N91-10614

Experiences with Ada in an Embedded System

Robert J. LaBaugh
Martin Marietta Astronautics Group
Space Systems
Denver, Colorado 80201

Introduction

This paper describes recent experiences with using Ada in a real time environment. The application was the control system for an experimental robotic arm. The objectives of the effort were to experiment with developing embedded applications in Ada — evaluating the suitability of the language for the application, and determining the performance of the system. Additional objectives were to develop a control system based on the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) in Ada, and to experiment with the control laws and how to incorporate them into the NASREM architecture.

Background

The arm to be controlled has five degrees of freedom — one degree in each of the shoulder and elbow joints, and a wrist with roll, pitch, and yaw. An Intel 80386 single board computer in a Multibus II system was used for the controller. The board contained an 80387 math coprocessor, two megabytes of RAM, and a single RS-232 serial port. The clock frequency for the system was 16 MHz. Rather than just use the 80386 as a fast 8086, the 80386 was operated as a 32 bit processor in the protected mode, which provides for segment sizes of up to four gigabytes.

The Ada compiler selected was the DDC-I cross compiler for the 80386, which was hosted on a MicroVAX. This compiler was targeted to a bare machine, so there was no operating system to either provide services or detract from the performance of the system. The runtime system supplied with the compiler provided all of the services needed to support the features of the language, including initialization of the hardware, memory management,

time management, the Ada tasking model, and interrupt handlers. An operator interface for the application was implemented using the standard Ada Text_IO package. This package uses the RS-232 port on the single board computer for the standard input and output of Text_IO.

Development Approach

The software development system is shown in Figure 1. It consisted of a Rational R1000, a MicroVAX II, and a PC clone. The systems were connected via Ethernet, which was used to transfer files between the systems. Initial program development was done on the Rational. To facilitate code debug and checkout on the Rational, Ada routines to simulate the hardware were developed. These were used to replace the low level hardware interface routines. When the target hardware and compiler became available the source code was moved to the MicroVAX. Target peculiar modifications were made to the code, such as the specification of task entries as interrupt handlers and the hardware interface routines. The code was then compiled and linked on the MicroVAX, and the resulting load module was downloaded to the PC. The PC served as the controller for the in-circuit emulator, which was used to load and control the execution of the code in the target system.

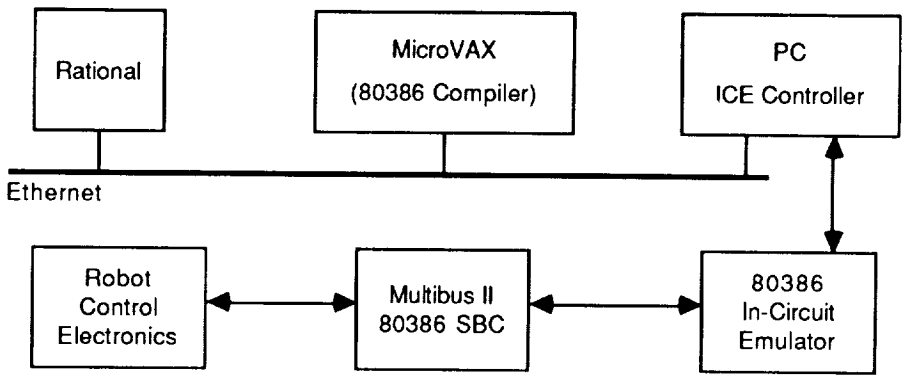


Figure 1. Development System

Even though the capabilities of in-circuit emulators are improving, this was a less than optimal environment for debugging code. Having to move from one terminal to another, moving files from one system to another, and the limitations on file names on the PC all hinder code development and checkout. The movement is clearly toward being able to compile, download, and debug from a terminal on the host development system. There are some systems which currently allow this, but the targets are connected to the host by an RS-232 line. The relatively slow download speeds limit the size of programs which can be effectively developed using these systems.

Ada Features Used

Ada tasks and task rendezvous were used for synchronization and communication between tasks. Task priorities were established using the priority pragma. An interrupt handler was coded in Ada to service the timer used to provide the control loop cycle. This was accomplished using an address clause for a task entry — which is the technique specified in the Ada Language Reference Manual for defining interrupt handlers. The Low_Level_IO package was used to communicate with the hardware controlling the joints on the arm.

There was one package where machine code insertions were used. This was used to provide procedures to disable and enable interrupts. These routines were not really needed by the initial application. They were used to assure safe initialization of the hardware, which was already guaranteed by the sequencing of the initialization routines. However, these routines become necessary as more Multibus II features are used. This is because some logical operations, such as accessing a single Multibus II interconnect space register, require accesses to multiple hardware ports.

