Air Force Institute of Technology
AFIT Scholar

Theses and Dissertations | Student Graduate Works

12-1996

Implementation of an ADA95 Crosscompiler for the Real-Time Executive for Military Systems (RTEMS)

Kendall M. Lemley Jr.

Follow this and additional works at: https://scholar.afit.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation

Lemley, Kendall M. Jr., "Implementation of an ADA95 Crosscompiler for the Real-Time Executive for Military Systems (RTEMS)" (1996). *Theses and Dissertations*. 5873.
https://scholar.afit.edu/etd/5873

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact AFIT.ENWL.Repository@us.af.mil.

AFIT/GCS/ENG/96D-16

IMPLEMENTATION OF AN ADA95 CROSS-COMPILER FOR THE REAL-TIME EXECUTIVE FOR MILITARY SYSTEMS (RTEMS)

THESIS

Kendall M. Lemley, Jr., Capt, USAF
AFIT/GCS/ENG/96D-16

19970409 037

DTIC QUALITY INSPECTED 2

Approved for public release; distribution unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

AFIT/GCS/ENG/96D-16

IMPLEMENTATION OF AN ADA95 CROSS-COMPILER FOR THE REAL-TIME EXECUTIVE FOR MILITARY SYSTEMS (RTEMS)

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air Education and Training Command

In Partial Fulfillment of the Degree of

Master of Science in Computer Engineering

Kendall M. Lemley, Jr., B.S.

Capt, USAF

December 1996

Approved for public release; distribution unlimited

Acknowledgments

First, I would like to thank my advisors, Maj Mark Kanko and Maj Dean Schneider, for their guidance, support, and encouragement throughout the entire thesis process. Additional thanks go to my third committee member, Maj Keith Shomper, whose comments helped greatly in improving the quality of this document. I could not have performed this research without the help of Mr. Joel Sherrill at On-Line Applications Research Corporation, Redstone Arsenal, Alabama. His prompt, clear answers to my many questions, and willingness to diligently pour through error messages proved invaluable.

Finally, I would like to thank my wife, Michelle, for putting up with me for the past several months. Without her love, help, and support, this effort would not have been possible. Thank you.

Table of Contents

Page

List of Figures

Page

Abstract

This thesis effort represents a continuation of the assessment of the Unified Telerobotics Architecture Project (UTAP), a proposed Air Force standard. This architecture was developed by the NASA Jet Propulsion Laboratory and the National Institute of Standards under contract to the Air Force Materiel Command Robotics and Automation Center of Excellence at Kelly AFB, Texas.

Due to operating system constraints, the only UTAP implementation to date has required a separate software interface layer, adding complexity and overhead to the overall system, while reducing portability. This thesis proposes a long term effort to design and implement UTAP-compliant application software devoid of this interface layer. Because the Ada programming language offers increased portability, and other software engineering benefits, emphasis is placed on developing a run-time infrastructure that will allow UTAP applications to be written in Ada. The first several steps of building this infrastructure is performed, including implementation of an Ada cross-compiler and real-time operating system.

Further UTAP research is recommended. The run-time infrastructure should be completed and UTAP application software developed using the Ada95 tasking model. Recommendations for UTAP specification improvements are also made.

IMPLEMENTATION OF AN ADA95 CROSS-COMPILER FOR THE REAL-TIME EXECUTIVE FOR MILITARY SYSTEMS (RTEMS)

1. Introduction

1.1 Background

According to Webster, a robot is "an automatic device that performs functions normally ascribed to humans" [Web84]. There are several advantages in using a robot to perform such tasks. A robot can be programmed to perform a series of predictable or repetitive operations quickly and efficiently. Robots are capable of accomplishing precise tasks with pinpoint accuracy. Perhaps most importantly, robots can be made to operate in dangerous environments without risk of human injury [Tay90].

A telerobot is defined as a robot with a human "in the loop." The degree of human involvement varies from system to system. Certain telerobots require complete real-time control by a human operator using a joystick, trackball, or similar device. Others allow only partial control of selected subsystems. In either case, telerobotics combines the advantages of robots with the ability of humans to think and respond. Telerobots are becoming increasingly prevalent in the Air Force today, used to perform tasks such as ground support of the C-5A/B aircraft [Anc95].

Telerobotics and computer technology are very closely related. In a typical configuration, one or more microprocessors interpret human input and produce the signals needed to control the various parts of the robot. The framework by which these communications occur is the telerobotic control architecture. This control architecture

is specified in the application software used to control the robot. The application software, in turn, relies on the microprocessor's operating system software to provide basic services and communications.

With software playing such a large role in robotics, there has been greater emphasis on applying sound engineering principles to robotics software development. Some important software engineering principles include:

a. Maintainability – "the ease with which software can be understood, corrected, adapted, and/or enhanced" [Pre87]

b. Portability - the ability for software to be moved from one platform to another with little or no change

c. Reusability - "the creation and reuse of software building blocks" [Pre87]

The Ada programming language was developed by the Department of Defense to better embody these software engineering principles than previous languages. According to Federal Aviation Administration and Carnegie Mellon Software Engineering Institute studies, Ada is the language of choice for building reliable, maintainable software systems [AF94]. Ada was designed to be used in embedded real-time military applications, but has been successfully used in many other domains as well, including rail, commercial aviation, biomedical, and communications [Ada96]. To date, Ada has yet to find a niche in the robotics community.

The Robotics and Automation Center for Excellence (RACE), at Kelly Air Force Base, Texas, has defined an open telerobotics control architecture in an attempt to standardize the interface between telerobotic subsystems. Ultimately, this Unified

Telerobotic Architecture Project (UTAP) will allow all Air Force robotics application software to be built from a library of hardware and software modules sharing the same UTAP interface [UTA95]. A significant UTAP goal is to better incorporate the key software engineering principles discussed earlier into the telerobotics domain.

1.2 Problem

To date, the UTAP architecture has been demonstrated to work, but only at a single Air Force site. In 1995, the AFIT Robotics lab was redesigned to be compliant with UTAP. However, in this implementation, the operating system was unable to provide features needed to fully embrace the UTAP philosophy. Furthermore, the operating system did not support the Ada programming language, and therefore, did not allow the UTAP implementation to utilize Ada's many benefits [Anc95].

1.3 Purpose

The purposes of this research were to further the development of a standard interface for robotics applications, and to bring the benefits of the Ada programming language to bear on the robotics domain. In order to do this, the ramifications of infusing Ada into the existing UTAP implementation needed to be explored. Unfortunately, the infrastructure needed to write UTAP-compliant applications in Ada did not exist. The goal of this research was to develop this infrastructure so that future

research can focus on building UTAP-compliant routines in Ada, without worrying about the underlying language and hardware support.

1.4 Approach/ Methodology

This research was conducted using the following general steps:

a. Conducted analysis of the UTAP specification and the work that had been done to date. Determined whether the goal of this research could best be accomplished by developing an Ada compiler for use with the current operating system, or abandoning the current operating system altogether.

b. Conducted a critical review of existing real-time operating systems for potential incorporation into the AFIT Robotics Laboratory. Selection criteria are discussed later in this document.

c. Selected a compiler and real-time operating system (RTOS) that are compatible with existing laboratory hardware and supports the Ada programming language.

d. Implemented the new compiler and RTOS in the laboratory.

1.5 Assumptions and Constraints

The assumption was made at the outset that this research would utilize software written in the Ada programming language to the fullest extent possible. This will allow the research to more effectively determine the ease with which Ada could be implemented on typical robotics hardware.

All necessary hardware for this research is available in the AFIT Robotics and Automation Applications Group Lab B facility. Determining which software components to use (such as compilers and operating systems) and from where they

might be obtained was part of the research effort. The top-level hardware architecture can be seen in Figure 1.

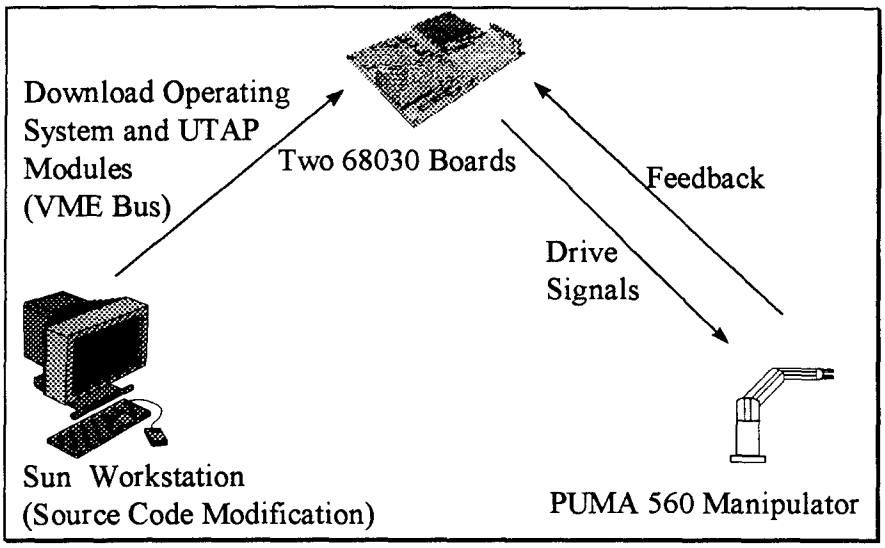

Figure 1: Laboratory Hardware

1.6 Organization

This document is organized into five chapters. Chapter 1 provides basic information on telerobotics and an overview of this effort. Chapter 2 contains a review of previous efforts. Chapter 3 describes in detail, the analysis performed at the outset of this research. Chapter 4 contains research implementation, results and long term benefits. Chapter 5 provides the author's conclusions and recommendations. This document contains several appendices as well. Appendix A describes various pitfalls and lessons learned associated with building the GNU Ada Translator (GNAT) cross-compiler. Appendix B contains the step-by-step procedure for building the GNAT

cross-compiler. Appendix C lists the steps needed to compile the Real-Time Executive for Military Systems (RTEMS) under GNAT version 3.05. Appendix D contains a listing of RTEMS source code changes needed for compilation.

2. Literature Review

2.1 Introduction

In this literature review, the following three questions will be answered:

- Are the goals of the Unified Telerobotic Architecture Project and the Ada programming language compatible?
- Do proper tools exist to support using Ada in the field of robotics?
- What research has been done to date?

The answers to these questions were used to lay the foundation for this research.

2.2 The Unified Telerobotic Architecture Project (UTAP)

The software engineering community has worked hard in recent years to better implement the concepts of "reuse" and "portability." Software reuse allows developers to build complex new systems without having to do so from scratch. Portable software can be run on different hardware platforms and operating systems with minimal modification. A good example of these concepts at work is the personal computer industry. With personal computers, although customers buy software and hardware products from thousands of different companies, the products all communicate properly with one another despite differences in the target platforms. This happens because the communication interfaces among the various subsystems are standard and well-defined [RACE96].

The field of robotics, on the other hand, is plagued by a lack of interface standards. Although 80% of robotics software is common to all robotics applications, vendors tend to start their development efforts with a blank sheet of paper. The result is robots that cannot communicate with one another, and robotics software that cannot be reused. This drives up both development and maintenance costs [RACE96].

The Air Force Robotics and Automation Center of Excellence (RACE) at Kelly AFB, Texas has sponsored UTAP. The purpose of this effort is to define a standard interface among hardware and software modules. These modules can be reused across multiple platforms, thus reducing development and maintenance costs. The ultimate goal of UTAP is for all Air Force telerobotics applications to be built from a single library of compliant modules [UTA95].

The backbone of UTAP is the specification of modules and messages. UTAP defines twenty software modules which can encompass the architecture of any telerobotics system. Not every system will contain every module, and some systems may even need multiple instances of certain modules. A key feature of UTAP is the means by which these modules communicate with one another. Any communication among modules occurs through the issuing of a "message." The UTAP specification explicitly defines 339 messages which carry out all inter-module communication. The specification also allows for the inclusion of unique, programmer-defined messages as needed [UTA95].

To date, the UTAP specification has only been implemented on a very small scale. The first successful implementation is described in [Anc95]. Lt Kevin Anchor redesigned the AFIT Robotics and Automation Application Group (RAAG) Lab B facility to comply with the UTAP specification. His implementation included three modules and fifteen messages. Because the operating system did not allow the passing of messages between modules, his implementation also included a separate interface layer, which was necessary to keep the UTAP-compliant modules from being operating system-dependent. The overhead added by this interface layer seemed to have contributed to a degradation in overall system performance. Lt Anchor recommended eliminating this interface layer by implementing UTAP-compliant software under an operating system that supported generic message passing. Furthermore, he recommended that future UTAP applications be written in a different programming language to demonstrate that the specification was programming language-independent. Ada was the suggested programming language for demonstrating this capability [Anc95].

UTAP has been implemented in two other instances as well. In late 1994 and early 1995, Advanced Cybernetics Group, Inc. (ACG) implemented a small scale UTAP-compatible environment using the commercial Adept V+ programming language [ACG95]. A separate implementation using V+ on an Adept 550 manipulator was designed by Capt Matthew June in a thesis effort conducted in parallel with the effort described in this document [June96].

2.3 The Ada Programming Language

The U.S. Air Force, in its Guidelines for Successful Acquisition and Management of Software Intensive Systems, calls Ada the "enabling technology" for software engineering. It cites a 1991 study in which Ada was chosen over C and C++ by a group of industry experts as the language of choice for developing well-engineered software. In a separate Federal Aviation Administration study, Ada outperformed C, Pascal, Jovial, and Fortran in capability, efficiency, reliability, and maintainability. A 1991 Software Engineering Institute study drew similar conclusions in comparing Ada to C++ [AF94].

Although originally developed by the U.S. Department of Defense, Ada has been used in a number of critical, non-DOD real-time applications as well. For instance, the Ada language was chosen for use in the automatic piloting system on the new Paris underground train, METEOR. The system automatically controls train traffic, regulates speed, manages alarm devices, and maintains control of both automatic and non-automatic trains on the same line. Reliability, security, and long-term software lifecycle costs were all factors considered in choosing Ada over other languages. Ada has also been used in several airborne applications, such as the Airbus 340 and Boeing 777 commercial aircraft [Ada96].

2.3.1 Ada Tasks for UTAP Messages

A key feature of Ada that is useful in the development of real-time software is its tasking model. Ada tasks allow programmers to consider concurrency when developing real-time systems. The allocation of concurrent modules to one or more processors is handled completely by the compiler and run-time environment. Communication among various concurrent modules is performed through rendezvous. Rendezvous can be made more flexible using selective acceptance, delay alternatives, timed entry calls, conditional entry calls, and families of entries [Coh96].

