UNCLASSIFIED

NRL Report 8346

Architecture Research Facility: An Experiment in Software Engineering

HONEY S. ELOVITZ

Information Systems Staff
Communication Sciences Division

December 31, 1979

NAVAL RESEARCH LABORATORY
Washington, D.C.

Approved for public release; distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
NRL Report 8346

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
ARCHITECTURE RESEARCH FACILITY:
AN EXPERIMENT IN SOFTWARE ENGINEERING

5. TYPE OF REPORT & PERIOD COVERED
Final report on one phase of a continuing NRL Problem.

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Honey S. Elovitz

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Department of the Navy
Naval Research Laboratory
Washington, DC 20375

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
NRL Problem B02-52
61153N, RR014-09-41

11. CONTROLLING OFFICE NAME AND ADDRESS

12. REPORT DATE
December 31, 1979

13. NUMBER OF PAGES
34

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Architecture Research Facility
Computer programming
Information-hiding modules
Modularization
Program design
Software
Software Engineering
Undesired events

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Software developers often complain that researchers in the field of software engineering propose new ideas without testing these ideas in practical applications. The Architecture Research Facility (ARF) was developed utilizing several software engineering techniques in order to discover their usefulness in actual software system developments. Such techniques as the complete design and documentation of the individual components and interfaces prior to coding; design reviews; code specification in pseudolanguage; code reading prior to testing; information-hiding modules; run-time error-checking mechanisms; strong-typing; and the use of support software tools are discussed.

(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)
We describe the motivation for using the techniques as well as how these techniques were applied to ARF's development. Staff reactions to using these techniques were favorable, although at the time some frustration at the lack of apparent coding progress was felt.

Our results would prove useful to software developers planning to use new development techniques, since we highlight many of these techniques' strengths and weaknesses.

UNCLASSIFIED

CONTENTS

I. INTRODUCTION 1

II. SOFTWARE ENGINEERING TECHNIQUES EMPLOYED 1

III. THE ARCHITECTURE RESEARCH FACILITY 2

IV. COMPLETE DESIGN DOCUMENTATION AND REVIEW 4

V. CODING SPECIFICATION 6

VI. CODING 7

VII. CODE READING 8

VIII. INFORMATION-HIDING MODULES 8

IX. UNDESIRED EVENTS DURING RUNTIME 9

X. ABSTRACT DATA TYPES AND STRONG TYPING 10

XI. SUPPORT SOFTWARE 12

- Preprocessor 12
- Interactive Testing Package 13
- Other Support Tools 14
- Unavailable Support Tools 14

XII. TESTING AND OPERATIONAL EXPERIENCES 15

XIII. SCHEDULING 16

XIV. CONCLUSION 17

ACKNOWLEDGMENTS 18

REFERENCES 18

APPENDIX A — Samples of Code Specifications 20

APPENDIX B — The ARF Coding Standards 28

UNCLASSIFIED

ARCHITECTURE RESEARCH FACILITY: AN EXPERIMENT IN SOFTWARE ENGINEERING

I. INTRODUCTION

Software developers often complain that researchers in the field of software engineering propose new ideas without testing these ideas in practical applications. The Architecture Research Facility (ARF) was developed utilizing several software engineering techniques in order to discover their usefulness in actual software system developments. This paper discusses our experiences. We describe the motivation for using the various techniques, how these techniques were applied to ARF's development, staff reactions to using these techniques, and an evaluation of the usefulness of such software engineering techniques.

Our results would prove useful to software developers planning to use these new development techniques since we highlight many of these techniques' strengths and weaknesses.

The ARF is a general-purpose simulator for computer architectures. The formal description language, the Instruction Set Processor (ISP) [1], is translated by a compiler into a sequence of primitive instructions called register transfer modules (RTMs). The ARF interprets the RTM instructions, thereby simulating the target machine whose architecture was originally described in ISP.

ARF's primary purpose was to provide software support to the Computer Family Architecture (CFA) project [2]. At the same time, the ARF development was used to gather information about several software engineering techniques. These techniques were evaluated in the context of software projects of ARF's size.

ARF's project personnel consisted of five full-time technical participants and two secretaries. As development progressed, this staff was reduced to three and then to two full-time technical participants. Of the five technical participants, three were designers while two persons were hired to code from detailed specifications.

II. SOFTWARE ENGINEERING TECHNIQUES EMPLOYED

In addition to providing a support tool for selecting a standard computer architecture, the ARF development was used to test several design and implementation techniques discussed in the software engineering literature. The following techniques were used:

1. Design and documentation of the individual system components and interfaces prior to coding [3]. (See Section IV.)

*Manuscript submitted June 14, 1979.

2. Design reviews prior to producing code specifications [4]. (See Section IV.)

3. Code specification in a pseudolanguage prior to coding [3]. (See Section V.)

4. Code writing by a staff member other than the designer or specifier. (See Section VI.)

5. Code reading prior to testing by at least one design-level project member other than the programmer [5]. (See Section VII.)

6. Information-hiding modules [6]. (See Section VIII.)

7. Run-time error-checking mechanism designed into the system from the start [7]. (See Section IX.)

8. Run-time simulation of strong typing to aid in error detection [8,9]. (See Section X.)

9. A FORTRAN preprocessor and other support software [10]. (See Section XI.)

Several of the software development techniques listed above are more easily achieved when an appropriate programming language is used during system development. Unfortunately, it is not always possible to find programmers familiar with such languages. One requirement was that the ARF be developed on an in-house PDP-10 in a language that was transportable to a PDP-11. Although the PDP-10 supports many languages, our PDP-11 did not have compilers for all of them. We were also not aware of existing cross compilers for any of the languages. Consequently, our choice was limited to BLISS (the PDP-10 has a cross compiler) or FORTRAN (the PDP-11 has a compiler). BLISS [11] is a sophisticated systems programming language, possessing many useful capabilities for projects such as ARF. It is also a complex and difficult-to-understand language. FORTRAN is widely known and widely used. Since inexperienced programmers were to be hired to code from detailed coding specifications, FORTRAN was chosen to implement the ARF.

III. THE ARCHITECTURE RESEARCH FACILITY

We define the architecture of a computer as the information that a programmer needs to know in order to write all programs that will correctly run on the computer. For two computers to have the same architecture, they must offer a common set of programmer-accessible registers and a common instruction set; execution of an instruction must affect the registers in the same way in one computer as in the other. A family of computers having a common architecture, but widely varying abilities, is IBM's System/360 series.

A designer intending to use the ARF specifies an architecture in the Instruction Set Processor (ISP) language. This description is translated by an ISP compiler [12] into a description of a register structure called register transfer modules (RTM) (see Fig. 1), and a program for the RTM machine that allows it to implement the proposed architecture. The RTM program can be thought of as analogous to the firmware in a microprogrammed implementation of the proposed architecture. An RTM simulator program running on an existing computer then interprets the RTM instructions, thereby simulating the target machine (the machine whose architecture is described in ISP) (see Fig. 2). A program written for the target machine would be executed by the RTM instructions provided that these instructions were so designed. The ARF's command language has the ability to load a target machine's memory.