Software Application

NASREM defines a layered, hierarchical control system with common interfaces between layers. The lowest layer in the hierarchy operates at the highest frequency, with a decreasing frequency of operation with each higher level. Ada tasks were used to implement the NASREM layers, with the priority of the tasks decreasing with increasing levels in the hierarchy. The requirement that the argument to the priority pragma be a static expression

prevented the use of a generic package in defining the NASREM levels. However this was a minor inconvenience as there was very little code involved in defining the control structure within a level.

The initial application concentrated on the two lowest levels of the NASREM architecture. The servo level reads current joint positions and sends motor commands based on the error between the current and desired position. This level was driven by a programmable hardware clock which generated a periodic interrupt. The primitive level determines evenly spaced points between desired end points and performs the kinematic transformations. The elemental move level initially consisted of simple canned motion generators, and the task level simply selected the motion to be performed. The robot control function and the operator interface were both run on the same CPU, with a total of eleven Ada tasks in the application.

The entire application was coded in Ada. No non-standard pragmas or special interface routines to the runtime system were used. In addition, we were able to effectively write low level code in Ada. This included interrupt handlers, hardware interface routines, Multibus II message passing routines, and control of a DMA processor. The hardware, and the code generated by the compiler, provided more than adequate performance for the system. In experimenting with the control laws the control loop cycle time was varied between 10 and 50 milliseconds. For most of that range all levels of the NASREM architecture were able to complete in a single cycle. Since the NASREM architecture is set up for approximately a ten to one ratio in frequency of operation between levels, this leaves plenty of room for growth.

Current activity includes splitting the robot control function from the operator interface function and executing them on two CPUs. The initial interface and communication between the processors is via shared memory. As an alternative, Multibus II message passing will also be investigated. This is being done as an exercise in distributing the application. Items of interest are the difficulty of implementing various communication schemes and the relative performance.

Lessons Learned

Most of the things which could be considered lessons learned are more appropriately classified as common sense. Specifically, while being able to use a host system for initial debug and test is a useful development tool, it does not eliminate the need for low level testing on the target system. This testing is needed to establish the correctness of the hardware-software interface definitions, and to build confidence in both the hardware and low level software routines. Having a set of programs to incrementally checkout the low level functions and interface also provides the basis for trouble-shooting as problems arise. Such routines were needed to isolate hardware failures and identify improper system initialization, which happened if a specific sequence was not followed for powering on the electronics racks and computers.

Another major lesson learned was that portability is not automatic with Ada. There were two specific instances of this. The first involved differences in the tasking implementation between the Rational and the 80386 target. Tasks of equal priority are time sliced on the Rational, but this is not the default for the DDC-I runtime system. A task which was to run in the background, and which checked flags in an infinite loop, was elaborated before some of the higher priority tasks were initiated. Since the task didn't allow for any type of context switch, as soon as it started executing on the 80386 it kept control of the CPU, preventing the further elaboration of the system. Inserting a delay statement inside the loop fixed the problem. The other experience with non-portable Ada code involved a public domain math functions library. The functions used by the application worked correctly on the Rational. However on the 80386 system one of the functions produced erroneous results for certain input values. It was discovered that this math package had hard coded values for machine specific parameters. We did not try to determine if this was the cause of the problem as an alternative math functions library was available. This does point out the need for extensive test data, and a test mechanism, for "reusable" Ada packages.

There still seems to be a tremendous resistance to using Ada language features for embedded, real-time applications. Some of this comes from "experts" who have heard Ada is not

efficient enough, or just cannot support various real-time or "system" functions. This resistance is probably a positive sign. It used to be said that Ada was too inefficient for almost *all* applications, not just real-time applications. Unfortunately system specifics, such as a particular compiler, target, or any operating system involvement, tend to be forgotten or ignored. There are certainly systems which cannot come close to supporting time critical applications, but this does not mean all systems are that way. Much more surprising is the push by some Ada compiler vendors (and, less surprising, real-time kernel vendors) to promote special, non-Ada runtime systems. This could be seen as an attempt to distinguish their product, or provide a higher performance system where needed. However, it could also be viewed as an attempt to circumvent shortcomings in their runtime system implementation — which could lead to speculations of what else might be inefficient or poorly implemented in the system. The use of such systems greatly reduces the portability of the code and adds another complex system which has to be maintained.