If the appropriate operating system and compiler are in place for the target hardware, Ada tasking can be used to implement UTAP modules and messages. Each module can be declared as an Ada task, and messages implemented as rendezvous. A module would be characterized by the messages that it is capable of sending and receiving. If a module were capable of sending a message, it would make an entry call somewhere within its source code, containing the appropriate data, to the appropriate task capable of receiving it. Modules capable of receiving a message would have an accept statement for each message it could receive.

In the following Ada source code example, a Robot Servo Control (RSC) task sends a US_OK_MODIFY message to the Object Knowledgebase (OK). These modules and messages are from the UTAP specification.

```
task body RSC is
        begin
                ...
                ...
                OK.US_OK_MODIFY         (msgid => 1805,
                                         obj_id => X,
                                         size => Y,
                                         data => Z);
                ...
                ...
        end RSC;


task type OK is
        entry US_OK_MODIFY (msgid, obj_id, size, data);
end OK;


task body OK is
        begin
                ...
                accept US_OK_MODIFY (msgid, obj_id, size, data) do
                        -- Perform US_OK_MODIFY algorithm
                end US_OK_MODIFY;
                ...
        end OK;
```

For an Ada tasking UTAP implementation to be successful, the UTAP specification must more clearly define what modules can send and receive different messages. The message algorithms must also be fully defined. Specific UTAP specification improvements are recommended in Chapter 5.

2.4 Previous Work with Ada and Robotics

Despite the obvious advantages of using Ada in real-time applications, Ada has yet to find a strong niche in the robotics community. Some work was done at Texas A&M University nearly a decade ago, but the researchers had difficulty obtaining Ada device drivers for their laboratory [Volz96]. Device drivers are critical because they provide the means by which the microprocessor communicates with both the telerobot operator and the telerobot itself.

Much of the previous work done with Ada and robotics underscores the need for UTAP. As early as 1984, researchers were able to identify what Ada might bring to the robotics industry. Dr. Richard Volz (at the time on the faculty at the University of Michigan, now the head of the Computer Science Department at Texas A&M) writes,

> ". . . designing new robot languages for nearly every new robot may satisfy the particular programming needs of each robot, but it is counterproductive from the standpoint of developing integrated manufacturing cell technology. Standardization is clearly needed."

Volz goes on to describe an effort where Ada was used to specify interfaces among several manufacturing cell components, including robots. While he stopped short of suggesting that Ada is a panacea for standardization within the robotics industry, he does conclude that the data abstraction, strong type checking, expressive power, and portability of Ada are promising, and that further investigation is warranted [Volz84].

In future research, Dr. Volz and Dr. Arch Naylor published papers introducing the concept of "software/ hardware components" to be used as building blocks in developing integrated manufacturing systems. This object-oriented philosophy involves decomposing manufacturing cell systems into various components and defining the interfaces using Ada package specifications. If the Ada package specifications of the components are standardized, different vendors' Ada package bodies can be linked in and used in a true "plug and play" fashion [Volz87, Volz88].

The similarity between the way robotics interface problems were described in the Volz papers and the UTAP documentation is remarkable. Both describe the robotics industry as being rooted in the practice of developing non-portable, stand alone, proprietary systems with little regard for compatibility or reuse. The result is expensive, unmaintainable, single-use robotics software that cannot communicate with anything other than itself. A widely-used standard interface using the benefits of the Ada programming language would go a long way toward solving many of these problems.

2.5 Candidate Operating Systems

In order to develop Ada applications in the AFIT Robotics Lab, either a new operating system had to be installed, or an Ada compiler had to be written for the existing operating system. Because the existing operating system was such a poor match with UTAP (i.e. it does not support generic message passing), the first option

was chosen for the purpose of this research effort. The search for a real-time operating system for use in this research focused on several key criteria, including Ada support, hardware compatibility, technical support availability, technical support cost, and product cost. The application of these criteria is discussed in Chapter Four.

2.5.1 Real-Time Executive for Military Systems (RTEMS)

RTEMS is an object-oriented, real-time executive designed to be used in embedded applications. It was developed by On-Line Applications Research Corporation under contract to the U.S. Army Missile Command at Redstone Arsenal, Alabama.

As seen in Figure 2, RTEMS is a software platform which provides features and services to an application program. RTEMS features include multitasking,

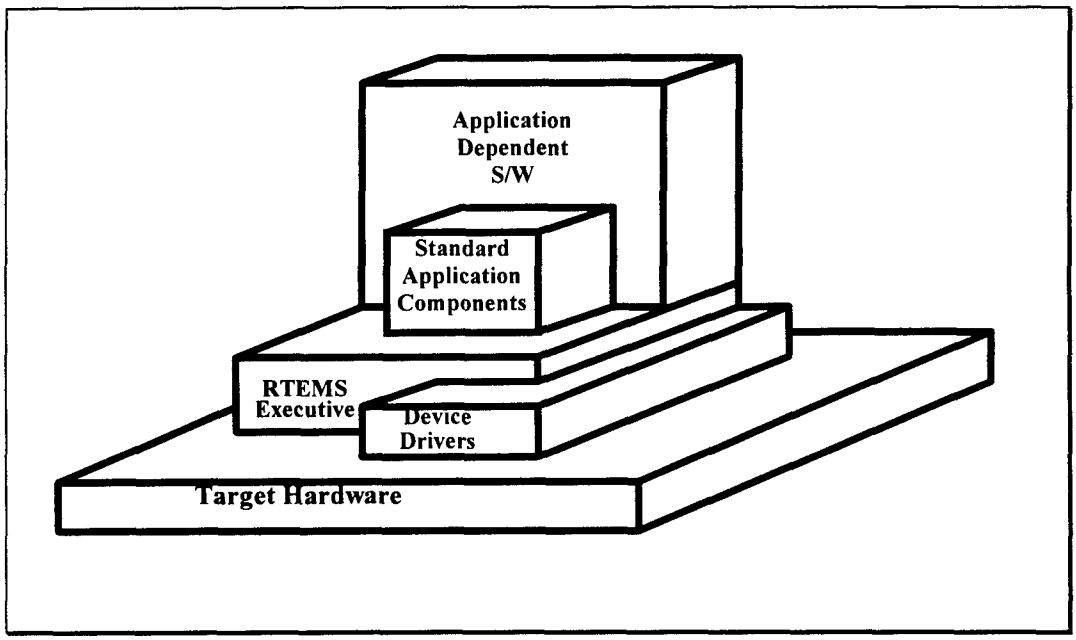

Figure 2: RTEMS Framework [RTE95]

a variety of task scheduling options, and generic message passing among multiple processes. Messages queues can be created and deleted dynamically, and are allowed to operate either in first in/first out, or priority modes. Other RTEMS features can be seen in Figure 3.

RTEMS versions written in both C and Ada are available to the public via the World Wide Web [RTE96]. Applications can be developed in Ada, C, and assembly language [Acu94]. To date, RTEMS has been used in a number of diverse applications, including the U. S. Army's Avenger missile system [Acu96].

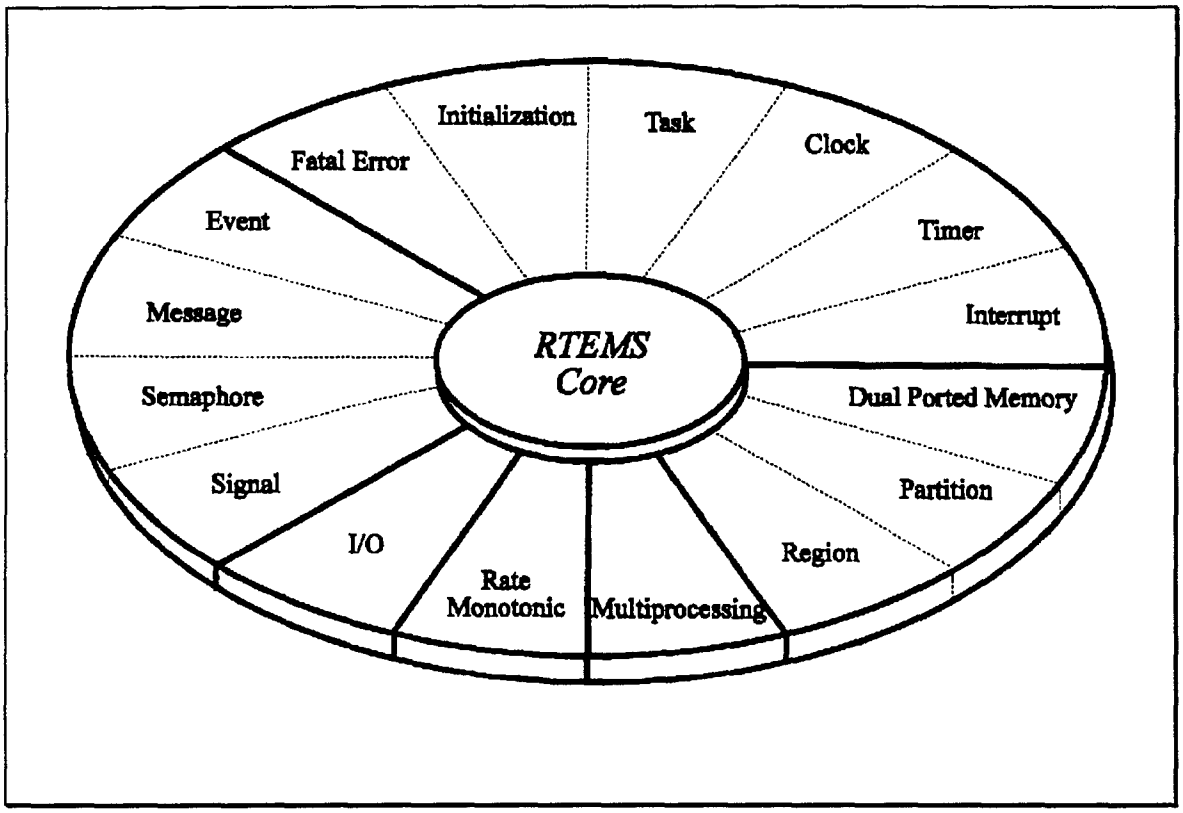

Figure 3: RTEMS Composition [RTE96]

2.5.2 Ada Avionics Real-Time Software (AARTS)

Another candidate was the Ada Avionics Real Time Software (AARTS) operating system, developed by TRW and American Computational under contract to Wright Laboratories. As the name suggests, it has been used primarily in airborne applications. Figure 4 shows how AARTS typically fits into an avionics application, providing the interface between the CPU and the application software.

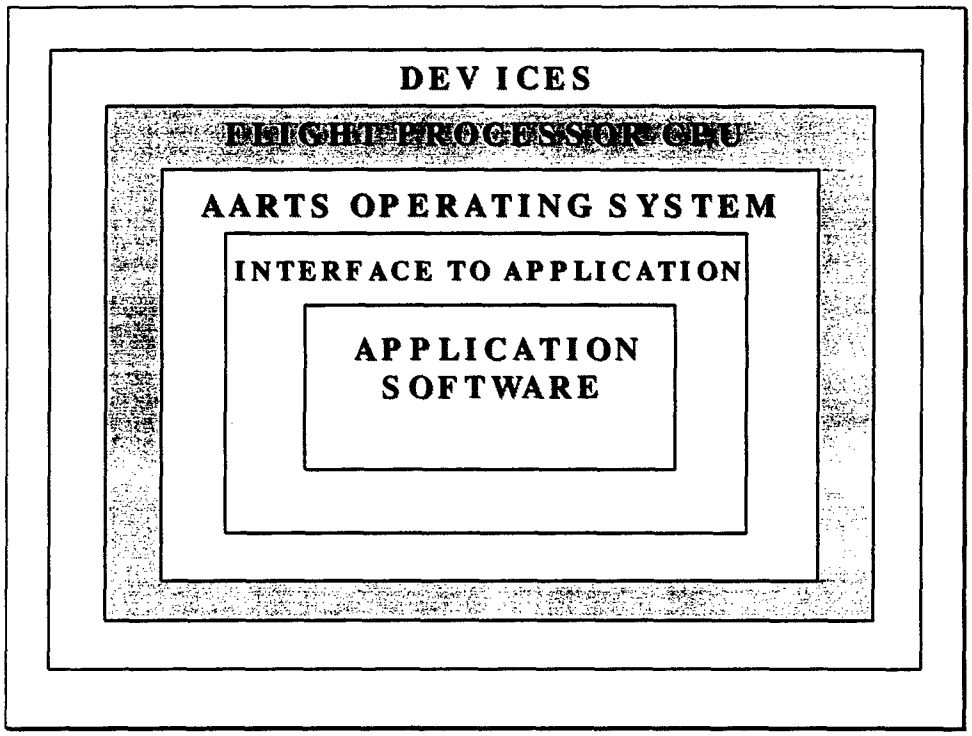

Figure 4: AARTS Framework [AART95]

The key feature of AARTS is its portability. The vendor is currently in the process of commercializing AARTS as a fully portable, host-independent operating system. AARTS includes most standard operating system features, including generic

message passing between applications through its Message I/O package. The source code for this product is available through Wright Laboratories [Mie95].

2.5.3 LynxOS

LynxOS is an operating system developed by Lynx Real-Time Systems of Los Gatos, California. It is designed to look and feel like UNIX from an application developers perspective, and has been used previously to implement robotics systems. Versions are available for the AFIT laboratory hardware in both self-hosted and cross-hosted modes. Ada support is not available directly from Lynx, but can be purchased from one of two third party vendors, OC Systems, or Rational Software Corporation [Lyn96].