UNCLASSIFIED

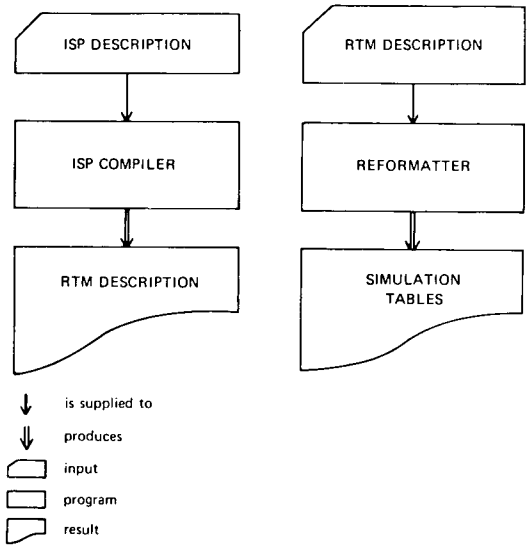

Fig. 1 — Presimulation processing

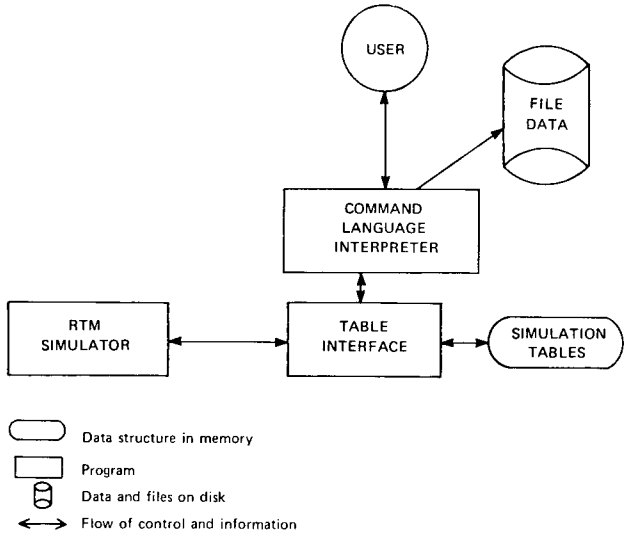

Fig. 2 — The run-time simulator

The ARF's main data structures are simulation tables consisting of the following items:

1. Primitive RTM instructions constituting a program for simulating the target machine

2. Descriptors for various types of data items such as variable-length registers, arrays, or labels

3. Symbolic names with pointers to facilitate communication between ARF and the user

4. The actual values of the data items described by item 2.

These tables are produced by the reformat program from the output of the ISP compiler.

Since the ARF user may want to interact with the simulation, the Command Language Interpreter (CLI) program accepts commands and requests for information typed by the user at a terminal. The CLI then uses the rest of the ARF to carry out the user's requests. A user may install a simulation from a file, start the simulation at any labeled point in the ISP description, set and clear breakpoint conditions that specify when a simulation will be interrupted automatically, restart a simulation, display or alter the contents of a register or elements of an array, etc.

IV. COMPLETE DESIGN DOCUMENTATION AND REVIEW

A detailed system design and extensive design documentation should exist prior to system implementation [3]. We decided that the ARF design would be thoroughly documented and reviewed prior to any coding effort. The ARF design proceeded in two stages, producing two types of design documents. First, the ARF was divided into modules and their interfaces were defined. Then, the design and implementation of each module could proceed at different rates. Detailed design documents were circulated to the ARF staff as well as to outside consultants for review. Design review meetings, consisting of all ARF technical personnel, were held to discuss and resolve discrepancies. Design documents were updated to reflect design changes, and the documents were then given a new version number and redistributed for additional criticism. The design and review process continued until all of the designers were satisfied with the design.

The documentation produced by the ARF project was detailed and voluminous (Table 1). Criticism of the ARF's design was solicited from persons working on all aspects of ARF as well as from knowledgeable persons outside the system development. The criticism obtained was quite useful and resulted in many valuable system changes made early in the design stages. The design and review process required multiple iterations; the time and effort involved in preparing and distributing documents were tremendous. A major bottleneck in the document revision and dissemination was secretarial support. The time to revise, retype, and release documents for subsequent reviews inhibited progress. This bottleneck was also present during the code specification stages but by that time several secretaries were trained to use on-line text editing; revision then proceeded more quickly, and designers were able to design rather than perform on-line editing.

Obtaining design criticism from project personnel in a timely fashion was difficult. Most personnel were assigned design tasks and had schedules to meet; design document review was

UNCLASSIFIED

Table 1 — Summary of the ARF Documentation

Document	Number of Versions	Number of Pages
Simulator		
Design	7	74
Code specification	8	102
CLI		
Preliminary design	2	34
Design	4	52
Code specification	7	145
Table interface		
Design	9	110
Code specification	6	116
Reformatter		
Design	1	6
Code specification	8	81
Breakpoint flag utilities		
Design & code specification	7	24
Register functions		
Design	4	25
Code specification	4	44
Paging utilities		
Code specification	4	31
Storage layouts	8	23
Miscellaneous utilities		
Character functions	3	6
String functions	4	11
Byte functions	2	14
Error-handling functions	2	8
Interrupt-handling functions	1	3
Preprocessor utility	1	5
Testing utility	1	19
ISP Compiler		48
Running ISP at NRL	—	29
Coding standards	3	7
User documentation		
ARF User's Manual	2	32
ISP Primer	4	26
57 Miscellaneous memos	—	347
A total of 83 documents	—	1422

usually given low priority. Finally, all participants exhibited a considerable amount of frustration at the apparent lack of progress during the early stages of the project. It seemed as though only voluminous amounts of paper were produced: design documents, revision of design documents, memos about proposed design changes, and coding specifications. No code or coding effort was evident.

Recommended solutions to most of the indicated problems can be offered.

1. Recognize the need for time to prepare documentation and plan for it in the project schedule.

2. Hire trained personnel early to provide support for documentation production.

3. Use on-line document preparation and editing to alleviate revision bottlenecks.

4. Provide time for project personnel to read thoroughly and criticize designs of all system components.

5. Recognize that design is an iterative process and plan the project schedule accordingly.

6. Be patient with the apparent lack of progress in producing code. Premature coding will result in the recoding of redesigned modules.

V. CODING SPECIFICATION

Design decisions should not be made by the programmer [13]; coding specifications should restrict the programmer's freedom. Unfortunately, there is no standard coding specification language.

Since clarity and ease of use are paramount in writing code specifications, we wanted to eliminate obstacles to writing clear specifications. Inflexible syntax is one of those obstacles; our designers wanted to concentrate on the unambiguous clear specification of the code rather than the syntax of the specification language. The language had to be algorithmic in nature, permitting specification to the level of detail necessary for the coder; this permits less experienced personnel to code while releasing persons with more technical backgrounds to design and specify. In addition we wanted the specifications readily understood; we did not intend to (and did not) distribute specification language manuals.

Since there was no coding specification language that fulfilled our needs, we decided to permit the designers, who were also our specifiers, to create their own specification languages. Three different languages resulted: one ALGOL-like [14], one BLISS-like [11], and one SIMPL-like [15]. Appendix A has a sample of each coding specification language used.

Many bugs were found while reading specifications and translating them to FORTRAN [16] that otherwise would not have been detected until testing. Perhaps, more importantly, the use of coding specification permitted us to multiply our effective manpower. We were able to use inexperienced personnel as coders while our more qualified personnel only designed and specified, keeping the coders busy coding.

UNCLASSIFIED

Based on the experiences and recommendations of our coders and specification readers, we decided that the SIMPL-like language was the best of the three specification languages. Although the ALGOL-like language was the most familiar, it lacked the ability to identify throughout the specification the location of variables (common block, parameter file, or local storage).

The BLISS-like language was difficult to read, with great possibility of ambiguous interpretations. It possessed *if-then-else* structure with no *end* indication. Several errors resulted when coders translated simple statements as part of the *else* block rather than as the next executable statement. Our coders as well as the specification readers complained that the BLISS-like specifications statements were closely crowded onto one page and indentions were not adequately used.

The SIMPL-like language possessed several desirable add-on syntactic features. This language had a mechanism for differentiating between local and global variables. Each variable's type could be clearly specified, along with the permitted type of access. Common blocks and predefined files with parameter values (parameter file) had to be identified, with a clear indication as to who was responsible for declaring, defining, and initializing the files. All global variable definitions contained an indication of the common block or file where the variable was defined. Such features could easily have been included in any of the specification languages.

