within which error handlers are implemented; or (2) enclose each statement in a block with an exception handler, as in this block:

\[
\begin{align*}
\text{begin} & \quad X := \text{expression}_1; \\
\text{exception} & \quad \text{-- no matter what goes wrong, clamp} \\
& \quad \text{when OVERFLOW} \Rightarrow \\
& \quad X := 4000; \\
\text{end}; \\
& \quad \text{-- etc--}
\end{align*}
\]

Both of the techniques were useful. In some circumstances it was possible to determine a priori that only RANGE_ERROR could occur and then the explicit clamping was used. In other cases, the special scope was inserted to localize the error handling.

Communications processing

The model controller may receive communications interrupts at any time from either port. In MCOS, the communications task functions at the highest priority level to service such interrupts. As discussed in a previous section, the Ada mechanism for interrupt handling is inadequate, and is in the process of being revised by the language design team. By handling the communications interrupts in preliminary Ada, there is no way to guarantee that they will receive the immediate and dedicated attention that is demanded.

In addition to handling port interrupts, the communications task consists of the following units: (1) message_buffer_manager package encapsulates the input and output buffers together with the access routine, message_handler; (2) message_handler routine decodes the incoming message and calls the appropriate subroutine to process the message and send the reply; (3) error_counter_manager package contains the hardware registers (which record the occurrence of transmission errors), and corresponding software access routines.

Hardware dependencies

The communications process is, perhaps, the most difficult to design and program because of the many direct connections between software and hardware. There are certain circumstances which require a machine code insertion in the Ada program to provide a high-level interface between the hardware and the rest of the software implementation. One advantage to Ada is that it permits such machine-dependent code to interface with the high-level code, isolating and minimizing the degree of machine dependencies. For example, a machine code routine is required to reset the watchdog timer to avoid a timeout.

Decoding messages

The communications messages which are implemented in MCOS are grouped into three categories: station messages, task messages, and process I/O messages. MCOS responds to messages received from the host computer, but does not initiate them. Station messages involve retrieving status information about the controller, and getting/resetting transmission error counters. Task messages allow the host to switch the controller into tracking or standby modes. Process I/O messages get/set values in the database of control blocks.

Each message has a specific command code which indicates the content of the message. The command codes are implemented by a representation specification for elements of an enumeration type, where elements of the type are assigned internal codes corresponding to values of command codes, as shown in Figures 3c and 3d. Ada supports the principle of separation of logical properties from physical properties. However, in the case of an enumeration type, the ordering of elements in the logical specification must correspond to the ascending numerical values assigned in the representation specification. Yet, despite this dependency, Ada enforces a textual separation in that all associated representation specifications must follow the logical specifications in the declarative part.

The incoming message buffer is an array of bytes. The first part of the message containing the command code must be decoded before the rest of the message can be processed. The decoding was implemented in MCOS via UNSAFE_CONVERSION of the appropriate bytes into the command code enumeration type. When a case is done on the command code, illegal codes are caught by the when others alternative.

Processing messages

Legal commands are processed by their respective subroutines, whose stubs are internal to the message_handler procedure. The procedures themselves are separately com-

\begin{verbatim}
TYPE COMMAND IS
  (GET_STATUS, GET_ERROR_CTRS, READ_DATA_STANDARD,
   RESET_ERROR_CTRS, STANDBY, STARTUP, SELECT,
   SECURE_RELEASE, SET_RESET_HOLD);
\end{verbatim}

Figure 3c—Logical specification of command enumeration type (ordering of elements is determined by ordering in Figure 3b).
Hindrances

Design. Otherwise, such functions will require a machine
applications are the lack of a well-defined scheduler and the
inadequacy of the mechanism for interrupt handling. The
Ada language design team has acknowledged these prob­
lems, and, hopefully, will rectify them in the final language
Ada's strong typing is superfluous in response transmission,
helped and at other times hindered the program development
process.

SUMMARY AND CONCLUSIONS

As applied to a typical process control problem, the Model
Controller Operating System, preliminary Ada sometimes
helped and at other times hindered the program development
process.

Hindrances

The major deficiencies of preliminary Ada for real-time
applications are the lack of a well-defined scheduler and the
inadequacy of the mechanism for interrupt handling. The
Ada language design team has acknowledged these prob­
lems, and, hopefully, will rectify them in the final language
design. Otherwise, such functions will require a machine
code implementation.

A related problem is synchronizing access to shared data
in time-critical applications. Implementing mutual exclusion
using the rendezvous construct is awkward and inefficient
as compared to other synchronization primitives such as spin
locks.6

Another hindrance to program development is the required
bottom-up textual presentation of information. This is ex­
hibited by the restriction of no forward referencing in spec­
isfications. Although easier for compiler implementation, lin­
ear elaboration of declarations is not easier for either writers
or readers of Ada programs. A textual presentation reflect­
ing the top-down design process would be preferable.

Also detracting from the readability of Ada programs is
the lack of explicit import lists of objects. Import lists are
not required, yet without them program maintenance is ham­
pered. To avoid the excessive burden on program develop­
ners, the import lists could be automatically generated by
compilers or text editors.

Helps

The major advantage of programming in Ada is the support
provided by packages for encapsulation and information­
hiding. The grouping of logically related data objects, types,
and/or associated procedures greatly enhances the logical
program structure. For instance, levels of protection of
shared data objects may be implemented in the package
body, concealing details from the users of the package.
Ada’s separate compilation facility was used extensively
to support modularization and enhance program structure.
Separation of logical interface from physical implementation
is a positive influence on program development.
Ada’s strong typing is a definite plus. High level data def­
initions improve the readability of the code. In this regard,
enumeration types are particularly useful.

A necessary companion to strong typing is the ability to
escape it when a different view of the object is required,
such as in decoding a message buffer. This is neatly provided
by Ada’s UNSAFE_CONVERSION function.
UNSAFE_CONVERSION identifies those areas of the pro­
gram where the safety checks of strong typing are tempo­
rarily suspended. Without this feature, a greater proportion
of the program would have required machine code imple­
mentation. A related aid to systems programming in Ada is
the coupling of logical to physical representations via the
representation specification.

Issues in programming methodology

During the design and coding of MCOS, some uncertain­
ancies about Ada were raised. They were eventually resolved
as the authors gained experience with the language, and
through consultations with various persons more closely
connected with the Ada language development.*

Issues identified during the MCOS exercise were: (1) the

* In this regard, the authors would like to acknowledge John Barnes, Dennis
Cornhill, Mark Davis, Robert Firth, John Goodenough, Oliver Roubine, and
Peter Wegner.
interplay between traditional block structure and separately compiled modules, and how it affects program structure; (2) using visibility restrictions to advantage; (3) separate compilation of specification and implementation parts to reduce recompilation dependencies; (4) exception handling; and (5) dependency of logical representation on physical representation.

A programming methodology for Ada is required. A user's guide (an Ada cookbook) would facilitate program development. Due to the mixing of standard features with novel ones, the best Ada solution for a particular problem often cannot be ascertained. Current reference documentation is inadequate.

Conclusion

Without a doubt, systems programming is facilitated by using Ada, as compared to a full assembly language implementation. As with any high level language, a small proportion (5-10 percent) of the program will require assembly language, either to maximize efficiency or to interface with hardware. Ada provides an interface to assembly code. Yet, due to the power of the language, machine-dependent code may be kept to a minimum.

REFERENCES