2.5.4 VxWorks

Like LynxOS, VxWorks is a commercial product. It is marketed by Wind River Systems as a complete development and execution environment for all types of embedded, real-time applications. It consists of a scaleable run-time system consisting of microkernel, operating system, I/O, networking, and development tools. Figure 5 shows the various pieces of the VxWorks development environment. Supported host

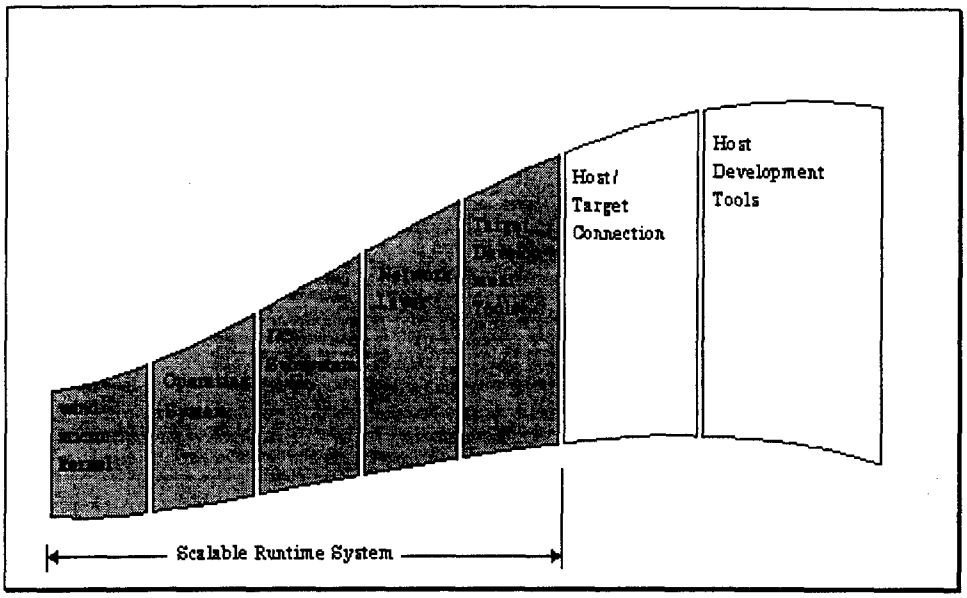

Figure 5: VxWorks Integrated Development Environment [WRS96]

platforms include UNIX workstations, while targets include the entire 68000 family of processors, among others. VxWorks comes only with C and C++ development tools, but Ada support is once again available from third party vendors [WRS96].

2.6 Summary

The beginning of this chapter posed three questions to be answered by this literature review:

- Are the goals of the Unified Telerobotic Architecture Project and the Ada programming language compatible?

  Answer: Yes. Both encourage standardization, and both were designed for embedded real-time systems.

- Do proper tools exist to support using Ada in the field of robotics?

  Answer: There are several operating systems available to support such an effort. Device drivers will likely need to be developed by the researcher.

- What research has been done to date?

  Answer: Not much. The UTAP specification has been demonstrated to work, but its portability remains unproven. Ada has yet to be used much in the robotics community.

In short, the field of robotics will be well served by the software engineering goals embodied by the Ada programming language. This is an area which is ripe for research. The next chapter describes the analysis performed at the outset of this effort.

3. Analysis

This chapter describes the analysis portion of this research. Since [Anc95] was the first attempt to implement the UTAP specification, it seemed necessary to first analyze how successful it was in doing so. This chapter describes how some of the problems in [Anc95] could be alleviated by removing obstacles brought about by the operating system and programming language used. A research effort that removes these obstacles is then proposed.

3.1 Anchor's Research: A Good First Step?

Lt Kevin Anchor was the first person to actually implement the UTAP specification and test it on real hardware. In doing so, he encountered many obstacles, some of which he managed to overcome, some of which he did not. The difficulties fell into two main categories. First, several constraints were brought about due to the lack of required detail in the UTAP specification itself. The second category of difficulties were those brought about by inherent constraints in the operating system used in the effort.

3.1.1 Lack of Detail in UTAP

The developers of UTAP stated that their ultimate goal was for modules developed in accordance with the UTAP specification to be capable of being used in a

portable, "plug and play" atmosphere. The specification, as it stood at the beginning of this thesis effort, was a good first step, but it fell significantly short in containing the required amount of detail to achieve this goal.

Figure 6 depicts the UTAP-defined modules, as well as the permitted interfaces between them. The most serious problem with the UTAP specification was its failure to adequately describe the responsibilities of these modules. For example, Lt Anchor

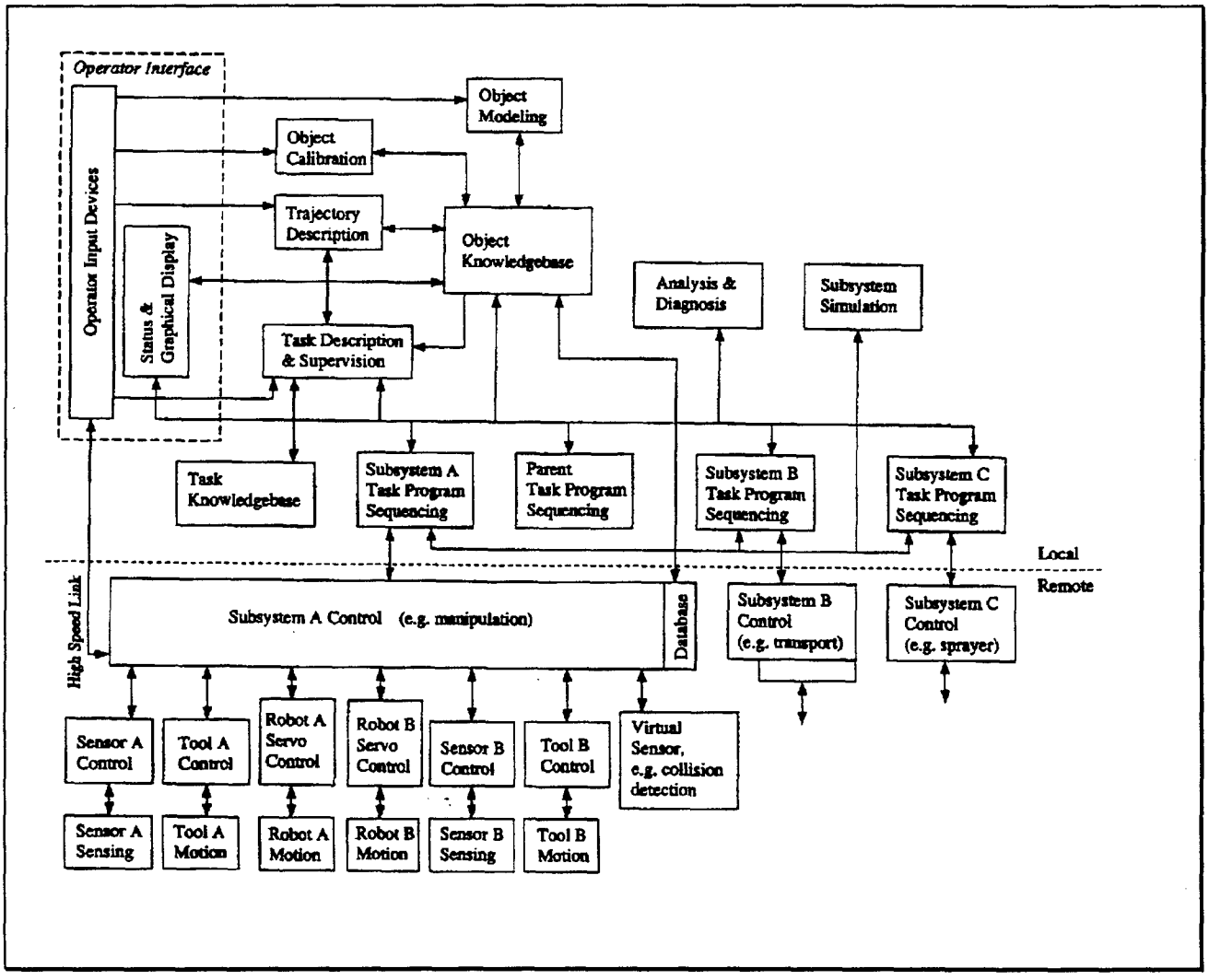

Figure 6: UTAP Architecture [UTA95]

had difficulty determining the functions and responsibilities of the Object Knowledgebase (OK) module. According to the specification, the OK "stores information about objects in the task environment including geometry and task information" [UTA95]. Lt Anchor interpreted this to mean that the OK could be a repository for position values from the various joints of the PUMA 560 manipulator -- a fair assumption if one were to consider the robot joints to be objects in the environment. However, if another designer were to interpret this to mean that the OK should contain only positional information about objects to be manipulated by the robot, an obvious discrepancy would exist. The overall UTAP goal of module portability would be severely hindered. The short description of OK responsibility quoted above was the only such description contained within the UTAP specification. The descriptions for the other nineteen modules were similar in their lack of detail.

The means by which the various modules are to communicate with one another are called UTAP messages. A message is defined by the specification as "an instance (or program) written in the interface language" [UTA95]. Unfortunately, the UTAP specification defines messages in even less detail than it does modules. According to the UTAP specification:

> "The exact semantic meaning of many of the UTAP messages has not been explicitly spelled out in English. For now, the intent and meaning of UTAP Application Programming Interface messages should be apparent from the message name" [UTA95].

A few of the 339 messages listed in the UTAP specification were US_TLC_NURBS_FEED, US_TDS_LOAD_ORIGIN, and US_POST_LIST. The first

two letters of a UTAP message were always 'US'. The next group of letters was the name of the module issuing the message. For instance, the first message listed above contained the letters TLC, meaning Task Level Control. Each message also had an "interface description" in Appendix H of the UTAP specification. The interface description was written in C++ and provided a description of what the message data structure should look like. For example, here is the interface definition for US_TLC_NURBS_FEED:

```
#define US_TLC_NURBS_FEED 677
struct us_tlc_nurbs_feed_msg_t {
        int msgid;
        double sStart;
        double sEnd;
};
```

The interface definition provided merely a cursory listing of what should be included in the message. After that, it was up to the programmer to decide what the message was actually supposed to do. The algorithms to perform the three messages above may be clear to some robotics experts, but the mere names will not provide enough information to ensure portability among modules built by different designers. Furthermore, the specification indicates what modules issue which messages, but not how the different modules should respond to messages. A final issue was raised by Lt Anchor: the UTAP specification said nothing about how these messages were to be passed. If one designer were to pass messages by value, and another by reference, the resulting products would not be able to communicate. In short, a significant amount of detail must be added to the UTAP specification if it is to achieve its ultimate goal.

3.1.2 Operating System Constraints

The operating system used in [Anc95] was a Carnegie Mellon product called Chimera. This operating system was used in the AFIT Robotics lab prior to the onset of UTAP research in 1995. It was chosen for use in [Anc95] for a variety of reasons, including the fact that it was already in use, well understood, and free. It made sense for the first UTAP research to be performed using an operating system that the researchers were familiar with.

Unfortunately, there were many constraints levied on the effort due to the use of Chimera. To begin with, Chimera required each module to follow a very specific format. All modules were executed cyclically and were forced to contain certain functions that were executed at predetermined times in the execution cycle. Furthermore, modules were not permitted to communicate with one another directly. Rather, all inter-module communication occurred via the Chimera Global State Variable Table. Finally, the only existing language support was for an operating system-dependent version of C/C++ using a modified GNU C compiler known as GICC.

Since the goal of UTAP is the development of system-independent, portable robotics software, Lt Anchor had difficulty implementing the UTAP philosophy within the rigid confines of Chimera. Any software developed according to the precise Chimera function naming scheme immediately violated UTAP, and would be incapable

of running under any other operating system. Software written for Chimera, and compiled using GICC, needed extensive modification just to be compilable by any other compiler. The entire UTAP philosophy was predicated on the ability for the various modules to communicate directly with one another via messages. This was impossible under Chimera as well, due to the forced communication through the Global State Variable Table. In short, Chimera and UTAP were a very poor match.

Lt Anchor did manage, however, to device a scheme where UTAP modules could be written to run under Chimera. An interface layer was designed within the Chimera framework. The overall software structure, as well as all hooks to the operating system, were built within the interface layer. After performing basic Chimera initialization, the interface layer would pass the thread of control off to a UTAP-compliant function. These functions performed the UTAP module algorithms and issued UTAP messages when appropriate. When a UTAP message was issued, a call would be made back into the interface layer, and the interface layer would implement the message by updating the Global State Variable Table. This Global State Variable Table was considered to be the implementation of the UTAP Object Knowledgbase.

The basic flow of control can be seen in Figure 7. This figure shows the software in three distinct pieces - the UTAP-compliant modules, the Chimera operating system, and the interface layer. Data would be read from the trackball device by an interface layer routine called int_jtrackball. Since Lt Anchor's intent was for the

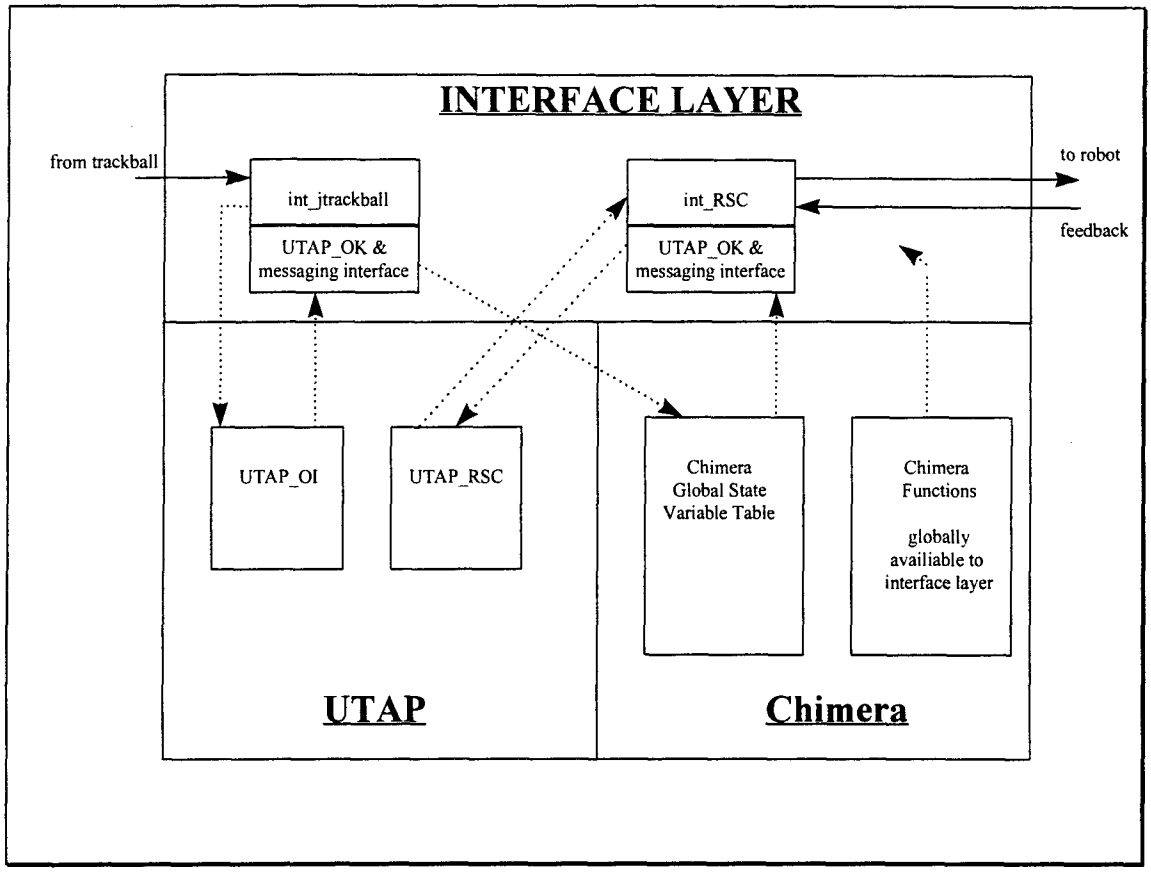

Figure 7: UTAP/ Interface Flow Control

software to be UTAP-compliant, *int_jtrackball* would do little more than pass control off to the UTAP-compliant Operator Interface (UTAP_OI) module. UTAP_OI would interpret the trackball data and send an appropriate message to the UTAP Object Knowledgebase (UTAP_OK). The entire messaging scheme was predicated on giving UTAP_OK direct visibility into the Chimera Global State Variable Table. Not only did this violate the UTAP goal of portability, it required UTAP_OK to actually be located within the interface layer. The UTAP-compliant Robot Servo Control (UTAP_RSC) routine was responsible for sending commands to the robot itself. It would periodically

send a message to the UTAP_OK for trackball generated data. It could then communicate with the robot through an interface routine called *int_RSC*.