We learned several things from our specification effort:

1. Use a uniform coding specification language. Use of three different coding specification languages confused our coders.

2. Use a language with flexible syntax that clearly specifies the location of variables.

3. Use a language that is understandable and does not need a manual to explain its semantics.

VI. CODING

Working from detailed coding specifications permits coders to implement a specified module without designing the module and without much communication with other project members. Our coders translated their assigned coding specifications into FORTRAN and syntactically checked their results; they did *not* run the programs. While coding was in progress, a detailed set of written project standards was followed. These standards outlined the format of all source code produced for the project. For example, every variable used in a program had to be declared. Variable declarations were required to appear in the same relative position in all programs. The use of the documented standard was enforced by project members assigned to reading the code (see Appendix B).

Coding from detailed specifications can be frustrating to coders who sometimes prefer expressing their creativity in code. We encouraged our coders to contribute to the project by finding potential bugs in the specification rather than attempting to make design decisions and change their code specifications. (One of our coders eventually became a designer.)

VII. CODE READING

To experiment with egoless programming [17], we adopted a policy that all code would be read by at least one staff member other than the coder.

After an ARF coder finished his assigned task, a different project member, preferably the designer or code specifier, read the code. The code reader checked to see that no translation errors, misunderstandings, or violations of the project's coding standards occurred. If the code reader found errors in the algorithm's specification, the original designer and specifier were consulted and the appropriate changes were made to the design, code specification, and code. The coder made code changes that reflected code specification changes only. The entire code reading process was repeated until no errors were detected by the code reader. The program was then ready for run-time testing.

Twenty-four errors were found during the code specification reading and code reading stages: 4 translation errors, 14 specification errors, 3 code errors, 2 typographic errors, and one violation of the coding standards. For more detailed information about the errors occurring during the ARF project see Ref. 16.

VIII. INFORMATION-HIDING MODULES

Parnas [6] has introduced the concept of information-hiding modules. To partition a software system according to the information-hiding principle, one isolates (hides) each major design decision from the rest of the system; each information-hiding module manages a facility or resource for the remainder of the system. A module usually consists of a data structure and the routines that access the data. Isolation of the data structure (resource) permits changes to the managed resource with minimum effect on other system modules.

The ARF's information-hiding decomposition resulted in four major modules, each managing a separate ARF facility (see Fig. 2): the CLI provides all user communication, the simulator executes the target machine instructions, the table interface module manages the tables containing all simulation data, and the register functions provide all operations required to implement a 64-bit ARF register data type.

ARF project personnel designed and coded each module, making very few assumptions of the other modules' internals; any assumptions made were embodied in a well-defined and documented interface. Consequently, independent work was possible and little communication was necessary after the interfaces were defined. The abstract interfaces [18] to a module was the only information necessary for the implementors of other modules to know. The ease with which the system changes can be shown by several examples below:

1. The ISP compiler translates the instructions of any machine being described into a simple primitive set of operations called RTMs. These RTM instructions are reformatted into the proper table format by the reformat program. Both the reformat program and the simulator access the RTM instructions in the table interface module. Thus, the format and location of the RTM instructions are hidden by the table interface module. The modifications to add another RTM instruction were limited and easily recognizable, so that, once made, no side effects resulted. The reformat program was modified to expect the new operation; a new routine to simulate the operation was added to the RTM simulator module; the table interface module was supplemented with the necessary new functions. The modifications took one day to implement including design, specification, review, coding, testing, and documentation updates.

UNCLASSIFIED

2. The field size of one of the RTM operation addresses was enlarged to accommodate additional address space. We relocated the address field into an unused portion of another word in the table. This change did not affect any of the other system modules; the table interface module was modified so that only the routine accessing the field was aware of the new location. The modification required less than half a day to implement. Other modifications such as rearranging the table structure, expanding fields, and adding entries were accomplished as easily.

IX. UNDESIRED EVENTS DURING RUNTIME

Even the best engineered system will contain system errors. In many situations, errors do not force a machine abort during execution, but cause incorrect results. We use the term *undesired events* (UEs) [7] to encompass run-time errors, incorrect program performance, undesired external stimuli, as well as hardware errors. Ignoring UEs at a project's beginning requires the later addition of error handling. Such an ad-hoc design addition for handling UEs can destroy a system's structure. To avoid major redesign after the fact, the ARF developers were concerned with making UE handling an integral part of the system.

The table interface module contains extensive error-checking code to prevent programs' manipulating incorrect data or performing undesired operations. Whenever an attempt to perform an erroneous action was made, the table interface module either returned to the caller with an appropriate return code set or called a separate UE-reporting routine, which printed debugging information and either continued or waited for a user response. Such information as the calling routine's name, the called routine's name, and the kind of undesired event was reported.

The ARF handled two kinds of undesired events: ARF errors and user errors. ARF errors were system bugs; about 50% of the table interface module code was devoted to detecting and reporting such errors. User errors were caused either by incorrect ISP programs or incorrect target machine programs and data. For example, a divide by zero in ISP would create a user error because of incorrect data.

Many ARF errors were handled by debugging code designed into the system with the idea that this code would be eliminated when ARF was correct and reliable. A preprocessor was implemented to permit any ARF source code line to be prefixed by a D, causing the preprocessor to convert such statements to FORTRAN comments. Non-error-checking ARF code was kept from depending on the evaluation of error-checking code. In this way, when error-checking code was eliminated, the remainder of the ARF code would still work properly. ARF code can be recompiled with most of the error computation omitted. A disadvantage of the error mechanism was the increased processing overhead; but the design permits the ARF code to be recompiled without the error handling — typically halving execution times.

The value of preplanning the error-detection mechanism is illustrated by the ease of reducing processing overhead in a debugged system as well as the fact that 14 errors were detected by the UE-handling mechanism [16].

X. ABSTRACT DATA TYPES AND STRONG TYPING

Abstract data types have been recommended as a means for improving system reliability and understandability [8,9,19,20]. Proper use of abstract data types can simplify the programmer's task; high-level data objects can be specified without concentrating on detailed, and perhaps machine-dependent, representations and operations.

In strongly typed languages, the data type of each variable must be explicitly declared. The mixing of data types without explicit and well-defined type-conversion rules is prohibited. These languages enforce a set of rules that define when one object can be bound to another object, such as in parameter passing. Such rules protect the programmer from misuse of data types. When an appropriate strongly typed language does not exist, type rules can be enforced at run time with a corresponding processing overhead.

The implementation of strong-typing features in the ARF development helped in system debugging. We also used abstract data types to enhance readability and changeability. The table interface module's responsibility was to provide access to and to manipulate the ARF tables for the entire ARF system. Table entries can easily be likened to abstract data types [9]. All access to the table entries was controlled by the table interface module; each entry's representation was concealed from all programs external to the module, making data representations inaccessible to the using programs (and unnecessary for the using programmer to know). We believe that concealing the data representation increased the system's reliability and changeability.

Because FORTRAN is not a strongly typed language, FORTRAN variables need not be explicitly declared and mixed-mode expressions are not prohibited. Since the ARF development used FORTRAN, we incorporated the most obvious strong-typing features by run-time checking and by coding standards. Such enforcement would normally be done at compile time when using a strongly typed language such as Pascal [21].

The ARF coding standards enforced explicit declaration of all variables. Since we could not automatically verify that all variables were declared explicitly, code readers were required to check for adherence to the coding standards. The standards required that each ARF routine contain a FORTRAN statement that declared all variables complex unless explicitly declared otherwise. The use of the FORTRAN "IMPLICIT COMPLEX (A—Z)" statement fulfills this requirement. Any variable not explicitly declared defaults to complex — a data type not used in the ARF development. Most attempts to use a complex variable as a subscript or DO loop index cause a compiler error.