Conclusions

We were able to implement a complex real time system in Ada, and did not have to resort to circumventing Ada language features or use a special, non-Ada run time system. This was a result of having hardware, and an Ada compiler and runtime system, with significantly more performance than was needed by the application. Futhermore, using the Ada tasking system allowed the initial debug and test of the code to be performed on the host development system, which was more accessable than the target system. This also allowed the debug and testing to begin before the target system was available. Another advantage of using Ada tasks and having sufficient performance margin was that it allowed the application to be implemented primarily by junior engineers. Some guidance was provided on implementing the interrupt handler and cyclic task execution. Otherwise they were able to use textbook tasking solutions, such as having tasks to coordinate exclusive access to resources. All of this indicates that as Ada compilers continue to mature the idea of leveraging of skills can be extended to the real-time arena.

THE VIEWGRAPH MATERIALS

FOR THE

R. LABAUGH PRESENTATION FOLLOW

EXPERIENCES WITH ADA IN AN EMBEDDED SYSTEM

Robert J. LaBaugh

MARTIN MARIETTA ASTRONAUTICS GROUP
SPACE SYSTEMS
DENVER, COLORADO

INTRODUCTION

EMBEDDED APPLICATION

- CONTROL OF EXPERIMENTAL ROBOT ARM

OBJECTIVES

- DEVELOP EMBEDDED APPLICATION IN ADA
  - EVALUATE LANGUAGE FEATURES
  - DETERMINE PERFORMANCE
- DEVELOP CONTROL SYSTEM BASED ON NASREM ARCHITECTURE IN ADA
- EXPERIMENT WITH CONTROL LAWS