The work performed in [Anc95] was tremendously useful in that it was able to demonstrate proof of concept. However, the rigid nature of the operating system made it difficult to envision any long-term UTAP solution using Chimera. Clearly, a better long term solution would involve the implementation of a much more flexible programming environment.

3.2 Research Direction

After conducting the above analysis, a decision had to be made as to the direction of this research effort. While the UTAP specification was not sufficient in detail to achieve its ultimate goal of allowing for the development of portable, independently-developed modules, a significant amount of additional research was possible. Therefore, the remainder of this effort was to concentrate on improving upon the research described in [Anc95]. This would be done by removing those obstacles brought about by the Chimera Operating System, while ignoring those brought about by inadequacies in the UTAP specification itself. Recommendations for UTAP specification improvements can be found in Chapter 5.

The spirit of the UTAP specification was for the various UTAP modules to communicate with one another using language- and operating system-independent constructs. Unfortunately, constraints levied by the existing Chimera operating system

made it necessary to develop UTAP source code that was incapable of running under any operating system other than Chimera.

The decision was made for this effort to focus on replacing Chimera with an real-time operating system (RTOS) that was much more flexible and allowed generic message passing for inter-module communication. Due to the potential software engineering benefits of using Ada, as described in Chapter 2, this research would focus on those operating systems that could support applications written in that particular programming language.

The ultimate long-term goal of this research evolved into a plan to rewrite the current UTAP implementation using Ada tasks. A scheme was envisioned where UTAP processes would be implemented as Ada tasks, and UTAP messages implemented either as Ada rendezvous or operating system function calls. However, before any UTAP-compliant application code could be developed, both a new operating system and the means by which to cross-compile Ada source code needed to be found and installed (regardless of the operating system chosen). Once these were found, the operating system could be installed on the target hardware. Ideally, the chosen operating system would already have a board support package, containing necessary device drivers to operate peripherals on the specific 68030 board in the AFIT Robotics Lab. If not, these device drivers would need to be developed and integrated with the operating system. Finally, there would likely be functionality from the existing UTAP application that would need to be invoked by RTEMS. Specifically, a means by which

to download executable files from the host to the target board would be needed (the Chimera operating system's *chim* program should be adequate). Also, RTEMS must be made able to perform robot calibration (these scripts already exist under Chimera, and may need to be translated). Once a complete operating system and compiler package are in place, the UTAP source code found in [Anc95] can be reengineered into Ada tasks, compiled, and tested on the target hardware. This turned out to be a tremendously aggressive set of tasks for a single thesis; therefore, not all steps could be accomplished within the scope of this effort. Section 4.1 describes the specific steps that were accomplished.

3.3 Summary

This chapter described several problems previously encountered in implementing the UTAP specification. Two main problems are discussed: lack of needed detail in the specification itself, and operating system constraints. A research direction is then proposed. It is suggested that replacing the operating system, and concentrating on a long-term UTAP solution using Ada tasks will ultimately alleviate many of the problems encountered in [Anc95]. The next chapter describes the successes and difficulties encountered in performing the first several steps of this long-term effort.

4. Results

This chapter describes how the first several steps of the research proposed in the previous chapter were performed. The key software products and documentation that resulted from this effort are then described.

4.1 Completing the First Steps

4.1.1 Choose an RTOS

The initial step in this effort was to choose a new RTOS to implement in the robotics lab. Several factors were used in making this decision, including Ada support, hardware compatibility, technical support availability, technical support cost, and product cost. Using these criteria, the options were narrowed to the two commercial and two government-owned products discussed in Chapter Two.

The two commercial products, VXWorks and LynxOS, both seemed to be workable, albeit expensive, choices. Both would be able to support Ada (using software purchased from third party vendors), and both had versions which ran on a 68030 platform. However, neither had off-the-shelf device drivers available for the specific 68030 board in the robotics lab. In the case of VxWorks, these drivers would have had to be developed from scratch. Lynx suggested either replacing the hardware, or purchasing a porting kit, both expensive propositions. In the end, fiscal constraints precluded the use of either of these alternatives.

AARTS was also a promising alternative. Being government-owned, the source code was available to AFIT free of charge. AARTS was written in Ada and fully supported the Ada tasking model. A version existed for use on the 68030 processor, but again, device drivers were not immediately available. The vendor strongly discouraged attempting to develop these drivers at AFIT, but did propose developing them for a fee. Again, this fee proved too high, and AARTS was eliminated from further consideration.

RTEMS, another government-owned operating system, seemed to be a sound choice. It was written in Ada, and had been used in a number of real-time applications around the world. Although, once again, device drivers were not immediately available, the vendor was confident they would not be difficult to write. Technical support was offered at an affordable price. The one drawback with RTEMS was that it did not yet fully support the Ada95 tasking model. With the available version of RTEMS, concurrent tasks were implemented as calls to low-level RTEMS library routines. These routines were not likely to be included in the next version of RTEMS, scheduled to be released in late 1996. The full Ada95 tasking model was supposed to have been supported under this new version of RTEMS; however, the timing of the release of this new version precluded its use in this research. Any tasking applications written for this effort would have needed to be modified extensively to run under the new RTEMS, once available.

Ultimately, RTEMS was selected, and the choice was made for this thesis effort to concentrate on installing the operating system, developing the necessary drivers, and, time permitting, writing non-tasking applications. Tasking applications can be added as a follow-on effort to this thesis under the next version of RTEMS.

4.1.2 Find and Implement Ada Cross-Compiler

RTEMS was not available in an immediately-executable format. Instead, the RTEMS source code had to be downloaded and compiled for the target platform. Therefore, a cross-compiler was needed not only to compile application code, but also to compile the operating system itself. Since the AFIT Robotics Laboratory did not yet have a compiler to cross-compile Ada source code from the host platform to the target, the next step was to find and implement this functionality. On-Line Applications Research Corporation, the RTEMS support vendor, strongly recommended the GNU Ada Translator (GNAT) to perform the cross-compilation from the Sparc workstation to the embedded 68030 platform. GNAT was freeware and was very closely related to the popular GNU C Compiler (GCC). In fact, to build GNAT executable from source code, the entire GCC source code directory was also needed. The source code for GCC and other needed accessories were readily available from several FTP sites. As a rule, GCC must be recompiled to include cross-compilation functionality, at which point cross-compilation can be performed using flags on the GCC command line.

Figure 8 shows the general flow of Ada source code through the GNAT family of products. The GNU Ada Run-time Library (GNARL) provides the interface to the

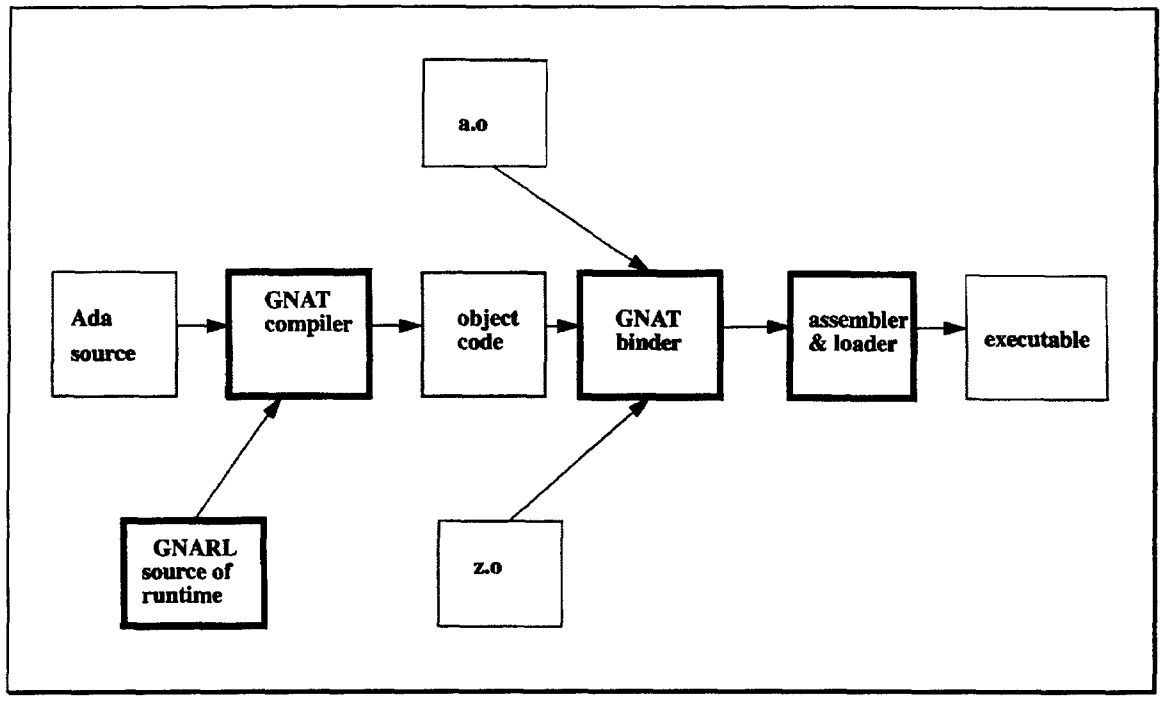

Figure 8: The GNAT System [GNAT94]

run-time environment. The GNAT compiler generates object code which is linked using the GNAT binder. The assembler and loader then produce the executable product.

The GNAT compiler itself is further decomposed in Figure 9. The Ada source code is structured into an abstract syntax tree (AST) and expanded. The Gigi phase is responsible for translating the Ada AST into a corresponding C AST. This AST is then provided to GCC to build the object code.

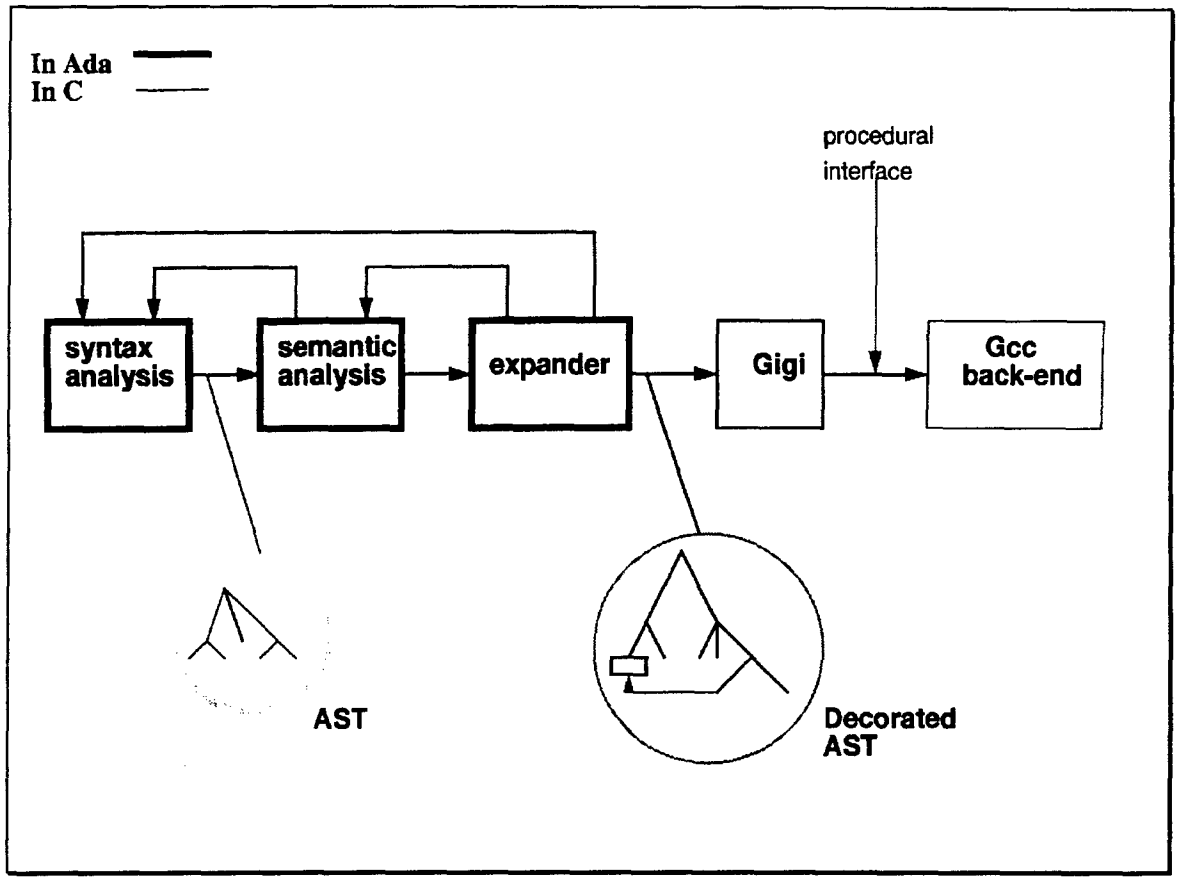

Figure 9: The Phases of GNAT [GNAT94]

Installing GNAT as a cross-compiler took much longer than expected. Appendix A enumerates the many lessons learned in accomplishing this part of the effort. Appendix B lists the step-by-step procedure that was eventually developed to build the GNAT cross-compiler.