The ARF design required four abstract data types:

LOG	true or false
INT	a 16-bit data word
REG	a 64-bit data word
STRING	a string of characters of a specifically declared length.

UNCLASSIFIED

These types were used in the coding specifications. In the case of LOG, INT and REG, the programmer was responsible for translating the type into the proper FORTRAN declaration: LOG into LOGICAL, INT into INTEGER, and REG into DOUBLE PRECISION.

The STRING data type was handled somewhat differently. ANSI FORTRAN [22] provides few character and string manipulation facilities. Usually, programs using string or character data are machine dependent, since the number of characters per word must be defined in a FORTRAN DATA or FORMAT statement. We wanted to avoid machine dependencies so that the system could be transported to a PDP-11. Consequently, the ARF programmers were not permitted to use a DATA statement to declare strings. A preprocessor command was implemented that would produce FORTRAN code for string definitions; the programmer declared the name, length, and value of the string that was needed. An ARF utility module was implemented that provided the functions necessary for manipulating the strings. In this way, an abstract data type, STRING, was implemented through the preprocessor definition and an information-hiding module that hid the representation of the string data type.

Not only were the programmers unaware of the string's internal data representation, but this string type could easily be transported to another machine without changing the ARF programs. A module for string manipulation was provided that performed all string accesses (e.g., substring, assignment, concatenation, appendage, etc.).

One ARF module defined a 64-bit REG data type and supplied all of the required type conversion routines and primitive operations such as addition, subtraction, multiplication, and division. FORTRAN primitives were used only for the INT types.

Some run-time verification of correct typing of passed parameters was accomplished via the ARF's abstract data type definition for table entries. Each table entry had a specified field for type identification set by the ARF module that formed the table. The table interface module was the only run-time module that had access to or knowledge of the identification field. Consequently, when a table interface subroutine was called with a table entry as a parameter, the table interface routine checked the parameter's identification field to verify that the proper table entry type was passed. Since each table entry type had a unique identification field value, an unexpected identification field value forced the ARF's UE-handling routine to be invoked.

In summary, the ARF development implemented strong-typing features in three ways:

1. All variables were required to be declared explicitly.

2. Four abstract data types were used in code specifications; one was defined at preprocess time.

3. Type conversion and manipulation routines were provided.

We were pleased with the results of using strong typing and data abstractions in the ARF development. These concepts helped in detecting approximately 50% of the errors in the table interface module [16]. Use of both data abstractions and strong-typing rules increased the readability of the ARF documentation as well as the ARF code.

XI. SUPPORT SOFTWARE

Support software usually becomes a major aspect of software system developments [10], especially if the software system will be used frequently or is expected to evolve to meet varying user requirements. Unfortunately, many software system developers do not plan their software support package prior to starting a project.

We recognized the need for support software but unfortunately did not have the time to anticipate this need prior to the start of the project. Consequently, several support tools were designed and programmed in parallel with the ARF development. Limited time and manpower precluded the development of many other desired tools.

This section discusses several tools that were developed for use during ARF's development. We also present some capabilities that were not implemented because of lack of time and personnel, although we believe that they would have been useful.

Preprocessor

The ARF was developed in ANSI FORTRAN [22]. Unfortunately, the current ANSI FORTRAN lacks such desirable features as compile-time inclusion of source libraries, compile-time parameters, and conditional compilation of code. Some FORTRAN implementations do not enforce these limitations, but since most extensions are not part of the ANSI standard, they cannot be used in software intended for use on more than one machine. The ARF runs on a PDP-10, which does support the above features; but, ARF was intended to be transported to the PDP-11, which does not support such extended FORTRAN capabilities. During the ARF's development, we recognized the need for a limited capability preprocessor that would provide some useful facilities lacking in ANSI FORTRAN.

Consequently, we designed and implemented a FORTRAN preprocessor; the FORTRAN code produced was ANSI standard. The preprocessor, written in FASBOL, a string manipulation language described in Ref. 23, was capable of running only on the PDP-10. We planned to transmit the FORTRAN source code to the PDP-11 via a telephone link. With the preprocessor, ARF programmers were able to use several nonstandard FORTRAN features that simplified their task and enhanced the readability of the FORTRAN code.

ANSI FORTRAN does not permit the definition of compile-time parameter constants, although some machine-dependent FORTRANs do permit such parameter definitions. Since there were several advantages in requiring our designers and programmers to use compile-time parameter definitions, we implemented this feature in our preprocessor. Compile-time parameter definitions permitted decision deferral of the parameter values such as table sizes. In this way, our designers determined the parameters' values rather than the programmers' "hard coding" the constants. Since our programmers were unaware of the actual parameter values, they were unable to make use of such constants in devious ways in their code. Compile-time parameters also permitted the use of mnemonic names in place of the constant's value.

The FASBOL preprocessor assisted in the implementation of a string data type. Although moving the ARF to a different machine requires the preprocessor to produce slightly different code for invocations of the string utility module, the ARF programs remain unchanged.

UNCLASSIFIED

Another feature implemented in the FORTRAN preprocessor is the ability to flag specific FORTRAN statements as conditionally compiled debugging statements. The ARF preprocessor permitted a programmer to flag any FORTRAN statements, including comments, as debugging statements. All debugging statements were translated to FORTRAN code by the preprocessor unless requested otherwise. The programmer could request that all debug statements appear as comments in the source listing. A reader could see immediately those statements that are not actually part of the ARF system code. This conditional compilation facility permits the debugging code to remain with no overhead even after the system is running, insuring easy reinsertion at a later time.

A desirable feature not implemented in the preprocessor was the ability to use more than six characters as a variable name. The usual FORTRAN six-character limitation hinders the production of readable code by restricting the invention of mnemonic variable names. The ARF preprocessor permitted ARF programmers to use more than six characters only for parameter names. The capability of using more than six characters for ARF variables was not implemented in the ARF preprocessor because of a shortage of time and personnel. Had the preprocessor implemented this capability the ARF code would have been more readable.

Many nonstandard FORTRAN compilers also provide the ability to specify a file to be included in-line in a program. This was recognized as being a useful feature for the ARF development, since the parameter definitions were built by persons other than the coder. The designer built a parameter definition file and specified in the coding specifications the name of the parameter file to be included in the code. An ARF coder used a preprocessor command at the appropriate point in the code, and the preprocessor automatically produced ANSI FORTRAN for the routine, using the specified parameter definition file. This required that the preprocessor also process the contents of the file being included in the code.

A logical extension of the parameter-file-include capability was to permit the specification of any FORTRAN source file to be included in-line. This provides a minimum-capability macro expansion facility. It was not possible to specify parameters to the expanded macro, and consequently the preprocessor was not as versatile as we would have liked. A full macro capability would have been extremely useful, since many of the FORTRAN subroutines and functions written to implement the information-hiding modules could have been implemented easily and more efficiently as macros. A macro facility in the source language is a great asset in implementing information-hiding modules. A macro facility was not provided in the preprocessor because of a lack of time and personnel to devote to an unbudgeted item.

Interactive Testing Package

Since the ARF is modularized according to the information-hiding principle, many of the modules consist of numerous subroutines or functions that retrieve or set values. The testing of so many different routines can be tedious, requiring many small driving programs whose only purposes are to invoke the proper FORTRAN routine with the specified parameters to report the results. Instead, the ARF project produced a general-purpose driver as a support tool. This driver was called the Table Access Routine Interactive Tester and was written in the intial stages of the ARF design; its original purpose was to assist in testing the table interface module. Any subroutine or function linked with the package can be invoked by specifying the routine along with the parameter values. The testing package then calls the routine and displays all the returned values.

The ARF UE reporting routine invoked the testing package when a UE occurred during debugging, providing the tester with interactive facilities to call the suspect ARF routine with various parameters to help discover the bug.