BACKGROUND

ROBOT ARM WITH FIVE DEGREES OF FREEDOM

- SHOULDER, ELBOW, AND WRIST WITH ROLL, PITCH AND YAW

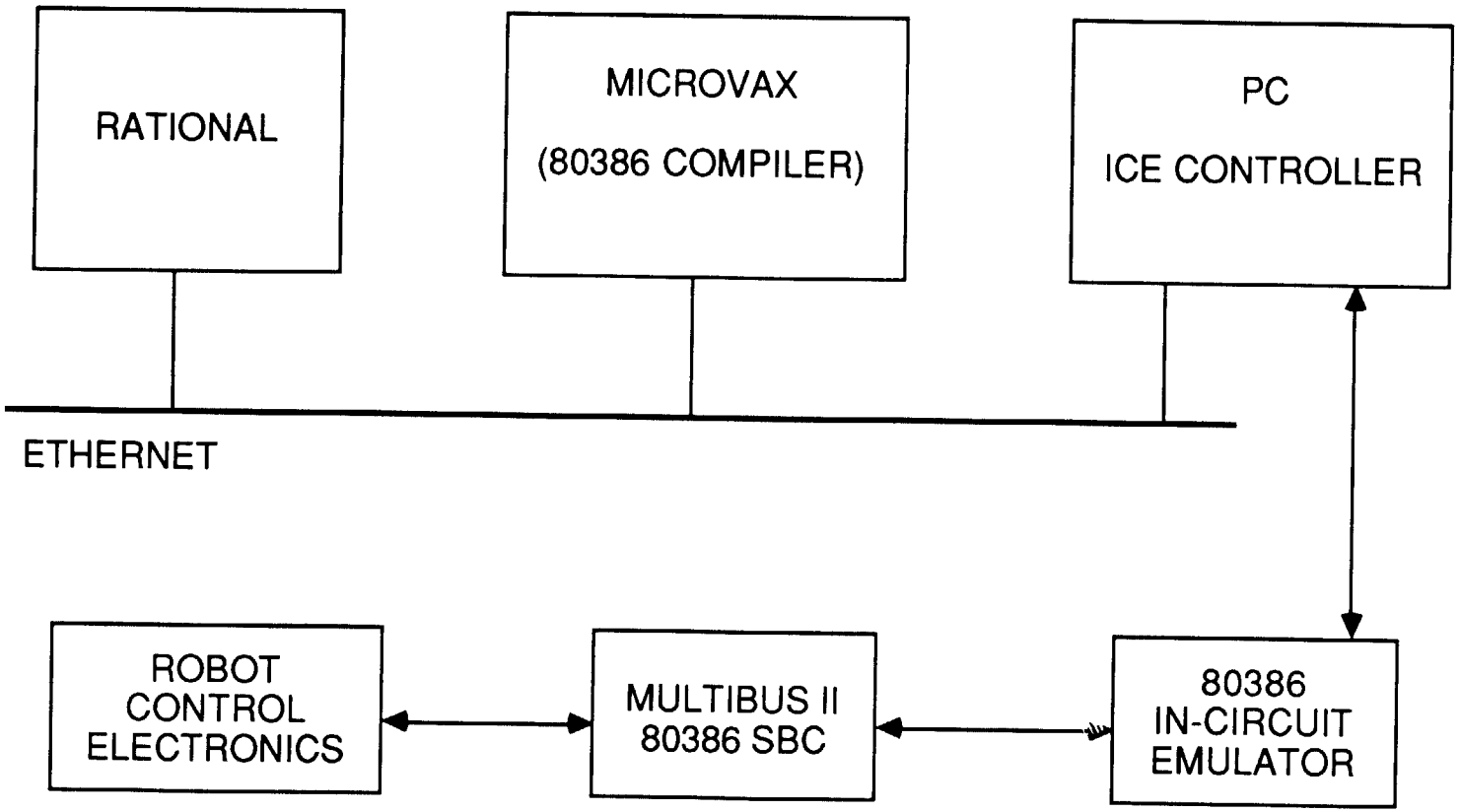
TARGET COMPUTER — INTEL 80386 SINGLE BOARD COMPUTER

- 80387 MATH COPROCESSOR
- 16 MHZ CLOCK FOR CPU AND COPROCESSOR
- 2 MB RAM
- RS-232 PORT

ADA COMPILER — DDC-I ADA 80386 PROTECTED MODE

- TARGETED TO BARE MACHINE
  - NO OPERATING SYSTEM
  - ADA RUNTIME SYSTEM PROVIDES COMPLETE CONTROL OF HARDWARE AND SOFTWARE
- TEXT_IO USES RS-232 PORT ON CPU BOARD
  - NON-BLOCKING I/O
  - USED FOR OPERATOR INTERFACE AND DEBUGGING

DEVELOPMENT SYSTEM



DEVELOPMENT APPROACH

INITIAL DEVELOPMENT AND TEST ON RATIONAL

- LOW LEVEL HARDWARE INTERFACE REPLACED WITH SIMULATION ROUTINES

SOURCE CODE DOWNLOADED TO MICROVAX

CROSS COMPILED AND LINKED ON MICROVAX

- ADA CROSS COMPILER FOR 80386 HOSTED ON VAX SYSTEM

LOAD MODULE MOVED TO IBM PC CLONE

- CONTROLER FOR IN-CIRCUIT EMULATOR
- USED TO LOAD AND CONTROL EXECUTION OF CODE IN TARGET SYSTEM

LESS THAN OPTIMAL ENVIRONMENT FOR DEBUG ON TARGET

- MOVING TOWARD COMPILING, DOWNLOADING, AND DEBUGGING FROM ONE TERMINAL
  - AVAILABLE NOW FOR SOME SYSTEMS, BUT LIMITED BY RS-232 SPEED

ADA FEATURES USED

ADA TASKS

- TASK RENDEZVOUS FOR SYNCHRONIZATION AND COMMUNICATION

INTERRUPT HANDLER CODED IN ADA

- ADDRESS CLAUSE FOR TASK ENTRY

PRIORITIZED TASKS

LOW_LEVEL_IO USED TO COMMUNICATE WITH HARDWARE INTERFACES

MACHINE CODE INSERTIONS

- USED TO DISABLE AND ENABLE INTERRUPTS
- NOT NEEDED IN INITIAL APPLICATION
  - USED FOR SAFE INITIALIZATION OF HARDWARE
- NEEDED AS MORE MULTIBUS II FEATURES USED
  - MULTIPLE HARDWARE ACCESSES PER LOGICAL OPERATION

APPLICATION

NASREM

- LAYERED, HIERARCHICAL CONTROL SYSTEM ARCHITECTURE
- COMMON INTERFACES AT ALL LEVELS
- DECREASING FREQUENCY OF OPERATION AS LEVELS INCREASE
- IMPLEMENTED WITH ADA TASKS

CONCENTRATED ON LOWEST TWO LEVELS

- SERVO LEVEL — SENDS MOTORS COMMANDS BASED ON ERROR BETWEEN CURRENT AND DESIRED POSITION
- PRIMITIVE LEVEL — DETERMINES EVENLY SPACED POINTS FOR SMOOTH MOTION, KINEMATIC TRANSFORMATIONS
- ELEMENTAL MOVE LEVEL — SIMPLE CANNED MOTION GENERATORS
- TASK LEVEL — SELECTS MOTION TO PERFORM

OPERATOR INTERFACE AND ROBOT CONTROL FUNCTIONS ON SAME (SINGLE) CPU

- 11 ADA TASKS

HARDWARE CLOCK INTERRUPT DRIVING CONTROL LOOP

RESULTS

ENTIRE APPLICATION CODED IN ADA

- NO SPECIAL KERNEL, SYSTEM CALLS, OR PRAGMAS
- "LOW LEVEL" CODE WRITTEN IN ADA
  - HARDWARE INTERFACE, DMA CONTROLLER, MULTIBUS II MESSAGE PASSING, ETC.