4.1.3 Use Cross-Compiler to Compile RTEMS Source Code

Once the cross-compiler was compiled and installed, the next step was to compile RTEMS. Since the device drivers for the specific 68030 board being used had

not yet been developed, the decision was made to first compile RTEMS with a collection of device drivers that was known to successfully compile on a different platform. The resulting executable version of the operating system could then be sent to the vendor to see if would run correctly on the target board, in this case a Motorola MVME136 (68020-based). The first step was to modify the RTEMS compilation configuration files within the source code directory tree to include the necessary device drivers (see Appendix C for specifics). Upon compilation, a new problem was discovered. The last time the RTEMS source code was compiled under GNAT, the most current version of the compiler was GNAT 2.02, a version no longer supported or maintained by the GNAT support vendor. Attempting to compile RTEMS with GNAT 3.05 resulted in many coding errors being found that GNAT 2.02 had missed. Unfortunately, with each compilation of the many files that make up the RTEMS source code, only an error or two would be found before the compilation would halt. After each attempt, the error or two that was found would need to be investigated to determine the best fix. In some cases, the solutions were fairly simple, and took little time to implement. In other cases, finding the solution required a more detailed understanding how the source code was designed. Once the source code compiled correctly, many linking errors were encountered due to changes in the GNAT 3.05 libraries, as compared to the previous libraries. A complete list of RTEMS source code and makefile modifications needed for compilation can be found in Appendix D of this document.

Ultimately, RTEMS version 3.2.1 was successfully compiled and linked for the MVME-136 board using GNAT 3.05. Because the AFIT Robotics lab did not have the proper hardware for testing, several executable products were sent to the vendor for testing on their hardware. Although running the "hello.exe" executable provided with the RTEMS source code on the target platform did result in a "Program Error" being raised, the mere fact that the operating system and device drivers were capable of writing this message to the screen is encouraging. This demonstrates that the cross-compiler, as well as the device drivers, do, in fact, work. The source of the run-time error may be the product of a GNAT error that has since been solved for GNAT 3.07. GNAT 3.07 was not available for this particular effort.

4.2 Products

Two key products that will eventually make up the run-time infrastructure of a UTAP-compliant system are provided as the result of this thesis effort. First, a working cross-compiler, capable of compiling Ada code for the AFIT Robotics Laboratory, has been developed and installed. This compiler can be found in the AFIT Hawkeye Laboratory and is available for future UTAP research. The second software product is a fully compiled version of RTEMS and it's source code. This product was also built in the AFIT Hawkeye Laboratory. This version has been compiled with the Motorola MVME136, 68020-based board support package. Although this is not the exact board that is located in the AFIT Robotics Laboratory, the device driver and

board support package source code can be studied and modified for use in the future research.

4.3 Documentation

This thesis effort has also resulted in the development of several important pieces of documentation. Although it was known that the GNU Ada Translator was capable of being configured to perform cross-compilation, there was no written documentation on how to do so. A detailed, 97-step procedure now exists and can be found in Appendix B of this document.

Since RTEMS had not yet been compiled using the latest version of GNAT, several hurdles were cleared in fixing coding errors that remained in the RTEMS software itself. A document containing a complete list of corrections needed to compile the latest version of RTEMS using GNAT 3.05 now exists and can be found in Appendix D of this document.

Finally, this thesis documents a viable idea for making future UTAP implementations less system-dependent. By developing UTAP modules using Ada tasks, and implementing messages using Ada rendezvous, future UTAP implementations will not need to rely on operating system constructs to provide these services. Now that most of the RTEMS infrastructure is in place, the potential exists for UTAP to be implemented within the software engineering-friendly confines of Ada, rather than the rigid constraints of Chimera.

4.4 Long Term Benefits

The potential long term benefits of this research are numerous. As with most computers, a significant portion of the cost of telerobotics systems resides in software development. If a means by which to decrease the cost of software through reuse can be made widely available, the cost of telerobotics systems can be brought down significantly. Even if the RTEMS-GNAT solution proposed in this thesis proves not to be a satisfactory long term solution for implementing UTAP, it does at least address the question of UTAP portability. If UTAP is to become a standard under which telerobotics systems are built, its portability must be established. This thesis represents an important first step toward accomplishing this.

4.5 Summary

This chapter has described the specific steps performed for this effort and enumerated the results. The two main software products were GNAT and RTEMS. The three key documentation results were a procedure for building cross-GNAT, a plan for developing UTAP in Ada, and a listing of RTEMS source code corrections for compilation under the latest version of GNAT. The final chapter will discuss the overall conclusions and recommendations for further research.

5. Conclusions and Recommendations

5.1 Conclusions

There are several conclusions that can be drawn from the work described in the previous chapters. First, the UTAP specification is an admirable attempt to address the need for a standard interface for telerobotics applications. However, the specification, as it stands today, lacks the level of detail needed to achieve its ultimate goal. Specific recommendations for improvement are found in the next section.

A second conclusion that can be drawn is that validating portability of the UTAP specification is a non-trivial exercise. In order to verify that applications developed under UTAP are truly portable, a researcher must not only develop compliant code, but also develop and/ or install different environments for the application code to be tested in. Although developing robotics code using unique or proprietary languages and operating system constructs may significantly simplify the research, this approach violates the underlying philosophy of UTAP. Also, developing the compilation and run-time environments in which to test UTAP applications takes a great deal of effort - effort that is seemingly unrelated to the research of UTAP itself. Yet, research of this type is crucial if the ultimate goals of UTAP are to someday be realized.

5.2 Recommendations for Further Research

5.2.1 UTAP Specification Recommended Improvements

The intent of this section is to recommend specific improvements to the UTAP specification. At this time, the specification is in its infancy, so problems are to be expected. Chapter 3 of this thesis lists some of the problems encountered from an application developers perspective. The UTAP specification developers deliberately left out many details describing the various UTAP constructs. This was probably done to reduce the number of constraints a developer would encounter. However, application developers need a certain amount of detail if they are to design compliant software that is truly portable.

First, an effort should be made to more precisely define the module responsibilities. At this time, all that exists in the UTAP specification is Annex B, Section 1, a five page description of all twenty modules. Each module has a one sentence description of its responsibilities. This is not sufficient to ensure portability of modules designed by different engineers in different environments. This section would be enhanced significantly if simple algorithms were included to better describe what the modules are actually supposed to do. Therefore, the addition of module algorithms and detailed descriptions to Annex B is strongly recommended.

Secondly, the UTAP specification lists several hundred standard message types that are supposed to define how all the modules communicate with one another. Once again, the UTAP specification lacks sufficient detail to fully define these interfaces. Annex H lists each message and the data which is supposed to be sent along with it. The specification does not specify what actions a module is to take upon receiving a specific message. Although it would add a certain amount of bulk to the specification, the inclusion of message definition is necessary if the ultimate goals of UTAP are to be achieved. Therefore, adding message algorithms and definitions to Annex H is also strongly recommended.

5.2.2 Follow-on Efforts

In this research, an initial effort was made toward developing an environment that could accommodate UTAP applications written in standard Ada, void of operating system-dependencies that would violate the UTAP goal of portability. Unfortunately, developing this environment took more time than available in a single thesis effort. Therefore, extensive follow-on research is recommended.

The most important next step will be to reassess the assumptions made at the beginning of this effort and determine if Ada/ GNAT/ RTEMS is the best environment in which to pursue long term UTAP research. Since most of the infrastructure is already in place, it may be prudent to continue this effort and begin to develop the necessary device drivers for use in the AFIT Robotics Lab. A second option might be

to acquire the C version of RTEMS and determine if Ada UTAP application code can be developed using it. A third option is to abandon the RTEMS/ GNAT line of research altogether, and purchase a commercial operating system and cross-compiler.

If the decision is made to continue the GNAT/ RTEMS effort described in Chapters 3 and 4 of this document, the next step will be to develop device drivers for the AFIT laboratory hardware. First, the researcher should determine what can be learned from the existing MVME136 drivers, and make the necessary changes. Once working device drivers exist, the system can be tested to ensure that application source code can effectively communicate with the PUMA 560 robot. As stated in Chapter 3, before writing any application code, it will be necessary to convert some of the Chimera robotics infrastructure (such as calibration, etc . . .) to be able to be invoked by RTEMS. This research has hypothesized that the *chim* program supplied by Chimera will be sufficient for downloading compiled binaries to the 68030 boards. This must be verified. Finally, the UTAP application code developed for [Anc95] needs to be redesigned into Ada tasks, compiled, run under RTEMS, and tested on the laboratory hardware. Since both Ada and C cross-compilers have been installed as part of this effort, mixed language UTAP applications can be studied as well.

Eventually, it will be necessary to develop other environments in which to test the UTAP application software. While RTEMS may prove to be a good environment to develop long-term UTAP solutions, it will still be important to demonstrate

portability by using different operating systems, hardware platforms, and even languages.

It will also be necessary to devote some time toward keeping up with the pace of the GNAT and RTEMS technology. In the next few months, it is likely that a new version of RTEMS will be released that will fully embrace the Ada95 tasking model. This will be an important step because it should allow UTAP researchers to develop Ada application code that is completely independent of the operating system. The GNAT compiler has new releases every several months, and it is possible that with these new products, cross-compilation support will be better addressed. Since detailed instructions now exist to build the GNAT cross-compiler, future attempts to update this functionality will be significantly easier. The findings of this research will be made available to both the RTEMS and GNAT vendors for inclusion in future products.

5.3 Summary

This chapter has summarized the conclusions and recommendations of this thesis effort. The UTAP specification represents a good first attempt to standardize telerobotics interfaces, but suffers a few shortcomings. Adding detail to the definitions of UTAP's modules and messages is recommended. Developing suitable environments in which to test UTAP applications is not a trivial exercise, but an important one if the goals of UTAP are to someday be realized.

Several recommendations for further research are made. The assumptions made at the beginning of this effort should be reassessed and a decision made if the integration of RTEMS and UTAP should continue. If not, other environments should be developed or purchased to continue Ada-UTAP research. The task communication constructs featured in the Ada programming language will eventually help in solving the need to look to the operating system for inter-module communication. Effort should be made to keep up with the pace of GNAT and RTEMS technology, as these products may result in a good long term UTAP solution.

Bibliography

[ACG95] DaCosta, F., and others. "Toward Rapid Implementation of Robotics Systems; Preliminary Draft Report," Advanced Cybernetics Group, Sunnyvale, California; 1995.

[Acu94] Acuff, Phillip R., "A Reusable Ada Real-Time Multiprocessing Executive for Military Systems," *Crosstalk: The Journal of Defense Software Engineering*, Software Technology Support Center, Hill AFB, Utah; December 1994.

[Acu96] Acuff and O'Guin, "RTEMS: A Technology Transfer Success Story," *CD-ROM Proceedings from Software Technology Conference '96*, Salt Lake City, Utah; April 1996.

[Ada96] Ada Success Stories, World Wide Web Page, http://www.seas.gwu.edu/faculty/mfeldman/success.html.

[AF94] Air Force Software Technology Support Center, *Guidelines for Successful Acquisition and Management of Software Intensive Systems*, September 1994.

[Anc95] Anchor, Kevin, "Design and Evaluation of Standard Telerobotic Control Software," MS Thesis AFIT/GE/ENG/95D-01, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1995.

[AART95] "Software User's Manual for the AARTS Operating System," Document Number MSTA15102, Contract Number F33615-87-D-1452. Prepared by American Computational Technical Services, Inc. for Wright Laboratories, Wright Patterson AFB, Ohio; August 1995.

[Coh96] Cohen, Norman H., *Ada as a Second Language*, McGraw Hill, New York, New York; 1996.

[GNAT94] Schonberg, Ed and Banner, Bernard, "The GNAT Project: A GNU 9X Compiler," *Proceedings of Tri-Ada '94*, Baltimore, Maryland; 1994.

[June96] June, Matthew L., "Analysis and Design of Standard Telerobotic Control Software," MS Thesis AFIT/GCS/ENG/96D-13, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1996.

[Lyn96] Lynx Real-Time Systems, World Wide Web Page, http://www.lynx.com/faq.html.

[Mie95] Miedlar, Mike, Electronic Mail Message, American Computational, Dayton, Ohio; December 1995.

[Pre87] Pressman, Roger, *Software Engineering: A Practitioner's Approach*, McGraw Hill, New York, New York; 1987.

[RACE96] Kelly AFB Robotics and Automation Center of Excellence, World Wide Web Page, http://www.kelly-afb.org/links/orgs/race.html.

[RTE96] United States Army Missile Command, Real-Time Executive for Military Systems (RTEMS) World Wide Web Page, http://lancelot.gcs.redstone.army.mil/rg4/rtems.html.

[Tay90] Taylor, Paul, *Understanding Robotics*, CRC Press, Hong Kong; 1990.

[UTA95] National Institute of Standards and Technology (NIST) Intelligent Systems Division, *Unified Telerobotic Architecture Project (UTAP) Standard Interface Environment*, 1995.

[Volz84] Volz, Mudge and Gal, "Using Ada as a Programming Language for Robot-Based Manufacturing Cells," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-14, No. 6; November/December 1984.

[Volz87] Volz and Naylor, "Design of Integrated Manufacturing Control Software," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-17, No. 11; November/December 1987.

[Volz88] Volz and Naylor, "Integrated Manufacturing Software in Ada," *Procededings from Conference on Computer Controlled Mechanics*, Stockholm, Sweden; May 1988.

[Volz96] Volz, Richard A., Electronic Mail Message, Texas A&M University, College Station, Texas; May 1996.

[Web84] *Webster's New World Dictionary*, Simon and Schuster, New York, New York; 1984.

[WRS96] Wind River Systems, World Wide Web Page, http://www.wrs.com.

Vita

Capt Kendall M. Lemley, Jr. [redacted] He graduated from Fort Knox High School in 1988 and entered undergraduate studies at the United States Air Force Academy. He graduated with a Bachelor of Science degree in Computer Science on 27 May 1992 and received his commission the same day.

His first assignment was to Wright Patterson Air Force Base, Ohio as an aircraft propulsion development engineer in the Flight Systems Directorate of the Aeronautical Systems Center. He then moved to the B-2 System Program Office where he worked as a support systems engineer and project officer for B-2 software systems. In May 1995, he entered the School of Engineering at the Air Force Institute of Technology.

[redacted]

Appendix A

Lessons Learned and Pitfalls In Building Cross-GNAT

A.1 Immaturity of Cross-GNAT

When GNAT was chosen to perform cross-compilation for this thesis effort, it's level of maturity in this area was not known. Upon investigating how to set up GNAT to provide cross-compilation, it became clear that its use for this purpose is a very immature technology. Ideally, one might hope that cross-compilation could be accomplished simply by changing the configuration of a binary GNAT distribution. Unfortunately, this is not the case. One must first insure that the latest version of the native GNAT compiler is installed, then use that compiler to recompile GNAT source code with the configuration set to build a cross-compiler. The specific steps for building the cross-compiler are found in Appendix B of this document.