Without the support provided by this testing package, we would have required many individualized driver routines to test the table interface module and the RTM simulator, substantially increasing testing time and cost. Such a support tool can prove to be invaluable to any software development [24,25].

Other Support Tools

In addition to the preprocessor and the testing package, the ARF team developed a program that assigns unique identifiers to all ARF subprogram names. The ARF routines then had a mechanism for uniquely identifying themselves to the UE reporting routine. In this way, the UE reporting routine printed out diagnostics indicating the routine in which the error occurred.

Sometimes global definition changes, table size changes, and module modifications required large portions of the ARF to be recompiled. Since each module contained many subroutines, each separately compiled, the task to compile the system was tremendous. A program was written to preprocess and compile automatically all the routines in a specified module for the entire system. This program was a valuable support tool during system development and has remained useful throughout ARF maintenance.

Many PDP-10-supplied support tools such as text editors, linkers, listers, and the batch monitor system were used throughout ARF's development. The PDP-10 text editors were especially invaluable to our project since all documentation was prepared on-line, facilitating the numerous changes that occurred during the system's evolution.

Unavailable Support Tools

One of the most difficult development tasks is to prevent naming conflicts. A program that keeps track of all the names used in the system and insures that no name is used more than once as a subroutine or function name could easily reduce time and effort in project development. Name information must be compiled from the beginning of the project development or the task quickly gets out of hand. Another facility that would have been useful was a program to identify all the variables defined in each common block and to list the modules using each common block variable.

Software projects producing large amounts of documentation usually release multiple versions of design documents and system code. An automatic method for version numbering could permit project personnel to maintain up-to-date documentation with reduced technical effort diverted to administrative duties. An automatic means to cross reference and index all project-related documents assists in the understanding and reviewing of the system design as well as the prevention of naming conflicts.

The ARF support tools proved to be useful in the ARF's development, but more time prior to the beginning of the project should have been devoted to identifying the necessary tools and designing or acquiring the appropriate software from previous projects. Since initiating different procedures and using newly provided automated tools midway through the project would have been disruptive, we chose not to expand the capabilities of already usable tools.

UNCLASSIFIED

Thus, many attractive preprocessor enhancements were not provided and other support tool developments were not commenced after the ARF system development was under way.

XII. TESTING AND OPERATIONAL EXPERIENCES

The testing of a software system is influenced by the system's structure. The modularization of the ARF clearly defined the lines along which the testing of the system components proceeded. Each information-hiding module provided a clean, well-documented interface to the other system modules. Each ARF program was tested by a staff member other than the original programmer. Since the tester was usually unfamiliar with the code being tested, we hoped that he would be unlikely to make assumptions about the code that would lead him to design inaccurate or incomplete test data. The original programmer was responsible for changing code in which translation errors were found; under no circumstances was he permitted to alter designs or specifications. The original designer and specifier were responsible for adjusting design and specification documentation when an error or ambiguity was discovered.

The use of information-hiding modules permitted the modules to be demonstrated correct independently of each other. This reduced the combinatorial problem where each routine must be tested in conjunction with all of the other routines. Consequently, after the table interface module testing was completed, the testing of the RTM simulator and the CLI could proceed.

We intended to provide extensive test guidelines and data for the individual modules. Unfortunately, this aspect of the ARF development was sorely neglected. Each module, after being coded and read, was ready for testing. Ideally, the tester should have provided a test plan prior to initiating any testing of the module; but, at this phase of the project, we were short of manpower and time and were unable to expend the resources to provide a well-planned test procedure.

In spite of a lack of test planning, the individual module testing, system integration, and system testing proceeded quite smoothly. We attribute this to the use of information-hiding modules and the detailed design documentation and review processes. Few difficult-to-fix errors (taking more than one day to fix) were discovered, and integration took less than one month to accomplish. Several errors that surfaced during ARF's integration can be attributed to misunderstanding in the ISP compiler's interface.

Since the ARF system development was used as a software engineering experiment, an error-reporting procedure was developed. Throughout ARF's development and continuing during operation and maintenance, data were collected on the errors encountered. Errors found during specification review, coding, code reading, testing, and operation were reported on specially prepared error report forms. These forms contained information such as the error type, where the error occurred, how it was detected, how long it took to find the cause of the error, whether the cause was found, what the finder of the error thought was the cause, etc. All reports were returned to an individual not involved in the ARF development. The reports were reviewed and the reporter and the person responsible for the error were interviewed. All staff members were encouraged to report all errors as part of the experiment.

Since there is no convenient way of enforcing error reporting, we relied on both the staff's interest in software engineering and their belief that errors would not be held against them; error reports were not used to influence performance ratings. Initially, there was some confusion about what constituted a reportable error, but within six to eight weeks all errors, as far as we could tell, were being reported.

During the testing of the RTM simulator module, approximately ten errors were found [16]. Of these, nine were errors caused by incorrect coding specifications (several were duplicate errors found in routines doing similar tasks). The remaining error was caused by a FORTRAN call with the incorrect number of parameters. Most of these errors were easily found because of the previously discussed table interface module UE-handling.

In software systems consisting of many components (modules), the interfaces can be the most critical and vulnerable elements in the design [3,6]. When such a software system is developed solely in one installation, with exclusive control exercised by the project manager, the interfaces can be well defined, documented, and consistent. But, when a component of the system is developed elsewhere with the project manager exercising no control, interface difficulty can easily result.

The ARF was developed in-house except for one major component—the ISP compiler. The ISP compiler was developed at Carnegie-Mellon University (CMU) as a component of a computer-aided design system. At the time, ISP was the only well-developed hardware description language with a compiler.

We believe that the weakest link in the ARF's development was the interface between the ISP compiler and the ARF simulator. Before the ARF was developed, the ISP compiler was used in several other applications, so the ISP compiler's interface was well defined although not well documented. But, since we did not develop the ISP compiler, interface difficulties resulted. During system integration and testing, inconsistencies in the semantics and the interface were revealed. These difficulties did not create havoc with the ARF design, but they do seem to indicate that caution should be used when undertaking a project that uses an already produced software product that is not well documented.

XIII. SCHEDULING

Most software projects require schedules and milestones so that managers may judge the progress of the project. Although the ARF project enjoyed an unusual lack of pressure to meet a specific time schedule, we felt that to judge realistically the value of the software techniques employed, a schedule and set of milestones were necessary. We were able to have our schedule devised by staff members prior to starting the project. These persons were familiar with the project's goals and had previously participated in technical efforts.

With very little pressure to devise a schedule that would satisfy management's plans at the expense of realistic goals, the ARF staff was able to judge honestly the technical chores at hand and estimate the time involved for each. We were very conscious of the failure of most software projects to estimate time schedules accurately and hoped to avoid making similar mistakes. We attempted to be quite conservative in our estimates, but we had little previous experience with formulating schedules and milestones.

On reviewing our schedule, some persons claimed that we were too conservative. They were incorrect; like other software projects, our schedule slipped. But, unlike other software projects, we did not slip in the coding and the testing stages of development, only in the design stages. We spent more time in design than expected, although we had planned and scheduled large portions of development time for design and specification. Some of the delay was caused

UNCLASSIFIED

by the documentation support bottleneck, but the main cause was that the design stage just required more than the allotted time. The effort in design was not wasted; less time was required for integration and testing of the system than expected. ARF was operational about ten months after development started rather than the originally estimated seven months.

Several factors caused our schedule slippage and should be considered when devising a schedule.

1. Computer down-time. When designing and coding in parallel, computer down-time can be devoted to design. But, when the ARF project was using the computer, the design was complete and down-time was lost time.

2. Personnel sickness and vacation. A project's schedule must take into account sick and vacation time.

3. Delays in obtaining personnel. Plan to obtain all personnel prior to the project's start; otherwise, incorporate appropriate delays into the schedule to account for bringing new people up to speed.

These "trivial" factors are rarely considered an important aspect of a project's time schedule, but, as Brooks [13] points out, "Our techniques of estimating are poorly developed."

XIV. CONCLUSION

After the Architecture Research Facility became operational, it was maintained by one of the original staff members. Capabilities were added to expand the user interface. The software principles originally used in the ARF development were maintained; proposed changes were documented and coding specifications were written and reviewed prior to implementation. System modifications proceeded smoothly.

The ARF development error analysis has yielded interesting and informative results [16]. As for the influence of the software techniques employed, our conclusions are as follows:

- The utility of well-designed support tools should not be underestimated. The kind of support tools that are currently available as well as the support tools that will be required must be determined prior to the project's start. The project's schedule should be adjusted to take into account the support tools that must be developed in parallel with the project.

- Modularization along information-hiding lines is not only a useful principle but aids in documentation, understanding, testing, and integration of the system. The UE-handling permitted debugging to proceed smoothly for the entire system as well as the individual modules. These techniques permitted more manpower to be devoted to design and specification—the most important aspect of the system development. Detailed system documentation is a must and must be prepared before implementation starts; the precise specification of the system requires the emphasis in a project, not the coding.

ACKNOWLEDGMENTS

Many of the author's colleagues provided sound advice and support while the author learned to manage a software project. John Shore and David Weiss deserve special thanks for their help during that time. Kathryn Heninger, Kenneth Hayes, and Bruce Wald provided many helpful suggestions while reviewing this report. David Parnas must be recognized for both his guidance during the project and as the advocate of the many techniques so successfully used. Finally, the author must thank all the participants of the ARF project for their cooperation and support in applying the techniques and making this report possible: Jeffrey Entwistle, Gregory Lloyd, John McHugh, Alan Parker, John Shore, and Paul Strauss.

REFERENCES

1. C.G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.

2. *Computer*, Oct. 1977.

3. H.D. Mills, "Software Development," *Proc. 2nd Internatl. Conf. Software Engrg.*, Oct. 1976.

4. M.E. Fagan, "Design and Code Inspections and Process Control in the Development of Programs," IBM System Development Division, TR21.572, Kingston, N.Y., 1974.

5. T. Baker, "Chief Programmer Team Management or Production Programming," *IBM Syst. J.* **11** (No. 1) 56-73 (1972).

6. D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Commun. ACM* **15**, (No. 12), 1053-1058 (Dec. 1972).

7. D. Parnas and H. Wuerges, "Response to the Undesired Events in Software Systems," *Proc. 2nd Internatl. Conf. Software Engrg.*, San Francisco, Oct. 1976.

8. T. Linden, "The Use of Abstract Data Types to Simplify Program Modification," *Proc. Conf. Data*, Mar. 1976 (also in *SIGPLAN Notices* **8** (No. 2)).

9. B. Liskov, "Programming with Abstract Data Types," *SIGPLAN Notices* **9** (No. 4), Apr. 1974.

10. D.J. Reifer, "Automatic Aids for Reliable Software," *Proc. Internatl. Conf. Reliable Software*, Apr. 1975, pp. 131-142.

11. Digital Equipment Corporation, "BLISS-10 Programmer's Reference Manual," DEC-10-LBRMA-A-D, Feb. 1974.

12. M. Barbacci, "The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator," Department of Computer Science, Carnegie-Mellon University, Aug. 2, 1976.

UNCLASSIFIED

13. F.P. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Pa., 1975.

14. R. Bauman, M. Feliciano, F.L. Bauer, and K. Samuelson, *Introduction to ALGOL*, Prentice-Hall, Englewood Cliffs, N.J. 1964.

15. V.R. Basili and A.J. Turner, "SIMPL-T A Structured Programming Language," CN-14, University of Maryland Computer Science Center, Jan. 1974.

16. D.M. Weiss, "Evaluating Software Development by Error Analysis: The Data From the Architecture Research Facility," NRL Report 8268, Dec. 22, 1978.

17. G.M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.

18. D.L. Parnas, "The Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems," NRL Report 8047 (June 3, 1977).

19. D.L. Parnas, J. Shore, and D. Weiss, "Abstract Types Defined as Classes of Variables," *Proc. Conf. Data*, Mar. 1976 (also in *SIGPLAN Notices* **8** (No. 2)).

20. O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, Inc., London, 1972.

21. N. Wirth, "The Programming Language Pascal," *Acta Informatica* **1**, 35-63 (1971).

22. American National Standard FORTRAN, ANSI X3.9-1966, Mar. 1966.

23. Digital Equipment Corporation, "FASBOL II, a SNOBOL Compiler for the PDP-10," Maynard, Mass., Dec. 1972.

24. D.J. Panzl, "Test Procedures: A New Approach to Software Verification," *Proc. 2nd Internatl. Conf. Software Engrg.*, Oct. 1976, pp. 477—485.

25. E. Satterthwaite, "Debugging Tools for High Level Languages," *Software Practice and Experience* **2**, 197-217 (1972).

Appendix A
SAMPLES OF CODE SPECIFICATIONS

This appendix includes samples of three code specification formats: SIMPL-like, ALGOL-like, and BLISS-like.

UNCLASSIFIED

1. SIMPL-like Code Specifications

CODE SPECIFICATION

Routines to clear access counters

<ROUTINE>	[PHYSICAL FIELD]
>CLRDCT<	[READ COUNT]
>CLWTCT<	[WRITE COUNT]
>CLMOCT<	[MONITOR COUNT]

Routines to increment access counters

<ROUTINE>	[PHYSICAL FIELD]
>INRDCT<	[READ COUNT]
>INWTCT<	[WRITE COUNT]
>INMOCT<	[MONITOR COUNT]

PARAMETER FILES:

ACERRI.REQ	ERRINC Codes for calling >ERR<
ACNAM.REQ	parameter definitions for routine name codes used for calling >ERR<
ACATTP.REQ	ATRTAB entry type code
ARFSIZ.REQ	parameters for array size declarations and table entry size definition

COMMON BLOCK DEFINITION FILES:

ACSTOR.REQ	primary data structure for support of table access routines

EXTERNAL REFERENCES:

>ERR<	for error reporting
(Byte and Halfword routines)	as required to extract table entries

ALGORITHM

```
/* <ROUTINE> is the parameter containing the name of the
   routine being coded. */

!if !DEBUG
  !then /* code the following with 'D' in column 1 */

    if ATRBD = -1
      then

        call ERR(ERRCODE,<NOBIND> + <ERRCSZ> * <ROUTINE>)
        return

      end

    if ATRTP is not appropriate for the desired [PHYSICAL FIELD]
      then

        call ERR(ERRCODE,<ILLTYP> + <ERRCSZ> * <ROUTINE>)
        return

      end

  !end /* !DEBUG */

!if routine being coded clears a counter
  !then

[PHYSICAL FIELD] := 0

  !else /* routine must increment a counter*/

if CTSTAT{ACSTOR}
then /* Counting will be effective */

[PHYSICAL FIELD] := [PHYSICAL FIELD] + 1

end

!end
return
```

UNCLASSIFIED

2. ALGOL-like Code Specifications

>XATINF<

FUNCTION: Binds a scalar ATRTAB entry and retrieves the length (LEN), RBIT, and value of the entry before unbinding it.

COMMENTS AND DESCRIPTION: This function combines a set of operations that are frequently repeated within the RTM sequencer.