MORE THAN ADEQUATE PERFORMANCE FOR 50 HZ CONTROL LOOP

- ALL NASREM LEVELS ABLE TO EXECUTE IN 20 MSEC CYCLE

IN PROCESS OF SPLITTING CODE ONTO TWO CPUS

- ROBOT CONTROL
- OPERATOR INTERFACE
- INITIAL COMMUNICATION VIA SHARED MEMORY
  - ALSO EXPERIMENT WITH MULTIBUS II MESSAGE PASSING
- EXERCISE IN DISTRIBUTING APPLICATION TO HANDLE GROWTH

LESSONS LEARNED

ABILITY TO TEST AND DEBUG ON HOST DOES NOT ELIMINATE NEED FOR LOW LEVEL TESTING ON TARGET

- NEEDED TO BUILD CONFIDENCE IN BOTH THE HARDWARE AND SOFTWARE
  - CORRECTNESS OF INTERFACE DEFINITIONS
- NEEDED TO PROVIDE BASIS FOR TROUBLE-SHOOTING PROBLEMS
  - HARDWARE FAILURES, IMPROPER SYSTEM INITIALIZATION (POWER TURN ON)

PORTABILITY IS NOT AUTOMATIC WITH ADA

- DIFFERENCES IN TASKING IMPLEMENTATION IN RUNTIME SYSTEMS
  - TIME SLICING OF EQUAL PRIORITY TASKS ON RATIONAL
  - TIME SLICING NOT THE DEFAULT ON TARGET
- PUBLIC DOMAIN PACKAGES NEED SUPPORTING TEST DATA AND TEST MECHANISM
  - MATH FUNCTIONS LIBRARY — HARD CODED, MACHINE SPECIFIC PARAMETERS VS USE OF ADA LANGUAGE FEATURES — ERRONEOUS RESULTS FOR CERTAIN INPUT

LESSONS LEARNED (continued)

TREMENDOUS RESISTANCE TO USING ADA LANGUAGE FEATURES FOR EMBEDDED, REAL-TIME APPLICATIONS

(I.E. STANDARD ADA RUN TIME SYSTEM, INTERRUPT HANDLERS VIA TASK ENTRIES, LOW_LEVEL_IO)

- FROM "EXPERTS" WHO HAVE HEARD ADA IS NOT EFFICIENT ENOUGH OR CANNOT SUPPORT VARIOUS REAL-TIME/SYSTEMS FUNCTIONS
  - SPECIFICS (COMPILER, TARGET SYSTEM) TEND TO BE IGNORED
  - POSITIVE SIGN (REDUCED AREA OF "NON-USABLE")
- FROM COMPILER VENDORS AND REAL-TIME KERNEL VENDORS
  - PUSHING NON-STANDARD KERNELS/OPERATING SYSTEMS
  - ATTEMPT TO DISTINGUISH PRODUCT, OR CIRCUMVENT SHORTCOMINGS IN THEIR RUNTIME SYSTEM IMPLEMENTATION
  - COUNTERPRODUCTIVE — REDUCES PORTABILITY, ADDS ANOTHER COMPLEX SYSTEM WHICH HAS TO BE MAINTAINED

SUMMARY/CONCLUSIONS

COMPLEX REAL-TIME SYSTEM IMPLEMENTED ENTIRELY IN ADA

- HARDWARE AND COMPILER WITH SUFFICIENT PERFORMANCE

USING STANDARD ADA RUNTIME SYSTEM ALLOWED INITIAL DEBUG AND TEST ON THE HOST SYSTEM

- MORE ACCESSABLE THAN THE TARGET SYSTEM
- AVAILABLE EARLIER

PERFORMANCE MARGIN AND USE OF ADA FEATURES ALLOWED JUNIOR ENGINEERS TO IMPLEMENT THE APPLICATION

- GUIDANCE PROVIDED FOR IMPLEMENTING INTERRUPT HANDLER AND CYCLIC TASK OPERATION
- ABLE TO USE "TEXTBOOK" TASKING SOLUTIONS — EXCLUSIVE ACCESS TO RESOURCES

ADA COMPILERS ARE MATURING ENOUGH TO BE USED IN EMBEDDED SYSTEMS

PANEL #4

TOOLS

D. Drew, Unisys
P. Usavage, Jr, General Electric
J. F. Buser, Software Development Concepts