There is not yet any documentation, either published or on the World Wide Web, on how to build cross-GNAT. There is plenty on how to build cross-GCC, and plenty on how to build native GNAT; however, there are several source code changes that are needed before one can combine the two procedures to build cross-GNAT. At this time, to build cross-GNAT, one must either become intimately familiar with the over 300,000 lines of GNAT source code and makefiles, or find someone who has done it before, and get a list of source code and makefile changes needed. Given a

UNIX diff output, one can modify existing GNAT source code using the GNU patch utility.

Finding someone who has built cross-GNAT before is not easy. Posting inquiries to applicable mailing lists and newsgroups (discussed later in this section) resulted in very few responses. The first response confirmed that the first step was to build cross-GCC and a Cygnus product called Newlib for the target. However, it was very vague about what specific source code and makefile changes would be needed for the GNAT source code. Apparently, if these changes are not made, the cross-compiler begins to be compiled by partially-compiled pieces of itself, and will not work. The second response was much more detailed, but it assumed that permanent storage and certain GNU tools were available on the target platform. Unfortunately, this is not the case in the laboratory being used for this research. In this case, the target machine did not have any permanent storage capability, so all executable programs must be downloaded from the host. This capability will ultimately be provided by a downloading utility called *chim*, which runs on the host machine.

Fortunately, the RTEMS vendor, On-line Applications Research Corporation (OAR), was able to provide support in building cross-GNAT. They had already built cross-GNAT (albeit an earlier version), and had originally recommended it for use in compiling their operating system. With their help, the procedure found in Appendix B was developed.

The bottom line is that, at this time, cross-GNAT is very immature. Finding someone who can help is difficult, and finding documentation on how to build it is impossible. This document is the first known effort to annotate the steps and pitfalls associated with building cross-GNAT. In time, as OAR continues to recommend cross-GNAT for compiling RTEMS, and work on incorporating the needed patches into the GNAT public releases, this process will improve.

A.2 Lack of Available Support

When Ada Core (the support provider for GNAT) was contacted about trying to build cross-GNAT, the following response was received:

> "Please note that our support under this customer number does NOT include support for cross-compilation. It is definitely possible to do the kind of cross-compilation you are interested in, but this requires a quite different level of support (cross-compilation is much more complex than native compilation)."

It turns out that Ada Core does not provide any support for cross-GNAT whatsoever. Cygnus provides support for cross-GCC via the Internet, but does not support cross-GNAT either. They are however, interested in at least incorporating the basic steps of building cross-GNAT in their Frequently Asked Questions (FAQ) listing. At this time, OAR does not officially provide cross-GNAT support, although they seem to have the capability and are negotiating with Ada Core to be able to do so.

A.3 Problems with the Cygnus One-Tree FAQ

Building the cross-GCC compiler and Cygnus Newlib C library are prerequisites for building cross-GNAT. The Cygnus cross-GCC FAQ listing is the best source for accomplishing these important steps. The FAQ is readily available at the Cygnus FTP site (ftp.cygnus.com). It describes a procedure for building cross-GCC from a single source code tree. For this research, the single tree procedure was abandoned after the first several compilation attempts, so that the precise location of compilation errors could be more easily discovered. However, after developing the procedure found in Appendix B of this document, the single tree procedure was rerun to see if it would work. For reasons unknown, it did not. For anyone attempting to build cross-GCC, the Appendix B procedure is strongly recommended.

A.4 Native GNU Tools

The single biggest problem currently with the cross-GCC FAQ is its failure to mention what native GNU tools are needed to properly complete the procedure. Based on experience with building cross-GNAT with a sparc-sun-solaris2.5 host targeted to m68k-coff, the following native tools are needed to even begin executing the FAQ procedure:

GNAT (includes GCC)
GNU BINUTILS
GNU MAKE
GNU PATCH

Furthermore, it is best to build these tools from the readily available source code, rather than simply download and install a binary executable. The reasons are that first, it gives the user confidence that the tool will actually work on the machine on which it compiles. Second, it gives the user practice building tools from source code, an important aspect of setting up a cross-compiler. Instructions for building all needed native tools from source code can be found in the cross-GNAT procedure in Appendix B. Instructions for installation of GNU Patch are not included in this document, as this tool was already available on the host platform used in this research. It is critical, however, to ensure that the GNU version of Patch is installed rather than the UNIX version (the GNU version works on both Solaris and SunOS, whereas the UNIX version works only sporadically on SunOS).

A.5 Build Directories vs. Source Directories

The Cygnus One-Tree FAQ also recommends performing all compilation in directories separate from the directories containing the source code. The reason for this is to prevent compilation attempts that go awry from contaminating the source directories with partially or improperly compiled files. However, the cause of several early errors in this research were more easily found by compiling in the actual source code directories. Therefore, the procedure for building cross-GCC found in this document deviates from the FAQ in this regard. If a compilation attempt does result in

an error, it is easy to restore the directory to its initial state using the *make distclean* command.

A.6 Native and Cross Tools in the Same Installation Directory

For cross-GCC, it is apparently acceptable to install the cross-compiler in the exact same directory as the native compiler. One can then invoke the cross-compiler with the same "gcc" command by including the appropriate flags in the GCC command line ("gcc -bm68k-coff -m68030 foo.c" would be used to compile foo.c for a 68030 board). Unfortunately, cross-GNAT cannot currently be installed in the same directory as its native counterpart. Many of the native tools and libraries get overwritten and neither compiler will work. This problem was solved by installing all native tools in one sub-directory, and cross tools in another.

A.7 Amount of Memory/ Compile Times Needed

In order to build cross-GNAT a significant amount of memory and compilation time are needed. For example, compiling Cygnus newlib-1.7.0 takes three to five hours on a Sun Sparc-5 machine and creates a directory tree that occupies about 63 megabytes of memory. By the time cross-GNAT was completely built, the *gnu* directory took up nearly 350 megabytes of disk space. One must ensure that a significant amount of disk space is available before attempting to build cross-GNAT.

The lengthy compile times tend to magnify the effects of errors. For example, two files were missing in newlib-1.7.0 directory structure (the solution is discussed later). However, newlib had to be compiled approximately eight times before the error was diagnosed and the solution found. The nature of the error was such that the compilation process had to be repeated from scratch each time a different solution was attempted. At four hours per compilation attempt on average, the better part of a week was spent to solve this seemingly simple error.

A.8 Testing the Cross-Compiler

As was mentioned before, the cross-compiler is very immature. For example, using m68k-coff-gcc with the -c flag generates the .o and .ali files, as expected. However, gnatmake and gnatbind do not yet work for cross. When compiling a simple native Ada program, one usually can just type "gnatmake xxx.adb", and the compiler will generate not only the object files, but link in all necessary libraries and generate the executable as well. To do the same thing for cross, one must figure out which libraries are needed and manually link them using GNU and UNIX tools. This is a fairly complicated process, especially given the need to navigate the maze of libraries created in building the cross-compiler. Fortunately, the details of doing so are hidden within the RTEMS makefiles.

Fully testing the newly installed cross-compiler was not trivial. Since an operating system was not yet running on the target, any test programs developed could

not be run until RTEMS was compiled. Furthermore, nothing useful could be done until board support packages and device drivers were created for our specific board, since output to the user is dependent on being able to communicate with a terminal. All in all, this created a very frustrating catch-22. One could not tell for sure that the cross-compiler was working until RTEMS was compiled, yet RTEMS could not be successfully compiled without a working cross-compiler. The best that could be done was to cross-compile simple Ada programs using no IO, and see that they compiled without error. Until RTEMS was compiled, they could not be run on the target platform.

A.9 Determining the Host and Target Platforms

Two options were available for use as a host platform: Sun Sparc Workstations running either the SunOS 4.1.3, or Solaris 2.5 operating systems. Although the target platform was connected directly to a SunOS machine, the Solaris host was chosen to be used instead. The reasoning was twofold. First, most of the early success for this effort in building and installing GNU tools came using Solaris machines. Second, the laboratory being used in this effort was in the process of having SunOS replaced with Solaris on nearly all machines. Although, the machine connected directly to the robot was not slated to receive Solaris, this may well happen in the future. Furthermore, as long as the directly connected host remains SunOS, files can easily be compiled on a

Solaris machine, uploaded to the SunOS machine, and then downloaded to the robotics platform.

A discussion of how to specify particular platforms can be found in the GNU CC source code top level README file. Platforms are specified on the *configure* command line using "--host=*hostname*" and "-- target=*targetname*". Generally, the host machine need not be specified, as configure is usually able to figure it out on its own. The target can either be specified using "CPU-company-OS" (example: vax-dec-ultrix4.2 or sparc-sun-solaris2.4), or a recognized machine name (example: powerpc or sun4). For any CPU's in the Motorola 68000 series, *m68k-OS* is used. Since an operating system was not yet running on the target platform used in this effort (cross-GNAT will be used to compile an operating system), *m68k-coff* was used as the target.

A.10 Missing Files in Newlib-1.7.0

In order to compile Newlib-1.7.0, an error that was not yet documented in any FAQ or README files needed to be solved. The first several times compilation of Newlib-1.7.0 was attempted, the following error message was received:

```
configure: error: can not find install-sh or install.sh in ../.. ./../..
configure: error: ./configure failed for m68k
*** Error code 1
make: Fatal error: Command failed for target `configure-target-libgloss'
```

Mr. Joel Sherrill at OAR provided the solution to this problem. It involved merely linking the *install.sh* file from the newlib-1.7.0 directory into the *m68k-coff*

sub-directory. The next error was similar, but took a post to the crossgcc mailing list to find a solution:

> configure: error: can not run ./../config.sub
> configure: error: ./configure failed for m68k
> gmake: *** [configure-target-libgloss] Error 1

This solution was sent by Mr. Franz Fischer, a professor of computer science from Munich, Germany. The *config.sub* file in the newlib-1.7.0 directory needed to be linked into *m68k-coff* as well. Both solutions to these previously undocumented errors can be found in the procedure for building a GNAT cross-compiler (Appendix B).

A.11 Needing Crossgcc and Newlib Installed to Build Cross-GNAT

In order to build native GNAT from source code, one does not need to compile GCC from source first. The GNAT source code is copied into the GCC source code directory, and both compilers are built with a single command. The same could not be said for compiling cross-GNAT. To build cross-GNAT, cross-GCC and Newlib must both already be compiled and installed. Not doing so results in the following error:

> a-adaint.c:42: time.h: No such file or directory

Only after cross-GCC and Newlib are installed, can the GNAT source code be copied into a clean GCC source code directory, compiled and installed.

A.12 Join the Following E-Mail Lists

Two E-mail lists were very beneficial in finding help throughout this research. The first is operated by Cygnus and dedicated entirely to cross-GCC and embedded issues. To subscribe, send E-mail to *majordomo@cygnus.com* with "subscribe crossgcc your@email.address" in the body of the message.

Another important list to join is the GNAT-Chat list. This list is dedicated to the discussion of Ada in general and GNAT in particular. To subscribe send E-mail to *chat-request@gnat.com* with "subscribe your@email.address" in the subject line.

At this time, there is no list dedicated to cross-GNAT issues, although this may change in the future. It is possible to find a small handful of people on the above two lists that have built cross-GNAT and may have more information on any new mailing lists.

Appendix B

Instructions for Building the GNAT Cross-Compiler

The following instructions guide the reader through the process of setting up GNAT to handle cross-compilation. These instructions were used to build a cross-compiler between a host Sparc workstation running the Solaris 2.5 operating system, and a generic Motorola m68000 series target processor. Once the cross-compiler is built, Ada source code can be compiled for the 68030 board using **m68k-coff-gcc -c -m68030 foo.adb** on the Solaris terminal command line. Boldface type is used throughout this appendix to indicate exact keystrokes. Italics are used to designate directory paths and file names.

B.1 Native Tools

B.1.1 Install GNAT Binary Distribution

The first step toward building an Ada cross-compiler requires installation of a native compiler so that the compiler source code can be recompiled. If GNAT is already installed on the host system, this step may not be necessary. However, be wary, because older versions of GNAT will not necessarily compile newer versions. Installing the latest version of the native GNAT compiler is recommended to avoid potential problems with compiling cross-GNAT.

1. Get the latest binary GNAT distribution from ftp site cs.nyu.edu:pub/gnat. *Gnat-3.05-sparc-sun-solaris2.4-bin.tar.gz* was used for this research.

2. Create a directory called *gnat-bin*.

3. Move the downloaded GNAT binary file to *gnat-bin*.

4. Type **cd gnat-bin**

5. Type **gunzip gnat-3.05-sparc-sun-solaris2.4-bin.tar.gz** to uncompress file

6. Type **tar xf gnat-3.05-sparc-sun-solaris2.4-bin.tar** to untar GNAT

7. Type **cd gnat-3.05-sparc-sun-solaris2.4-bin**

8. Type **doconfig** and follow directions. The third option is recommended to install GNAT in the user's personal directory space.

9. When prompted for an installation point, *~username/gnat-bin/install* is recommended. Note: The entire path name must be typed (i.e. do not use ~username nomenclature).

10. Type **doinstall** to install GNAT

11. The installation process will create a file called *env-vals*, the text of which must be added to the user's .cshrc file (in the top level directory *~username*).

12. Add *set path = (~username/gnat-bin/install/bin $path)* to the end of the .cshrc file.

13. Type **source .cshrc** for changes to take effect

14. At this point native GNU Ada and C compilers have been installed. Test the setup by compiling simple programs. Using the -v option on the GCC compiler command line will allow the user to ensure the newly installed compiler is the one being used.

B.1.2 Install GNU Make

Having the latest version of GNU make is critical. Some of the later steps of this process will work fine using UNIX make; however, many will not.

1. Get the latest version of GNU make from the GNU ftp site (prep.ai.mit.edu). *Make-3.74.tar.gz* was used in this research.

2. Create a directory called *gnu*.

3. In the new *gnu* directory, create a new directory called *native-install*. This is where the rest of the native GNU tools that are needed to build a cross-compiler will be installed.

4. In the *gnu* directory, **gunzip** and **tar xf** the file that was downloaded from the ftp site.

5. Type **cd make-3.74**

6. Type **configure --prefix= ~username/gnu/native-install**
Note: The entire path name must be typed (i.e. do not use ~username nomenclature). The --prefix flag tells the makefiles to install GNU make in the specified directory.