SPECIAL MACHINE OR OTHER DEPENDENCIES: None

EXTERNAL REFERENCES:

>ATRBND<
>GATRLN<
>GRDBIT<
>GSCVAL<
>INRDCT<
>ATRUBD<

Table Interface Functions

EXTERNAL PARAMETERS:	FILE NAME	
<XATINF>	ACNAM.REQ	function identifier

CALLING ROUTINE: Various functions within the RTM sequencer >XIF<, >XBRANC<, >XWBYTE<, >XREAD<, >XWRITE<, >XCLONE<, >XCLTWO<, >XCLTHR<, >XCLFOU<, >XCLFIV<

CALLING SEQUENCE:

CALL XATINF (ATRAD, ATRLEN, ATRVAL, RMON)

PARAMETERS:

ATRAD	INT;R	ATRTAB address of desired entry
ATRLEN	INT;W	length, in bits, of data item whose ARTAB address is ATRAD
ATRVAL	REG;W	value of entry described by ATRTAB entry whose address is ATRAD
RMON	INT;W	RBIT of entry

GLOBAL DEFINITIONS: None

LOCAL DEFINITIONS:

ATYPE	INT;W	contains the type of ATRTAB entry at address ATRAD

ALGORITHM

```
begin
ATYPE ← ATRBND(ATRAD, <XATINF>)        bind ATRTAB entry
ATRLEN ← GATRLN(<XATINF>)              retrieve length of entry (LEN)
RMON ← GRDBIT(<XATINF>)                retrieve RBIT
ATRVAL ← GSCVAL(<XATINF>)              retrieve RTMSTO value of entry
call INRDCT(<XATINF>)                  increment RCOUNT counter
call ATRUBD(<XATINF>)                  unbind entry
return
end
```

UNCLASSIFIED

3. BLISS-like Code Specifications

CSYMB

FUNCTION: CSYMB converts the output of the NRL ISP compiler symbol to ATRTAB/SYMTAB/RTMSTO format.

DESCRIPTION: The output structure of the NRL ISP compiler is described in NRL1.REQ[252,1111]. CSYMB sequences through the symbol table converting an entry at a time until the entire symbol table has been converted. The TABEL INTERFACE functions are used to establish these tables. BLISS/FORTRAN functions are provided to interface to the NRL ISP compiler symbol table.

CALLING SEQUENCE:

CALL CSYMB

NOTE: Left-hand definition refers to the register or array name on the left side of a redefinition, i.e. in

A<15:0>:=B<30:15>

'A' is the left hand side. 'B', the right hand side, refers to a previously defined register (or array).

CODE SPECIFICATION

PARAMETER FILES:

ACSPAC	REQUIRE	ACcess functions SPACe codes
ACATTP	REQUIRE	ACcess functions ATtribute TyPes
ACNAM	REQUIRE	ACcess functions Subroutine NAMes

FORMAL PARAMETER: NONE

COMMON FILES: NONE

LOCAL STORAGE:

ATPAGE	INT;R,W	ATRTAB Page number
SYPAGE	INT;R,W	SYMTAB Page number
STPAGE	INT;R,W	RTMSTO page number
I	INT;R,W	Index in SYmbol table of NRL ISP compiler output
ATRADR	INT;R,W	Address of currently bound ATRTAB/SYMTAB entry
REG	INT(2);R,W	Contains constant value to be stored
NAME	STRING(var);R,W	NAME of entry (string format)
SYTTOP	INT;R,W	SYmbol Table TOP

EXTERNAL REFERENCES:

RGPAGE	FUNCTION	Reformat Get PAGE
GSYTMA	FUNCTION	Get SYmbol Table entry and Test for MAsk
GSYTRI	FUNCTION	Get SYmbol Table entry and Test for RIght-hand definition
RRESAT	FUNCTION	Reformat REServe ATribute table entry
GSYTCO	FUNCTION	Get SYmbol Table entry and est for COnstant
RGNAME	SUBROUTINE	Reformat Generate NAME
SIXASC	SUBROUTINE	convert SIXbit to ASCii
GSYTAR	FUNCTION	Get SYmbol table entry and Test for ARray
GSYTRE	FUNCTION	Get SYmbol table entry and Test for REgister
GSYTTR	FUNCTION	Get SYmbol table entry and Test for Temporary Register
SATRLN	SUBROUTINE	Set ATRibute table LeNgth
SARDIM	SUBROUTINE	Set ARray DIMension
SSYPNM	SUBROUTINE	Set SYmbol table Print NaMe
SNAME	SUBROUTINE	Set NAME
GSYTLE	FUNCTION	Get SYmbol table entry and Test for LEft hand definition
GSYTLA	FUNCTION	Get SYmbol table entry and Test for LAbel
RRESST	SUBROUTINE	Reformat REserve STorage
SCNVAL	SUBROUTINE	Set ConstaNt VALue
RREDEF	SUBROUTINE	Reformat REDEFine symbol
GSYSYT	SUBROUTINE	Get SYmbol table SYmbol Table top
ATRUBD	SUBROUTINE	ATRibute Table entry UnBinD
GSYPNM	FUNCTION	Get SYmbol table PriNt Name
GSYBCT	FUNCTION	Get SYmbol table Bit CounT
GSYWCT	FUNCTION	Get Symbol table Word CounT
GSYWPT	FUNCTION	Get SYmbol table Word PaTh

UNCLASSIFIED

ALGORITHM

```
!GET TOP OF SYMBOL TABLE
SYTOP<-GSYSYT(0)
!GET FTRST PAGES FOR ATRTAB/SYMTAB/RTMSTO
ATPAGE<-RGPAGE(<ATRTAB>)
SYPAGE<-RGPAGE(<SYMTAB>)
STPAGE<-RGPAGE(<RTMSTU>)
!GO THROUGH SYMBOL TABLE AND BUILD AIRTAB/SYMTAB
FOR I=1 TO SYTTOP DO  !IGNORE FIRST DUMMY ENTRY
 BEGIN
  IF NOT GSYTMA(I) !NOT MASK??
   THEN
    BEGIN
ATRADR<-RRESAT(ATPAGE,SYPAGE,I) !RESERVE AND BIND ATRIBUTE
                                !TABLE ENTRY
      !NOW FILL IN INFORMATION IN THE ATTRIBUTE/SYMBOL TABLE
      !DEPENDING ON THE ENTRY TYPE

      !GSYPNM WILL RETURN THE CONSTANT LENGTH, OR SIXBIT NAME
      !OF REGISTER OR ARRAY AT THIS POINT.

      !IF THE ENTRY IS A CONSTANT, GENERATE A UNIQUE STRING
      !NAME FOR IT.  OTHERWISE, CONVERT THE SIXBIT REPRESENTATION
      !TO STRING FORMAT.

      IF GSYTCO(I) !CONSTANT??
       THEN
        BEGIN
         CALL RGNAME(NAME,GSYPNM(I)) !GENERATE NAME
         !NOW SINCE CONSTANT VALUE IS KNOWN, STORE THE VALUE
         !FIRST MUST LOAD A REGGSTER WITH THE VALUE
         !THE FOLLOWING IS NOT RECOMMENDED PRACTICE IN
         !GENERAL, BUT ALLOWABLE HERE BECAUSE THE REFORMATTER
         !RUNS ONLY ON THE 10.
         REG[1]<-0 !MUST USE     REGISTER!!!!
         REG[2]<-GSYPNM(I)
        END
       ELSE CALL SIXASC(NAME,GSYPNM(1)) !CONVERT SIXBIT TO STRING
      IF GSYTAR(I) OR GSYTRE(I) OR GSYTIR(I) !ARRAY,REG,OR TREG??
       THEN CALL SATRLN(GSYBCT(I),<CSYMB>) !SET REG LENGTH
       ELSE !NOTHING
      IF GSYTAR(I)  !ARRAY???
       THEN
        BEGIN
         REG(1)<-0
         REG(2)<-GSYWCT(I)
         CALL SARDIM(REG,<CSYMB>)  !STORE ARRAY DIMENSION

         REG(1)<-0
         REG(2)<-GSYWPT(I,2)
         CALL SARBAS(REG,<CSYMB>)  !STORE BASE
```

```
IF REG(2) GT GSYWPT(I,0)   !SEE WHICH WAY IT GOES
  THEN
   CALL SARDIR(1,<CSYMB>)!ITS BACKWARDS
END
   ELSE !NOTHING
   CALL SNAME(NAME,<CSYMB>)  !SET THE NAME FIELD
   IF NOT (GSYTLE(I) OR GSYTLA(I))  !NOT LEFT HAND DEFINITION
                                    !AND NOT LABEL
    THEN CALL RRESST(STPAGE)  !RESERVE STORAGE
    ELSE !NOTHING

   IF GSYTCO(I)
    THEN CALL SCNVAL(REG,<CSYMB>)
   !SINCE THE RTM OPERATIONS REFERENCE INDICIES IN THE
   !NRL SYMBOL TABLE, AND THE TABLE ACCESS FUNCTIONS DEAL
   !WITH A ATTRIBUTE TABLE ADDRESSES, AND THE RTMSTO IS
   !GENERATED AFTER THE ATRTAB/SYMTAB, THE ADDRESSES OF
   !THE CURRENTLY BOUND ENTRY MUST BE SAVED IN AN UNUSED
   !FIELD THE PRINT NAME FIELD HAS ALREADY BEEN PROCESSED
   !AT THIS POINT, THEREFORE IT IS USED TO SAVE THE
   !ADDRESS OF THE CORRESPONDING ATRTAB ENTRY.

   CALL SSYPNM(I,ATRADR)  !SAVE ATRTAB ADDRESS
   CALL ATRUBD(<CSYMB>)   !UNBIND ENTRY
   CALL SYMUBD (<CSYMB>)

  END
 ELSE !IGNORE RIGHT HAND DEFINITION-ALREADY PROCESSED
ELSE  !DO MASKS LATER
CALL ATRUBD(<CSYMB>) !UNBINDS THE ENTRY
END

!NOW HANDLE REDEFINITIONS.  SINCE THE SYMBOL TABLE IS HASHED WE HAVE
!NO WAY OF KNOWING THAT THE BASE HAS ALREADY BEEN PROCESSED.

CALL RREDEF   !HANDLE REDEFINES FOR ENTIRE SYMBOL TABLE
```

UNCLASSIFIED

Appendix B

THE ARF CODING STANDARDS*

INTRODUCTION

Most of the code for the Architecture Research Facility (ARF) is to be written in American National Standard FORTRAN†. This standard is considerably more restrictive than most FORTRAN implementations, and certain extensions common to both the PDP-10 and PDP-11 versions of FORTRAN will be permitted. The second section identifies the permitted extensions.

The third section imposes some restrictions on the types of FORTRAN statements that may be used. These restrictions are imposed in the interests of uniformity of style and readability of the FORTRAN code. The fourth section provides an organization guide for FORTRAN main and subprograms produced for the ARF to insure a uniform format. The last section provides procedures and conventions to be followed during coding.

The motivation for this document is twofold. The primary purpose is to assure that as much of the ARF system as possible will run without change on both the PDP-10 and the PDP-11. The secondary purpose is to establish a uniform style and format for the subprograms that are to make up the ARF. This in turn will greatly simplify maintenance and modification of the system. All designers and coders for the ARF system should strive to make their code as readable as possible. The amount of supplementary material necessary for the understanding of a particular section of code should be kept to a minimum. While the algorithms used in developing the code should be designed in an efficient manner, it should not be necessary to use "cute" coding tricks to realize the algorithm. Larmouth's article, "Serious FORTRAN,"** is recommended reading for all ARF coders and designers.

As noted above, ANSI FORTRAN is considerably more restrictive than most implementations. Were the ARF intended to run on any computer with a FORTRAN compiler, it would be necessary to adhere strictly to the letter and spirit of the standard. Since only the PDP-10 and PDP-11 are involved, some extensions common to both versions are permitted. In general, only those extensions that are known to have the same effect in both implementations have been permitted. Because both compilers optimize to different degrees, coders are urged to remember that the FORTRAN standard does *not* require that variables local to a subprogram retain their values between calls.

In any case, where doubt exists as to the compatibility of a construct between implementations, it is the responsibility of the coder to devise a test of the construct and to show that it produces the same result on both machines. Coders should become familiar with both FORTRAN implementations prior to writing any code.

---

*The contents of this appendix were originally drafted by John McHugh.

†American National Standard FORTRAN, ANSI X3.9-1966, Mar. 1966.

**J. Larmouth, "Serious FORTRAN," *Software Practice and Experience* 3, 87-107, 197-225 (1973).

PERMITTED EXTENSIONS

1. The use of the IMPLICIT statement will be permitted.

2. Quoted strings of characters will be permitted in FORMAT and DATA statements. In DATA statements the strings may be no longer than the number of characters held by one word of the data type being initialized. For compatibility, this shall be as follows: Integer, 2 characters/word; real, 4 characters/word; double precision, 8 characters/word; complex, 8 characters/word. Character string manipulation must be done by supplied subroutines to preserve machine independence.*

3. Expressions of arbitrary complexity may be used as array subscripts.

4. Arrays may be initialized with DATA statements, but the whole array must be initialized in a single statement. The "implied DO" form of array initialization is not permitted.

RESTRICTIONS AND CONVENTIONS

1. Three-branch or arithmetic IF statements are not permitted without special permission.

2. Statement labels or statement numbers can appear only on CONTINUE and FORMAT statements.

3. The DIMENSION statement is not to be used.

4. Special permission is required to use the EQUIVALENCE statement.

5. All variables must be declared as to type. This will be enforced by requiring the statement

IMPLICIT COMPLEX (A—Z)

to appear in every program unit.

6. All I/O unit specifiers must be defined as PARAMETERS or held in variables.

7. The PREPROCESSOR/INCLUDE must be used for all declarations of COMMON blocks used in more than one routine.

8. The PREPROCESSOR/PARAMETER statement must be used in all instances where DO indices depend on array dimensions.

9. Program units must not exceed 100 executable statements and units exceeding 50 lines must be justified to the project management. An ideal program unit is one to two pages long including comments, declarations and code.

10. INTEGERS can contain, at most, 16 bits of information. Any quantities with more than 16 bits of information must be represented as RTM registers implemented as DOUBLE

*These standards were devised prior to the conception of the ARF preprocessor STRING facility. Afterward (and before any code was written) the use of the STRING facility was required.

UNCLASSIFIED

PRECISION and must be operated on only by subroutines and functions. (Routines restricted to the reformat program may use 36-bit integers internally.*)

PROGRAM FORMAT

Each program or subprogram for the ARF will adhere to the following format:

1. SUBPROGRAM or FUNCTION statement (omitted for main program).

2. Comments giving the function of the program unit in as few words as possible.

3. IMPLICIT COMPLEX (A—Z).

4. /INCLUDE statements for /PARAMETER files.

5. Local /PARAMETER declarations with appropriate comments.

6. Type declarations for subprogram parameters with comments identifying the purpose of each parameter and whether read or written or both.

7. EXTERNAL declarations for any subprograms used along with type statements for functions, and comments specifying the purpose of the subprogram.

8. /INCLUDE statements for common block declarations.

9. Type declarations for local variables and comments to indicate their uses.

10. DATA statements for local variables (if any).

11. All FORMAT statements (if any).

12. Executable code with appropriate comments.

CODING PROCEDURES AND CONVENTIONS

The following conventions should be followed for file name extensions on files used in the ARF.

.ARF	for FORTRAN routines requiring use of the preprocessor
.FOR	for preprocessor output or FORTRAN routines not requiring the use of the preprocessor
.MAC	for MACRO-10 assembly routines
.REQ	for files to be /INCLUDED by the preprocessor
.CMP	for COMPIL class command files
.CTL	for BATCH control files
.DOC	for specifications, memos, etc.

*The reformat program was never planned to run on the PDP-11.