7. Type **make** to build GNU make

8. Type **make install** to install GNU make in *~username/gnu/native-install*.

9. At this point, a working version of GNU make will be available. Change directories to *~username/gnu/native-install/bin* and type **ln make gmake**. This is important because some of the scripts used later invoke gnu make by calling "gmake" rather than "make".

10. Add *set path = (~username/gnu/native-install/bin $path)* to the end of the .cshrc file.

11. Type **source .cshrc** for changes to take effect.

B.1.3 Install GNU Utilities

Again, this step may not be necessary if all the latest GNU utilities are available on the system being used. Most of the later steps of this process should work fine

using UNIX utilities. However, for the sake of having a consistent set of GNU tools, binutils was built and installed for this research.

1. Get the latest version of GNU binutils from the GNU ftp site (prep.ai.mit.mil). *Binutils-2.7.tar.gz* was used for this research.

2. In the *gnu* directory, **gunzip** and **tar xf** the file that was downloaded from the ftp site.

3. Type **cd binutils-2.7**

4. Type **configure --prefix= ~username/gnu/native-install**
   Note: The entire path name must be typed (i.e. do not use ~username nomenclature).

5. Type **gmake all** to build GNU binutils

6. Type **gmake install** to install GNU binutils in *~username/gnu/native-install.*

7. At this point, the native versions of all needed GNU compilation utilities are available for use. Modifying the .cshrc file should not be necessary, as long as *~username/gnu/native-install/bin* is already in the path (done in the last section). It is recommended, however, that **source .cshrc** be typed before moving to the next section.

B.1.4 Install GCC and GNAT Compilers

The next step was to build native GCC and GNAT compilers from source code. This is probably not entirely necessary, given that binary executable versions were already installed in section A. However, it is useful to ensure that a native compiler can be built from source before proceeding to the more complicated task of building a cross-compiler from source. If this step cannot be successfully accomplished, there is no hope of successfully building the cross-compiler from source code, since the steps are similar.

1. Get the latest version of GCC source code from the GNU ftp site (prep.ai.mit.mil). *Gcc-2.7.2.tar.gz* was used for this research.

2. Get the latest version of GNAT source code from the GNU ftp site (prep.ai.mit.mil). *Gnat-3.05-src.tar.gz* was used for this research.

3. Get the GNAT source patches from the GNU ftp site (prep.ai.mit.mil). *Gnat-3.05-src-update.tar.gz* was used for this research.

4. In the gnu directory, **gunzip** and **tar xf** the files that were downloaded from the ftp site.

5. Type **cd gnat-3.05-src-update**. Depending on the current status of GNAT, there may be a source code patch or two in here that are needed. In this research, a file called *sparc.dif* was needed. To apply this patch type:

cd ../gnat-3.05-src/src/ada and then
patch -p0 <../../../gnat-3.05-src-update/sparc.dif

Note: "Patch" is a program that takes as input a UNIX "diff" listing between two files (or many files within a directory). Patch can then change a file to make it look like the file with which the diff listing was made. Type **man patch** or **man diff** for more details. It is important to note that the UNIX version of Patch does not work on Solaris machines. If UNIX Patch is being used, the user will need to log into a workstation with SunOS just long enough to apply patches, before going back to Solaris to do configuring and compilation. The GNU version of patch should work as advertised on both SunOS and Solaris.

6. After applying the patch, look at the screen output to make sure all "hunks" were successfully applied. If there were any problems, further study of the patch file will be necessary. It is possible to make source code changes by hand if the patch utility fails to work on any particular hunks.

7. Type **cd ../../../gcc-2.7.2**

8. Type **patch -p0 <../gnat-3.05-src/src/gcc-272.dif**
In this step, the C compiler source code is modified to allow Ada to be compiled as well. Again, it may be necessary to log onto an SunOS machine to perform this step if the UNIX version of patch is being used. Check to make sure all "hunks" were successfully applied.

9. Type **cp -r ../gnat-3.05-src/src/ada .**
This will copy all GNAT source code into the GCC directory.

10. Type **cd ada**

11. Type **touch treeprs.ads a-[es]info.h nmake.ad[bs]**

12. Type **cd ..**

13. Type **configure --prefix= ~username/gnu/native-install**
Note: The entire path name must be typed (i.e. do not use ~username nomenclature).

14. Type **gmake CC=gcc CFLAGS="-O2" LANGUAGES="c ada"**

15. Type **gmake CC=gcc CFLAGS="-O2" LANGUAGES="c ada" bootstrap**

16. Type **gmake CC=gcc CFLAGS="-O2" gnatlib_and_tools**

17. Type **gmake install LANGUAGES="c ada"**

18. At this point, new GCC and gnatmake executables compiled from source code are available. From now on, this version of the compiler will be used as long as the GNAT environment variables are reset. To do this, open the .cshrc file and find the GNAT environment variables pointing to the binary installation we did earlier (section A). Change these variables to point to the new installation and type **source .cshrc** at the command line prompt.

B.2 Cross Tools

Once all native tools are installed on the host platform, the Ada cross-compiler can be built. In order to do this, however, several other cross utilities must first be installed, including the C compiler and embedded libraries. For this research, the directories used to build the native tools (*gcc-2.7.2*, and *binutils-2.7*) were chosen to be

removed rather than reused to build the cross tools. Supposedly this can be avoided by entering the directories and typing **gmake distclean**.

B.2.1 Install Cross-Binutils

1. In the *gnu* directory, create a new directory called *cross-install*. This is where the rest of the cross-GNU tools will be installed.

2. In the *gnu* directory, remove the used binutils-2.7 directory. Do this by typing **rm -rf binutils-2.7**

3. Untar a clean directory by typing **tar xf binutils-2.7.tar**

4. Type **cd binutils-2.7**

5. Type **configure --target=m68k-coff --prefix= ~username/gnu/cross-install**
   Note: The entire path name must be typed (i.e. do not use ~username nomenclature).

6. Type **gmake all** to build the cross version of GNU binutils

7. Type **gmake install** to install the cross version of GNU binutils in *~username/gnu/cross-install.*

8. At this point, native versions of the GNU compilation utilities will be available for use. Add *set path = (~username/gnu/cross-install/bin $path)* to the end of the .cshrc file.

9. Type **source .cshrc** for changes to take effect.

B.2.2 Install Cross-GCC

1. In the *gnu* directory, remove the used gcc-2.7.2 directory. Do this by typing **rm -rf gcc-2.7.2**

2. Untar a clean directory by typing **tar xf gcc-2.7.2.tar**

3. A special patch is needed to configure GCC as a cross-compiler. This patch can be downloaded from ftp.cygnus.com:/pub/embedded/crossgcc and is called *crossgcc-gcc-2.7.2.patch*.

4. Type **cd gcc-2.7.2**

5. Apply the crossgcc patch by typing **patch -p1 <../crossgcc-gcc-2.7.2.patch** . Remember, the patch utility may not work on Solaris machines. Logging in to a SunOS machine may be necessary to do this step.

6. Type **configure --target=m68k-coff --prefix= ~username/gnu/cross-install**
Note: The entire path name must be typed (i.e. do not use ~username nomenclature).

7. Type **gmake CC=gcc all** to build the cross version of GCC.

8. Type **gmake install** to install the cross version of GCC in *~username/gnu/cross-install.*

9. At this point, the GNU C cross-compiler will be built and installed. However, it will not be usable until the embedded C libraries are installed (next section).

B.2.3 Install Cygnus Newlib

1. Get the latest version of Newlib from the Cygnus ftp site (ftp.cygnus.com). *Newlib-1.7.0.tar.gz* was used for this research.

2. In the gnu directory, **gunzip** and **tar xf** the file that was downloaded from the ftp site.

3. Type **cd newlib-1.7.0**

4. Type **configure --target=m68k-coff --prefix= ~username/gnu/cross-install**
Note: The entire path name must be typed (i.e. do not use ~username nomenclature).

5. At this time, there are important files missing from the *m68k-coff* directory. The next four steps alleviate this problem.

6. Type **cd m68k-coff**

7. Type **ln ../config.sub**

8. Type **ln ../install.sh**

9. Type **cd ..**

10. Type **gmake all** to build Newlib

11. Type **gmake install** to install Newlib in *~username/gnu/cross-install.*

12. At this point, all the necessary tools are available to build cross-GNAT.

B.2.4 Install GNAT Cross-Compiler

1. In the gnu directory, remove the used gcc-2.7.2 directory. Do this by typing **rm -rf gcc-2.7.2**

2. Untar a clean directory by typing **tar xf gcc-2.7.2.tar**

3. Type **cd gcc-2.7.2**

4. Reapply the crossgcc patch by typing **patch -p1 <../crossgcc-gcc-2.7.2.patch** . Remember, the patch utility may not work on a Solaris machine. If the UNIX version of patch is being used, it may be necessary to login to a machine with SunOS to do any patch applications.

5. Apply the patch needed to compile GNAT by typing **patch -p0 <../gnat-3.05-src/src/gcc-272.dif**

6. Assuming the *sparc.dif* patch was applied earlier, when the native version of GNAT was built, type **cp -r ../gnat-3.05-src/src/ada .** , to copy all GNAT source code into the GCC directory.

7. Mr. Joel Sherrill at On-Line Application Research Corporation has developed a patch that is also needed to setup GNAT as a cross-compiler. This patch has not yet been released to the public, but is available from Mr. Sherrill via electronic mail (joel@merlin.gcs.redstone.army.mil). To apply this patch, assuming it has been saved as *joel.patch* in the *gnu* directory, type **patch -p0 <../joel.patch**. Make sure to check to see that all "hunks" were successfully applied. For this research, several hunks failed without explanation, and the source code changes had to be made manually.

8. In the *gcc-2.7.2* directory, edit *Makefile.in*, and find the following line:

install-common: native install-dir $(EXTRA_PARTS) lang.install-common

Remove the word "native" from this line, and save *Makefile.in*. It will be necessary to override the file's write protection to perform this step.

9. Type **cd ada**

10. Type **touch treeprs.ads a-[es]info.h nmake.ad[bs]**

11. Type **cd ..**

12. Type **configure --target=m68k-coff --prefix= ~username/gnu/cross-install**
    Note: The entire path name must be typed (i.e. do not use ~username nomenclature).

13. Type **gmake CC=gcc CFLAGS="-O2" LANGUAGES="c ada"**

14. Type **gmake CC=gcc CFLAGS="-O2" LANGUAGES="c ada" cross-gnattools**

15. Type **gmake CC=gcc CFLAGS="-O2" LANGUAGES="c ada" gnatlib**

16. Type **gmake CC=gcc CFLAGS="-O2" LANGUAGES="c ada" install**

17. At this point, the GNAT Ada cross-compiler has been compiled and installed.

Appendix C

Setting Up RTEMS for Compilation

The following steps must be performed in order to configure RTEMS for compilation. The target board, for testing purposes, was the Motorola MVME136 using the 68020 processor. This is not the same board that is found in the AFIT Robotics Lab. A general description for what must be done to get RTEMS running on the AFIT board can be found in Chapter 5, Recommendations for Further Research.

C.1 Steps for Compiling RTEMS

1. Update the files *ada/Modules/rtems/oar-mvme136* and *Template*. Here is what both files should look like:

filename: ***oar-mvme136***

```
#%Module
# oar-mvme136,v 1.9 1995/08/15 19:42:29 joel Exp
# Personal module file for RTEMS
# Must setenv:
#  RTEMS_BSP
#  RTEMS_ROOT
#  RTEMS_GNUTOOLS
#  RTEMS_CUSTOM
# before sourcing the Template file
setenv RTEMS_VENDOR    gnu
setenv RTEMS_BSP       mvme136
setenv RTEMS_ROOT      /home/hawkeye12/96d/klemley/Thesis/rtems-3.2.1
setenv RTEMS_GNUTOOLS  /home/hawkeye12/96d/klemley/Thesis/gnu/cross-install
setenv RTEMS_HOST      Solaris-2.3

# If using CYGNUS libc or GNU libc, then must set RTEMS_LIBC_DIR
setenv RTEMS_LIBC_DIR  /home/hawkeye12/96d/klemley/Thesis/gnu/cross-install/m68k-coff
source /home/hawkeye12/96d/klemley/Thesis/rtems-3.2.1/ada/Modules/rtems/Template
#end oar-mvme136
```

filename: ***Template***

```
#%Module
#
#  Template,v 1.2 1995/05/11 19:21:17 joel Exp
#
# Base module for RTEMS
# Should be sourced from your personal module; ref ./tbennett
#

setenv        RTEMS_CUSTOM   /home/hawkeye12/96d/klemley/Thesis/rtems-
3.2.1/ada/make/gnu/custom/mvme136.cfg

#append-path    MANPATH       /home/hawkeye12/96d/klemley/Thesis/gnu/cross-install/man

# For convenience only; makefiles try *never* to use $PATH
#append-path PATH       /home/hawkeye12/96d/klemley/Thesis/gnu/cross-install/bin

setenv r              /home/hawkeye12/96d/klemley/Thesis/rtems-3.2.1/ada
setenv rs             /home/hawkeye12/96d/klemley/Thesis/rtems-3.2.1/ada/src

# end Template
```

2. Change the following line in /ada/make/gnu/os/Solaris-2.3.cfg:

   From: KSH= /usr/bin/bash
   To: KSH= /usr/bin/ksh

3. Change the following line in /ada/make/gnu/os/Solaris-2.3.cfg:

   From: M4=/netapps/${APPLICATION_SERVER}/usr2/gnu/bin/gm4
   To: M4= /usr/local/bin/m4

4. In the gnu tools directory, perform the following steps:

   cd ~username/gnu/cross-install/lib/gcc-lib/m68k-coff/2.7.2/adalib
   m68k-coff-ar crv librest.a *.o

5. Once the above steps are completed, the following steps can be used at any time to compile RTEMS:

   cd ~username/rtems-3.2.1/ada/Modules/rtems
   source oar-mvme136
   cd $r
   gmake install

Appendix D

RTEMS Source Code Changes for Compilation Using GNAT 3.05

The following source code and makefile changes were implemented to make RTEMS 3.2.1 compilable under GNAT 3.05:

1. The following function was added to /ada/src/exec/cpu/gnu/m68k/cpu_data.ads:

```
function ARE_PRIORITY_BIT_MAPS_EQUAL (
  LEFT  : in    CPU_DATA.PRIORITY_BIT_MAP_CONTROL;
  RIGHT : in    CPU_DATA.PRIORITY_BIT_MAP_CONTROL)
   return BOOLEAN
renames CPU_DATA."=";
```

2. The following line was added to /ada/src/exec/rtems/types.adb:

```
use SYSTEM;
```

3. The following line was added to /ada/src/exec/rtems/user_extensions.adb:

```
use CONFIGURATION;
```

4. The keyword "aliased" was added to the following type in /ada/src/exec/rtems/chain.ads:

```
type CONTROL_ARRAY is array ( TYPES.UNSIGNED32
   range 0 .. 65535 ) of aliased CHAIN.CONTROL;
```

5. The following line was added to /ada/src/exec/rtems/io.adb:

```
use CONFIGURATION;
```

6. The following line was added to /ada/src/exec/rtems/rtems_tasks.adb:

```
use THREAD;
```

7. The following line was added to /ada/src/exec/rtems/internal_threads.adb:

```
use CPU;
```

8. The following line was added to /ada/src/exec/rtems/internal_threads.ads:

```
with UNCHECKED_CONVERSION;
```

9. The following line was added to /ada/src/exec/rtems/mpci.adb:

```
use MP_PACKET;
```

10. The following line was added to /ada/src/exec/rtems/multiprocessing.adb:

```
with INTERFACES; use INTERFACES;
```

11. The following line was added to /ada/src/exec/rtems/initialization.adb:

```
use CPU;
```

12. The key word "all" was changed to "constant" in the following lines of /ada/src/exec/rtems/configuration.ads:

```
type EXTENSION_TABLE_POINTER is access constant
      CONFIGURATION.EXTENSION_TABLE;

type MULTIPROCESSING_TABLE_POINTER is
  access constant CONFIGURATION.MULTIPROCESSING_TABLE;

type MPCI_TABLE_POINTER is access constant
      CONFIGURATION.MPCI_TABLE;
```

13. The following line was added to /ada/src/lib/libbsp/gnu/m68k/mvme136/clock_driver.adb:

```
use System.Storage_Elements;
```

14. The following line was added to /ada/src/lib/libbsp/gnu/m68k/mvme136/timer_driver.adb:

```
use System.Storage_Elements;
```

15. The following lines were added to /ada/src/lib/libbsp/gnu/m68k/mvme136/bsp.adb:

```
with System.Storage_Elements;
use System.Storage_Elements;
```

16. The following line was added to /ada/src/lib/libbsp/gnu/m68k/mvme136/interrupt_time_test_support.adb:

```
use System.Storage_Elements;
```

17. The following line was added to /ada/src/lib/libbsp/gnu/m68k/mvme136/shm-support.adb:

```
use System.Storage_Elements;
```

18. In the /ada/src/tests/sptests directory, there are the following 24 subdirectories: sp01-sp09, sp11-sp17, sp19-sp25, spsize. Each of these directories contain a file called sptest.ads. The following lines were added to each of these files:

```
with INTERFACES;
use INTERFACES;
```

Each of these directories also contain a file called main.adb. The keyword ACCESS was changed to UNCHECKED_ACCESS in each of the following two lines:

```
SPTEST.INITIALIZATION_TASKS'UNCHECKED_ACCESS,
SPTEST.DEVICE_DRIVERS'UNCHECKED_ACCESS,
```

19. The following lines were added to /ada/src/tests/sptests/sp14/sptest.adb:

```
with SYSTEM;
use SYSTEM;
```

20. The following changes were made in the /ada/src/tests/sptests/spfatal directory:

    a. Change ACCESS in line 208 of sptest.adb to UNCHECKED_ACCESS.

    b. Changed ACCESS in line 34 of main.adb to UNCHECKED_ACCESS.

    c. The following lines were added to sptest.ads:

```
with INTERFACES;
```

use INTERFACES;

21. In the /ada/src/tests/tmtests directory, there are the following 31 subdirectories: tm01-tm29, tmck, tmoverhd. Each of these directories contain a file called tmtest.ads. The following lines were added to each of these files:

```
with INTERFACES;
use INTERFACES;
```

Each of these directories also contain a file called main.adb. The keyword ACCESS was changed to UNCHECKED_ACCESS in following line (except in tm20):

```
TMTEST.INITIALIZATION_TASKS'UNCHECKED_ACCESS,
```

22. In /ada/src/tests/tmtests/tm20/main.adb, the keyword ACCESS was changed to UNCHECKED_ACCESS in the following two lines:

```
TMTEST.INITIALIZATION_TASKS'UNCHECKED_ACCESS,
TMTEST.DEVICE_DRIVERS'UNCHECKED_ACCESS,
```

23. Changed ACCESS in line 167 of /ada/src/tests/tmtests/ tmoverhd/main.adb to UNCHECKED_ACCESS.

24. In the /ada/src/tests/mptests directory, there are the following 14 subdirectories: mp01-mp14. Each of these directories contain a file called mptest.ads. The following lines were added to each of these files:

```
with INTERFACES;
use INTERFACES;
```

Each of these directories also contain a file called main.adb. The keyword ACCESS was changed to UNCHECKED_ACCESS in each of the following two lines:

```
MPTEST.INITIALIZATION_TASKS'UNCHECKED_ACCESS,
MPTEST.DEVICE_DRIVERS'UNCHECKED_ACCESS,
```

25. In the /ada/src/tests/samples directory, there are the following 4 subdirectories: hello, ticker, base_sp, base_mp. Each of these directories contain a file called sptest.ads. The following lines were added to each of these files:

```
with INTERFACES;
use INTERFACES;
```

Each of these directories also contain a file called main.adb. The keyword ACCESS was changed to UNCHECKED_ACCESS in each of the following two lines:

MPTEST.INITIALIZATION_TASKS'UNCHECKED_ACCESS,
MPTEST.DEVICE_DRIVERS'UNCHECKED_ACCESS,

26. Make the following changes to the file /ada/make/gnu/compilers/gcc-mvme136.cfg:

a. Add the -x flag to the gnatbind command.

b. Ensure that the line with the reference to the file generic.o is commented out.

c. Ensure the line with the reference to the file mvme135.o is commented out.

d. Changed the line "$(GCC_LIB_HOME)/libgnat.a \" to "$(GCC_LIB_HOME)/adalib/libgnat.a \".

e. The following line should appear three times in the LINK_FILES list:

$(PROJECT_RELEASE)/lib/libbsp$(LIBSUFFIX_VA) \

The first occurrence is after the line which reads "$(LD_LIBS) \"
The second occurrence is after the first line reading "$(LIBC_LOW) $(LIBC_LIBC) . . ."
The third occurrence is after the line which reads "$(GCC_LIB_HOME)/adalib/librest.a \"

27. Changed "2.6.3" to "2.7.2" in two places within /ada/make/gnu/custom/mvme136.cfg.

28. Moved the file glue.c into /ada/src/lib/libbsp/gnu/m68k/mvme136/glue.c. Glue.c is missing from the RTEMS 3.2.1 directory structure entirely. The C_PIECES section of the Makefile in that directory must also be updated to include "glue". Here is the complete source code for glue.c:

```
/*
 * glue.c -- all the code to make GCC and the libraries run on
 *         a bare target board. These should work with any
 *         target if inbyte() and outbyte() exist.
 */

#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <errno.h>
#include <_ansi.h>

#ifndef NULL
# define NULL 0
#endif


/* gnat runtime references these for now  -- Capt Lemley */

char *getcwd( char *_buf, size_t _size) { return 0; }
int fork() { return 0; }
int execv(const char *_path, char * const _argv[] ) { return 0; }
int wait() { return 0; }


/*
 * support setting the errno array if it's there
 */
#ifdef errno
# define SET_ERRNO(x)       errno = x
#else
# define SET_ERRNO(x)
#endif

#ifdef __NO_UNDERSCORE__
# define _end    end
# define _exit       exit
#endif


extern int  _EXFUN (outbyte, (char x));
extern char _DEFUN_VOID (inbyte);
extern int  _DEFUN_VOID (havebyte);

/* just in case, most boards have at least some memory */
#ifndef RAMSIZE
# define RAMSIZE            (caddr_t)0x100000
#endif

/*
 * print -- do a raw print of a string
 */
int
_DEFUN (print, (ptr),
char *ptr)
{
  while (*ptr) {
    outbyte (*ptr++);
  }
/* Line added by Capt Kendall Lemley , 22 Oct
*/
  return 0;
}

/*
 * read  -- read bytes from the serial port. Ignore fd, since
 *          we only have stdin.
 */
int
_DEFUN (read, (fd, buf, nbytes),
      int fd _AND
      char *buf _AND
      int nbytes)
```

```
{
  int i = 0;

  for (i = 0; i < nbytes; i++) {
    *(buf + i) = inbyte();
    if ((*(buf + i) == '\n') || (*(buf + i) == '\r')) {
      (*(buf + i + 1)) = 0;
      break;
    }
  }
  return (i);
}

/*
 * write -- write bytes to the serial port. Ignore fd, since
 *          stdout and stderr are the same. Since we have no filesystem,
 *          open will only return an error.
 */
int
_DEFUN (write, (fd, buf, nbytes),
        int fd _AND
        char *buf _AND
        int nbytes)
{
  int i;

  for (i = 0; i < nbytes; i++) {
    if (*(buf + i) == '\n') {
      outbyte ('\r');
    }
    outbyte (*(buf + i));
  }
  return (nbytes);
}

/*
 * open -- open a file descriptor. We don't have a filesystem, so
 *         we return an error.
 */
int
_DEFUN (open, (buf, flags, mode),
        const char *buf _AND
        int flags _AND
        int mode)
{
  SET_ERRNO (EIO);
  return (-1);
}

/*
 * close -- close a file descriptor. We don't need
 *          to do anything, but pretend we did.
 */
int
_DEFUN (close ,(fd),
        int fd)
{
  return (0);
}

/*
 * sbrk -- changes heap size size. Get nbytes more
 *         RAM. We just increment a pointer in what's
 *         left of memory on the board.
 */
```

```
caddr_t
_DEFUN (sbrk, (nbytes),
       int nbytes)
{
  static caddr_t heap_ptr = NULL;
  caddr_t        base;
/* extern int _end; added by Capt Kendall Lemley, 22 Oct 96
*/
  extern int _end;

  if (heap_ptr == NULL) {
    heap_ptr = (caddr_t)&_end;
  }

  if ((RAMSIZE - heap_ptr - nbytes) >= 0) {
    base = heap_ptr;
    heap_ptr += nbytes;
    return (base);
  } else {
    SET_ERRNO (ENOMEM);
    return ((caddr_t)-1);
  }
}

/*
 * isatty -- returns 1 if connected to a terminal device,
 *           returns 0 if not. Since we're hooked up to a
 *           serial port, we'll say yes _AND return a 1.
 */
int
_DEFUN (isatty, (fd),
       int fd)
{
  return (1);
}

/*
 * lseek -- move read/write pointer. Since a serial port
 *          is non-seekable, we return an error.
 */
off_t
_DEFUN (lseek, (fd,  offset, whence),
       int fd _AND
       off_t offset _AND
       int whence)
{
  SET_ERRNO (ESPIPE);
  return ((off_t)-1);
}

/*
 * fstat -- get status of a file. Since we have no file
 *          system, we just return an error.
 */
int
_DEFUN (fstat, (fd, buf),
       int fd _AND
       struct stat *buf)
{
  SET_ERRNO (EIO);
  return (-1);
}

/*
 * getpid -- only one process, so just return 1.
```

```
 */
#define __MYPID 1
int
_DEFUN (getpid, (),
        )
{
 return __MYPID;
}

/*
 * kill -- go out via exit...
 */
int
_DEFUN (kill, (pid, sig),
        int pid _AND
        int sig)
{

/* Added by Capt Kendall Lemley, 22 Oct 96
*/
int _exit();

 if(pid == __MYPID)
   _exit(sig);
 return 0;
}

/*
 * stat -- get status of a file. Since we have no file
 *         system, we just return an error.
 */
int
_DEFUN (stat, (path, buf),
       const char *path _AND
       struct stat *buf)
{
 SET_ERRNO (EIO);
 return (-1);
}

/*
 * unlink -- since we have no file system,
 *           we just return an error.
 */
int
_DEFUN (unlink, (path),
       char * path)
{
 SET_ERRNO (EIO);
 return (-1);
}

/*
 * putnum -- print a 32 bit number in hex
 */
int
_DEFUN (putnum, (num),
     unsigned int num)
{
 char  buf[9];
 int   cnt;
 char  *ptr;
 int   digit;

 ptr = buf;
```

```
  for (cnt = 7 ; cnt >= 0 ; cnt--) {
    digit = (num >> (cnt * 4)) & 0xf;

    if (digit <= 9)
      *ptr++ = (char) ('0' + digit);
    else
      *ptr++ = (char) ('a' - 10 + digit);
  }

  *ptr = (char) 0;
  print (buf);
  return 0;
}
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1996	Master's Thesis

4. TITLE AND SUBTITLE
IMPLEMENTATION OF AN ADA95 CROSS-COMPILER FOR THE REAL-TIME EXECUTIVE FOR MILITARY SYSTEMS (RTEMS)

5. FUNDING NUMBERS

6. AUTHOR(S)
Kendall M. Lemley, Jr., Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
250 P Street
Wright-Patterson AFB, Ohio 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/96D-16

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
Mr. Gary Shupe
DISA/CFSW/JEXSV
5600 Columbia Pike
Falls Church, Virginia 22041

AFMC Robotics Center of Excellence
Capt. Thomas Deeter
SA/ALC/TIEST Bldg 183
450 Quentin Roosevelt Rd.
Kelly AFB, Texas 78241

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis represents a continuation of the assessment of the Unified Telerobotics Architecture Project (UTAP), a proposed Air Force standard. This architecture was developed by the NASA Jet Propulsion Laboratory and the National Institute of Standards under contract to the Air Force Materiel Command Robotics and Automation Center of Excellence at Kelly AFB, Texas.

Due to operating system constraints, the only UTAP implementation to date has required a separate software interface layer, adding complexity and overhead to the overall system, while reducing portability. This thesis proposes a long term effort to design and implement UTAP-compliant application software devoid of this interface layer. Because the Ada programming language offers increased portability, and other software engineering benefits, emphasis is placed on developing a run-time infrastructure that will allow UTAP applications to be written in Ada. The first several steps of building this infrastructure is performed, including implementation of an Ada cross-compiler and real-time operating system.

Further UTAP research is recommended. The run-time infrastructure should be completed and UTAP application software developed using the Ada95 tasking model. Recommendations for UTAP specification improvements are also made.

14. SUBJECT TERMS
Robotics; Telerobotics; UTAP; Unified Telerobotics Architecture Project; Ada; Ada95; GNU Ada Translator; GNAT; Cross-Compiler; Real-Time Executive for Military Systems; RTEMS; Portability; Software Engineering

15. NUMBER OF PAGES
91

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL