AD-A069 541 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2
SOFTWARE DEBUGGING METHODOLOGY. VOLUME III. LITERATURE AND SITE--ETC(U)
APR 79 M FINFER, J FELLOWS, D CASEY F30602-77-C-0165
UNCLASSIFIED RADC-TR-79-57-VOL-3 NL
1 OF 3
AD A069541

LEVEL III

A069542 SC

(12)

RADC-TR-79-57, Vol III (of three)
Final Technical Report
April 1979

AD A069541

SOFTWARE DEBUGGING METHODOLOGY

Literature and Site Surveys

System Development Corporation

Marcia Finfer
Jon Fellows
Dan Casey

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

SOFTWARE DEBUGGING METHODOLOGY, VOL III

DDC FILE COPY

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

79 06 06 015

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-57, Vol III (of three) has been reviewed and is approved for publication.

APPROVED:

FRANK S. LAMONICA
Project Engineer

APPROVED:

WENDALL C. BAUMAN, COL, USAF
Chief, Information Sciences Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Entry
1. REPORT NUMBER	(18) RADC-TR-79-57, Vol III (of three)
2. GOVT ACCESSION NO.	(19) TR-79-57-VOL-3
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	(6) SOFTWARE DEBUGGING METHODOLOGY. Volume III. Literature and Site Surveys.
5. TYPE OF REPORT & PERIOD COVERED	(9) Final Technical Report, Sep 77 - Oct 78
6. PERFORMING ORG. REPORT NUMBER	N/A
7. AUTHOR(s)	(10) Marcia Finfer, Jon Fellows, Dan Casey
8. CONTRACT OR GRANT NUMBER(s)	(15) F30602-77-C-0165
9. PERFORMING ORGANIZATION NAME AND ADDRESS	System Development Corporation 2500 Colorado Avenue Santa Monica CA 90406
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	62702F (16) 55810295 (17) 02
11. CONTROLLING OFFICE NAME AND ADDRESS	Rome Air Development Center (ISIE) Griffiss AFB NY 13441
12. REPORT DATE	(11) April 1979
13. NUMBER OF PAGES	199
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	Same (12) 198p.
15. SECURITY CLASS. (of this report)	UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	N/A

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)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Frank S. Lamonica (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Software
Software Debugging
Software Testing
Debugging Tools
Debugging Techniques
Debugging Methodology

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

A debugging study was conducted which surveyed current research being performed in the area of software debugging during integration-level testing. Particular emphasis was placed on assessing debugging tools and techniques which were applicable to embedded software developments. The purpose of the debugging study was to define a software debugging methodology applicable to diverse environments to be utilized during integration testing of system software. The results of the study are contained in three volumes. This volume presents the body of information used to derive the debugging methodology.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

339 900

TABLE OF CONTENTS

Section | Page

SECTION 1 - INTRODUCTION

1.1 Software Debugging Methodology Study Overview . . . 1
1.2 Scope . . . 2
1.2.1 Definitions Basic to the Study . . . 2
1.2.2 External Considerations Impacting Integration Phase Debugging . . . 3
1.2.3 Assumptions . . . 4
1.3 Background . . . 5
1.3.1 A Comprehensive Debugging Methodology . . . 6
1.3.2 The Software Development Life Cycle . . . 7
1.4 Task I Approach . . . 10
1.5 Summary of Conclusions . . . 11
1.5.1 Conclusions from Site Surveys . . . 11
1.5.2 Conclusions from Literature Survey . . . 12

SECTION 2 - SITE SURVEYS

2.1 Advanced Research Center (ARC) . . . 15
2.1.1 Hardware/Software Description . . . 15
2.1.2 Site-Specific Debugging Software . . . 18
2.1.3 Vendor-Supplied Debugging Software . . . 19
2.1.4 PEPE *Debugging* Tools . . . 21
2.2 Rome Air Development Center R&D Computer Facility . . . 22
2.2.1 Hardware/Software Description . . . 23
2.2.1.1 Honeywell 6180 . . . 23
2.2.1.2 RADC Associated Processor (RADCAP) Facility . . . 28
2.2.1.3 QM-1 Emulation System . . . 28
2.2.1.4 PDP-11/45 . . . 29
2.2.1.5 Advanced Research Projects Agency Network (ARPANET) . . . 31
2.2.2 Site-Specific Debugging Software . . . 33
2.2.3 Vendor-Supplied Debugging Software . . . 35
2.3 Telemetry Integrated Processing System (TIPS) . . . 42
2.3.1 Hardware/Software Descriptions . . . 42
2.3.2 Site-Specific Debugging Software . . . 45
2.3.3 Vendor Supplied Debugging Tools . . . 46
2.3.4 Testing and Debugging Methodology . . . 55
2.3.4.1 Software Integration . . . 56
2.3.4.2 Test Levels . . . 57
2.3.4.3 Phase III Testing and Associated Debugging Methods . . . 58
2.3.5 Debugging Methods Actually Used . . . 59
2.4 Space Computation Center (427M) . . . 60
2.4.1 Hardware/Software Description . . . 60
2.4.2 Site Specific Debugging Software . . . 61
2.4.3 Vendor Supplied Debugging Tools . . . 62
2.4.4 Testing and Debugging Methodology . . . 62
2.5 Satellite Control Facility . . . 63
2.5.1 Hardware/Software Description . . . 63

TABLE OF CONTENTS (cont'd)

Section | Page

SECTION 2 - SITE SURVEYS (cont'd)

2.5.2 Site-Specific Debugging Software. 67
2.5.2.1 Flight Support Computer System Debugging Tools. 67
2.5.2.2 Buffer and RTS Computer System Debugging Tools 74
2.5.3 Vendor-Supplied Debugging Software. 77
2.5.4 Testing and Debugging Methodology 77
2.5.4.1 Software Integration. 78
2.5.4.2 Testing Levels. 78
2.5.4.3 Debugging Methods Employed During Development Integration . . . 81
2.5.5 Debugging Methods Actually Used 83

SECTION 3 - LITERATURE SURVEY RESULTS

3.1 Debugging Environment Considerations. 85
3.1.1 Software Management Methodology 86
3.1.1.1 Government Software Management Policies 87
3.1.1.2 Computer Program Verification, Validation, and Certification. . 90
3.1.1.3 Internal Configuration Control. 93
3.1.1.4 Independent Verification and Validation (IV&V). 96
3.1.2 Software Tolerance Requirements 96
3.1.2.1 Role of Debugging in Design Deficiency Resolution 97
3.1.2.2 Effect on Debugging Methodology 98
3.1.2.3 Relation to Tolerance Requirements to Software Management Methodology. 98
3.1.3 Programming Language Considerations 99
3.1.3.1 Languages and Development Methodologies 100
3.1.3.2 Languages and Program Complexity. 103
3.1.3.3 Languages and Error Detection 103
3.1.3.4 Languages and Debugging Tools 104
3.1.4 Interactive vs Batch Systems. 105
3.1.5 System Architecture 106
3.1.5.1 Large Scale Machines. 107
3.1.5.2 Minicomputers 107
3.1.5.3 Microcomputers. 107
3.1.5.4 Distributed Systems 107
3.1.5.5 Real-Time Systems 107
3.1.6 Integration Concepts. 108
3.1.6.1 Top-Down Design and Structured Programming. 109
3.1.6.2 Bottom-Up Development 109
3.1.6.3 Top-Down Development. 111
3.1.6.4 Relation of Integration Concepts to other Debugging Considerations. 113
3.1.7 Testing Methodology 114
3.1.7.1 Computer Program Verification 115
3.1.7.2 Independent Computer Program Verification 121
3.1.7.3 Relation of Testing Methodology to other Debugging Considerations. 122

TABLE OF CONTENTS (cont'd)

Section | Page

SECTION 3 - LITERATURE SURVEY RESULTS (cont'd)

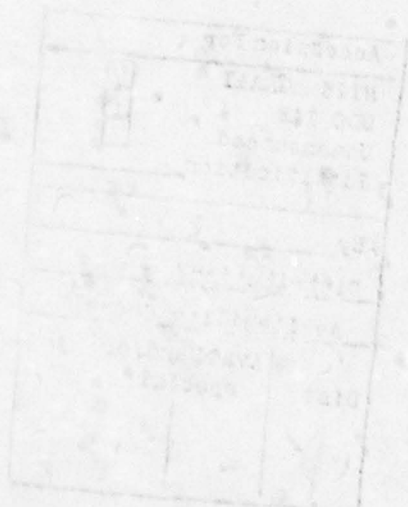
3.1.8 Testing Tools and Techniques 124
3.1.8.1 Static Code Analysis Tools and Techniques. . . . 125
3.1.8.2 Dynamic Monitoring Tools and Techniques. . . . 128
3.1.8.3 Code Testing Tools and Techniques. . . . 129
3.1.8.4 Relation of Testing Tools and Techniques to Other Debugging Considerations 133
3.2 Software Errors. . . . 137
3.2.1 Types of Errors. . . . 137
3.2.2 Error Symptoms 137
3.2.3 Error Messages 142
3.2.4 Relation of Software Errors to Other Debugging Considerations. . 143
3.3 Debugging Aids 145
3.3.1 Debugging Systems. . . . 145
3.3.2 Debugging Tools. . . . 146
3.3.2.1 Dumps/Displays 146
3.3.2.2 Traces 150
3.3.2.3 Set-Use Matrix/Cross-Reference Analysis Tool 152
3.3.2.4 Hardware Debugging Tools 153
3.3.2.5 Breakpoint/Traps 160
3.3.2.6 Reversible Execution/Backtracking. . . . 161
3.3.2.7 Program/Data Modifiers 162
3.3.2.8 Data Recording/Reduction 164
3.3.2.9 Comparators. . . . 165
3.3.2.10 Problem Status Reporters 166
3.3.2.11 Test Deck List Tool. . . . 166
3.3.2.12 System Status Summary Tool 166
3.3.2.13 Checkpoint Tools 166
3.4 Debugging Methods. . . . 167
3.4.1 Preparation Process. . . . 167
3.4.2 Understanding Process. . . . 168
3.4.3 Duplication Process. . . . 170
3.4.4 Elimination Process. . . . 170
3.4.5 Symptom Accumulation Process 171
3.4.6 Debugging Data Collection Process. . . . 172

APPENDIX A - Site Survey Bibliography 173
APPENDIX B - References 176

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special

LIST OF FIGURES

Figure		Page
1.	Software Development Life Cycle	8
2.	Advanced Research Center Computer Configuration	18
3.	TIPS Hardware Configuration	43
4.	Space Computation Center System Architecture	60
5.	AFSCF Computer System Overview	64
6.	SCF Software Community	65
7.	AFSCF Software Development Cycle	79
8.	Software Life Cycle Phases and Development Activities	88
9.	Software Development Life Cycle	91
10.	The Scope of Verification, Validation, and Certification	92
11.	Languages and their Abstractions	101
12.	Bottom-Up Development (Worst case)	109
13.	Bottom-Up Deveopment	110
14.	Top-Down Development - Worst Case	111
15.	Implementation by Builds	112
16.	Typical Development Cycle for Small Microcomputer System	113
17.	Microprogrammable Computer System Development and Validation Activity Chart	114
18.	Overlap of Terms Applying to Testing Methodology	116
19.	Software Errors	138
20.	Testing Tools/Techniques Applicable to Error Cateogires	141
21.	Typical Activation Commands for Debugging Tools	146



PREFACE

This document is a compendium of debugging information obtained from a survey of selected sites and published literature.

The purpose of this document is to provide research data to be used as supporting material to the derivation of a debugging methodology applicable to diverse hardware/software environments.

SECTION 1 - INTRODUCTION

This report is the result of the first task of the Software Debugging Study. It presents the results of an analysis of current software debugging capabilities as represented by a survey of software debugging techniques used at five selected sites and a survey of current debugging-related literature. The purpose of the report is to gather information on software/hardware environments and on the debugging techniques and procedures available and in use at the various sites and described in the literature. The information on debugging techniques, procedures, and environments is the basis for the development of a software debugging methodology and step-by-step debugging procedures in subsequent tasks. The data for this report was obtained from an analysis of material referenced in Annex 1 of the Software Debugging Study RFP, material available in SDC's Software Technology Department Software Engineering bibliography, selected abstracts, NTIS, and conference proceedings. In addition, examination of debugging techniques used at the US Army Ballistic Missile Defense Advanced Research Center, Rome Air Development Center, the USAF Satellite Control Facility, the USAF 427M Space Computation Center, and the SAMTEC Telemetry Integration Processing System, augment the literature survey results.

1.1 SOFTWARE DEBUGGING METHODOLOGY STUDY OVERVIEW

The objective of this study is to develop a methodology for defining a software debugging environment which will be utilized by Air Force software development engineers during the integration phase of software development. The study will be accomplished in four tasks.

- Task I consists of surveying, acquiring, and analyzing information related to current and advanced software debugging techniques. The information will be an analysis of available literature and an examination of techniques available and used at five Government software development sites. The information gathered will be source material for the following tasks.

- Task II will develop a software debugging methodology which provides guidance on the functional processes inherent in debugging, selection of tools, information requirements for software debugging activities, and debugging problems relevant to specific environments. The software debugging methodology will be developed via the definition of a software debugging process model. The model will describe functional processes involved in debugging and their relationship to each other. The model will also depict the various software support tools used in each process. The model will be used to investigate the interdependence of testing and debugging, the effectiveness of software support tools for specific environments, the multiple use of support tools for many functional processes, and the impact, if any, of the type of software under development on the debugging methodology.

- Task III will use the information from Task I and the model from Task II to develop a step-by-step debugging procedure directly applicable to the RADC environment and to a real-time communication system developed on a minicomputer configuration. Each functional process in the model will be examined with respect to the real alternatives that exist in the two environments. The step-by-step debugging procedure will be a translation of the software debugging model to a detailed level of specificity to form a handbook for defining the debugging process.
- Task IV will expand the bibliography in Annex 1 of the RFP with relevant references found in the course of the study.

1.2 SCOPE

Software testing and debugging are a complex set of interrelated tasks which are initiated at completion of the translation of a software design into a machine-executable format. They continue well into the operation and maintenance phase of the software life cycle. The objective of this study is to concentrate on those testing and debugging activities which are immediately concerned with the integration phase of the software development process. To clarify the scope of this study, the following discussion presents definitions which form the basis for investigation and indicate directions in which the study is proceeding. In addition, external considerations are identified which impact the software integration phase and related debugging activities. Further, the assumptions made at the initiation of this task are presented to further clarify the scope of investigation.

1.2.1 Definitions Basic to the Study

The following definitions are presented to familiarize the reader with terminology basic to the Debugging Methodology Study:

- Testing is the process by which a programmer ascertains that a program performs so as to meet its given functional and performance requirements or design characteristics. Testing finds software problems, but does not necessarily isolate their causes.
- Program integration testing is that testing applied to one or more separately compiled modules of code that have been combined, and when executed, interact with each other according to a design intended to solve or partially solve a function necessary to meet a requirements specification. Program integration testing is the test phase dealt with by this study.

- System integration testing is that testing applied to a software package as a whole within the total operational and computational environment in which it is intended to be employed. The debugging activities during system integration testing do not fall within the scope of this study, although many testing and debugging activities discussed are applicable to this phase.

- A test tool is a hardware or software capability which can be used to demonstrate that the program correctly exhibits the desired behavior in order to meet its given functional performance or design characteristics. Test tools find problems, but do not necessarily isolate causes.

- Debugging is the process of isolating the causes of logical flaws in the various transformations used to convert a desired capability into an executable implementation. Whereas testing finds the flaws, debugging finds the causes of the flaws. Debugging requires a representation of the logical models of each transformation used and a means of collecting evidence concerning the manner in which these transformations are flawed. The methodology of debugging concerns the manner in which these logical transformation models are understood and the identification of the nature of the information needed to isolate the cause of a flaw. The tools of debugging concern the collection of this information and the possible application of reverse transforms to produce evidence in a form appropriate to the logical model being analyzed.

- A debugging tool is a hardware or software capability which can be used to augment the symptoms and isolate the causes of an identified flaw to the point where a cause can be envisioned for that flaw. While the same tool may sometimes be used for both testing and debugging, a debugging tool differs from a test tool in the purpose for which it is employed. A test tool finds software problems, a debugging tool helps to find the causes of problems.

- Debugging methods are the ways symptoms of software problems are accumulated, factors eliminated, and a determination is made of what really should happen when the software is operated.

1.2.2 External Considerations Impacting Integration Phase Debugging

This interim report has examined debugging-related tools and techniques from the entire software development life cycle to provide a conceptual framework within which integration phases debugging activities can be considered. The amount of difficulty encountered during the integration phase is directly related to the nature and quality of efforts which have preceded it in the software development process. Accordingly, a study of debugging must inspect the contribution of software engineering tools and practices in the requirements analysis, design, and code phases to the integration phase debugging and testing processes. Section 3 of this report provides a comprehensive review of the relationship of debugging to several major pre-integration activities, such as:

- Software management methodology
- Testing methodology
- Development methodology

An additional dimension of debugging is provided by the computing environment in which problems must be solved. The hardware and software features which create this environment must be studied for their impact on a debugging methodology. Each of these issues has an effect on system complexity, and therefore on the complexity of debugging. The major issues of this type addressed in this report are:

- Choice of programming language
- Physical system architecture
- Sequential versus concurrent processes

In addition, this report briefly addresses the nature of software errors, the ways in which they manifest themselves, and the classes of debugging tools currently available and used for error detection.

1.2.3 Assumptions

The following assumptions limit the scope of this study:

- This study is not concerned directly with testing or program error correction; however, the ways these elements affect the debugging process are addressed.
- While this study is primarily intended for the use of RADC, the approach taken for each task will be general in nature, whenever possible, so that non-Air Force agencies and industry as a whole will find the information useful.
- No attempt is made to define any new debugging tool concepts which should be researched and developed. This study deals only with existing concepts. However, the results of the entire study should indicate deficiencies in current debugging capability and application.
- The results of the site surveys and literature search are assumed to be sufficiently comprehensive in detail and substance so as to provide the information requirements for the subsequent tasks. Where information is lacking, it is assumed to not currently exist and that lack of information will be valuable input for further investigations and directions in which to proceed.

- The computer environments to be examined with reference to this study shall be limited to large scale, real-time, minicomputers, distributed networks, and microcomputer systems.
- Cost factors related to debugging capabilities, tools, and techniques will not be an issue of direct concern to any of the tasks in this study.
- The existence (or non-existence) of particular debugging tools in specific environments will not be a limiting factor in the derivation of the debugging methodology or step-by-step debugging procedures.
- The debugging methodology and step-by-step debugging procedures derived in the course of this study will concentrate on the generalized process of error detection and solution within the testing and debugging framework. It will not specify the detailed analytical thought processes required of the debugger for a specific error in a given hardware/software environment.

1.3 BACKGROUND

Over the past 20 years the major effort in improving the techniques of software development has been directed at the coding, checkout, test, and integration phases of software. This effort has resulted in higher-order programming languages, interactive program development, and a wide range of software verification tools and techniques. In spite of the progress made, the integration phase of software development is the time when major schedule overruns usually occur and a substantial portion of the total software-system life cycle costs are expended. This implies that either effective techniques have not yet been developed, available tools are not used, or effective tools are not applied consistently as part of a comprehensive software testing and debugging methodology. Errors discovered during the integration phase include both design and implementation errors. Although the cause may originate in an earlier phase (i.e., analysis, design, or implementation) and may be eventually ameliorated through the development and application of improved software engineering techniques in those earlier phases, the errors that remain during integration must be found and corrected in the most efficient manner.

Design and implementation errors are often subtle and difficult to diagnose and correct. Design errors originating from inconsistent implementation of data and module interfaces can arise from a variety of factors, ranging from poorly defined formats to problems in information flow. Implementation errors arising from improperly interlocked data structures are very difficult to detect because of the exact interdependency of time and space occuring to cause a failure. Often these conditions can not be reconstructed. Even more important is the fact that these errors frequently slip through the contractor's Computer Test and Evaluation (CPT&E) activities and sometimes appear in the operational system. The testing applied to determine if

errors exist and the debugging applied to localize the cause and correction of errors are of secondary importance only to the verification of proper performance as the main objectives of the software integration phase. The selection and use of adequate debugging tools and techniques commensurate with the software/hardware environment are necessary if this objective is to be met.

Debugging tools that can be utilized during the integration phase may not be available for a specific application or hardware environment, or may not be procured or developed in time for their optimum use. Relatively larger and older systems often have a much greater variety of powerful debugging support tools than do smaller, newer systems and applications. Also, mature systems have had the elapsed time in which to develop the tools and amortize the investment necessary to produce them. Newer and smaller systems generally have neither the time nor the funds available to produce sophisticated debugging tools and small systems often do not have the computing power required to support them. Even if applicable tools are available, they may not be used as part of an integrated, systematic set of procedures. Hence, they may not yield the maximum benefit in finding and correcting errors that they were designed to deliver. Before the implementation of a software system is begun, the program management should have a systematic approach to the selection of software debugging tools for the specific application and environment. Guidance in tool selection and use is important before any large investment is made in implementing techniques for a particular application and equipment.

1.3.1 A Comprehensive Debugging Methodology

A comprehensive debugging methodology is needed to define the criteria for the selection of debugging tools and provide guidance for their proper use in the integration phase of software development. While software debugging tools are generally available and in widespread use, a debugging methodology is needed which provides a more disciplined approach to the testing process by developing guidelines for the employment of these tools. These guidelines should establish criteria on which to determine the sufficiency of a specific tool to meet the information requirements of debugging activities.

The development of a methodology for performing software debugging within a variety of different environments rests, first, upon an understanding of the functional process of testing, error detection, diagnosis, and correction. Secondly, it depends upon an understanding of the current state-of-the-art in debugging tools and techniques, and the availability of the tools that have been developed. Finally, it depends upon an understanding of the specific debugging needs, alternatives, costs, and benefits related to the specific project and its development environment.

Although the process of debugging is generally understood, there is much to be learned in terms of applying specific techniques and approaches. Also, there is a wide variety of new tools now available or being developed that greatly augment traditional debugging activities. To guide selection of a best set for a particular environment requires that the tools be surveyed and evaluated for acquisition and operation costs, benefits/yields,

and current state of availability. To provide guidance in technique and tool selection, this knowledge needs to be condensed and formalized into a set of guidelines.

The applicability of a particular tool to a particular system also needs to be assessed. For instance, systems involving inaccessible or costly sources of live data may require elaborate simulation systems to provide test data and environmental interactions; systems with static data bases may not. Elaborate test beds may be required for new, multi-site systems. Established systems may use an existing site as an integration and test facility. Different approaches are necessary and possible for powerful, large-scale computers as compared to mini or microcomputer installations. Network systems may require more elaborate techniques than single, stand alone facilities. While it is likely that most facilities will require a variety of techniques and methods, guidance in selecting a proper set of tools and techniques is highly desirable for an effective, economic operation.

1.3.2 The Software Development Life Cycle

The functional processes and activities performed in the integration phase can be better understood in the context of the entire software development life cycle. Many of the software problems found and corrected during the integration phase have their origin in previous phases of the system development life cycle. In addition, the activities of the phases following integration cannot be initiated until the software successfully passes the requirements imposed on the development process by the integration phase. For that reason, the entire software development life cycle will be briefly described in the following discussion.

The software development life cycle provides a structured framework of activities and milestones upon which the production of software can be tailored for each specific project. The major purpose of the structured sequence of milestones and activities is to provide software development managers with visibility into technical and financial progress. They help managers to control the process, to identify when problems occur and to take timely corrective action to prevent more serious problems from arising. They also provide a mechanism to help trace software functional and performance requirements to the actual code which implements them. The specific tailoring of the software development life cycle must consider the requirements of each project as well as how the software is procured (i.e., whether the software is developed in-house, or contracted for development by industry, or off-the-shelf software).

The software development life cycle begins when initial requirements for a functional capability to be provided by a software product are defined. It continues through the software product's operational phase because software continues to be developed and modified in response to changing requirements. It is terminated when that software product is no longer used.

Several different software development life cycle processes have been described. Most representative of these is the description contained in AFR 800-14, Volume II, which delineates a life cycle of six phases: analysis, design, code and checkout, test and integration, installation, and operations and support. Often, the phases overlap, and there is no clear demarcation of where one phases ends and another begins. Figure 1 illustrates the computer program life cycle depicting the overlap of some phases, the feedback of information between phases, and some of the milestones.

The analysis phase is that time when the software functional/performance requirements are defined. Generally, a requirements specification describing the user inputs, the user outputs and the required processing is produced during this phase. The preliminary top-level computer program design accomplished during the analysis phase provides validation that the functional capability stated in the requirements specifications can, in fact, be developed. There is a great deal of iteration between the definition of functional/performance requirements and the development of a top-level system and subsystem design. This design is derived as a result of a system engineering process, which not only analyzes and defines the requirements for software but determines the requirements for all software interfaces, including those with other software, computer hardware, and other equipment, and personnel. System engineering studies are generally conducted during this phase, which may include feasibility studies, use of simulation and modeling, computer program design, and, perhaps, even some prototype development and test of high-risk software.

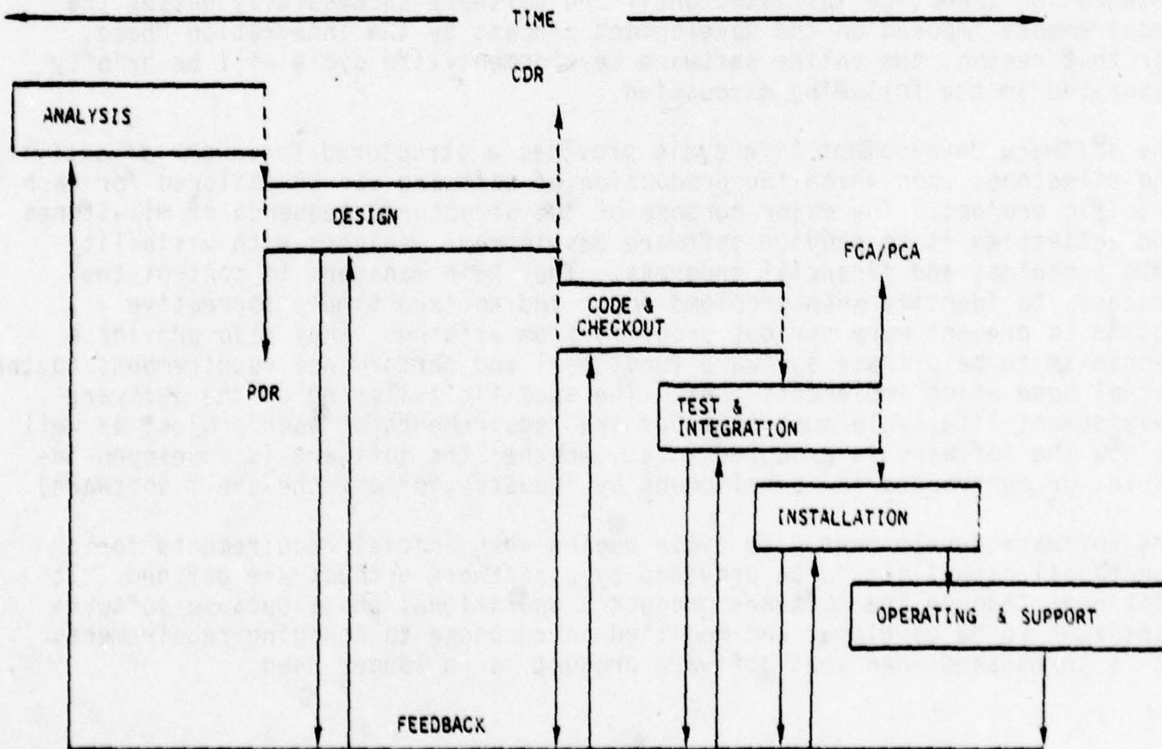


Figure 1. Software Development Life Cycle

The principal activity of the design phase is to determine the major components of the software system/subsystem, the interfaces between these components, and the requirements to be satisfied by each component. In addition, the data base is defined, software engineering studies (e.g., performance measurement) are conducted to verify the design and compatibility of requirements, and system constraints are evaluated. Software sizing estimates are updated and revised. The design phase begins with a Preliminary Design Review (PDR), which provides visibility to both the customer and contractor management into the overall software system/subsystem design. The design phase concludes with a Critical Design Review (CDR). At this time the detailed computer program and subprogram (i.e., module) designs are reviewed to determine that they are compatible, that they meet the requirements of the contract and that they will fulfill the required functional capability as stated in the requirements specifications. During this design phase, test plans and procedures are developed concurrently with the system/subsystem design. The initial test plans should be available at PDR and detailed test procedures should be available at CDR. They should be reviewed by the customer to ensure that the developer's plans for demonstrating software functional/performance capability meet acceptance criteria.

The code and checkout phase is perhaps the most simple and straightforward because it is the best understood and there are numerous support tools and techniques to aid in the process. The activities in this phase include code implementation, test, and debug of modules, programs, and related program data.

The test and integration phase overlaps the code and checkout phase. As computer programs become available, they are tested first individually and then as a group, and are finally integrated into the overall software package. While software testing occurs in this phase, the preparation for testing is pervasive throughout the entire software development life cycle. A testing strategy is usually specified in the proposal and conditions for product acceptance are ordinarily specified in the contract. At the end of test and integration, the software is generally available with all of its supporting documentation for qualification and acceptance by the customer.

The installation phase is when the software is integrated into the overall system. It is generally checked-out using operational personnel. Any incompatibilities between the hardware and software must be resolved at this time to be accepted by the customer. The Formal Configuration Audit (FCA) and Physical Configuration Audit (PCA) are part of the acceptance procedure which determines that the final software Product Specification matches the qualified computer program and that the required acceptance tests have been performed.

The operation and support phase consists of the remainder of the software development life cycle. This phase generally requires some level of effort devoted to maintenance of the software, which often decreases as the operational software is exercised over time, and problem areas are identified and resolved. However, during this phase, changes to the software's performance requirements are inevitable. Depending upon the size of each change, the

entire set of life phases may need to be repeated, including analysis through installation. All of the phases may occur for a given change concurrently with the operation and support phase for the software.

The life cycle phases described above apply to all software development efforts to a more or lesser degree, depending upon the functional/performance software capability required.

1.4 TASK I APPROACH

Although this task, Information Acquisition, has resulted in a large quantity of detailed information relating to various aspects of debugging, it is seen to be a necessary requirement for subsequent analysis which will result in the definition of a debugging methodology and a step-by-step debugging procedure. The work performed in Task I was originally to include a literature search and surveys of the following sites:

- Rome Air Development Center (RADC)
- General Services Administration (GSA)
- U.S. Army Advanced Research Center (ARC)
- U.S.A.F. 427M Space Computation Center (SCC)

After initiation of this task, a decision was approved to substitute the SAMTEC Telemetry Integrated Processing Subsystem (TIPS) site for the GSA site due to availability of the site to the debugging study's personnel. The TIPS project was selected because it consists of a complex network of mini-computers in a real-time environment. In addition, it was found that none of the sites surveyed included the environment specific to an Independent Verification and Validation (IV&V) contractor charged with a set of tasks which concentrated on testing and debugging activities. For that reason, the Air Force Satellite Control Facility (AFSCF) was also surveyed. This site revealed the largest store of information relating to testing and debugging tools and techniques since it is a mature system with established testing/debugging processes.

While much of the information obtained from the site surveys may appear to be overlapping or redundant, it is presented in its entirety for the following reasons. First, it is necessary to identify the debugging capabilities available to a specific hardware/software environment. Once these capabilities are identified, commonality between information systems provided by the debugging capabilities may be observed. Second, project specific development methodology, including testing and debugging processes, need to be examined individually and collectively to determine how best to incorporate specific debugging tools and techniques in a generalized debugging methodology. Third, operating system and debugging tool interfaces need to be examined and defined to identify user information requirements which, when illuminated will optimize testing and debugging activities.

The literature survey was initiated using all available software technology sources of reference. This included the following:

- Bibliography supplied by RADC
- SDC's Software Technology Department's Bibliography
- NTIS abstracts
- Computing Review Abstracts
- Computer and Control Abstracts
- Computer Abstracts
- Computer conference proceedings

In total, approximately 500 references were initially found. Of these, 200 references were read in their entirety. The information obtained from the literature survey forms the background on which the debugging methodology will be based. It should be noted that vendor-supplied documentation was not included in the literature search. (This type of documentation was thoroughly examined during the site surveys.) In addition, the literature survey reflects bibliographic entries encompassing the 1966 to 1977 time period.

1.5 SUMMARY OF CONCLUSIONS

This section presents the preliminary conclusions from Task I, Information Acquisition of Software Debugging Methodology Study. These conclusions shape the direction which the remainder of the study will follow by forming the assumptions for Tasks II and III of the study. The conclusions are presented in the order in which the relevant subject areas are treated in the body of this report.

1.5.1 Conclusions from Site Surveys

The site surveys provided an opportunity to observe debugging activities in several different phases of system development and across several different system architectures. (Section 2 contains the site specific observations which were collected.) The general conclusions follow:

- Planning for debugging, i.e., tools, techniques, and procedures, should begin in the system design phase or earlier. Limited resources and schedule conflicts prevent development of meaningful debug capabilities and procedures on an "as you go" basis.

- Many programmers prefer insertion of their own debug code into a program over the use of available tools. It has not been possible to determine if this preference is due to unwillingness to learn yet another system, or the increased capability to tailor programmer inserted code to reveal abstract program functions.

- Mature software systems employ much more sophisticated project-applicable debugging tools and techniques than do newly established systems. Knowledge of problem areas grows with each iteration of the software development phase processes.

- Vendor-supplied debugging systems have become very sophisticated and powerful but are sometimes awkward to use and do not suitably fit the application. These debugging packages appear to be a good and cost-effective starting point; however, they should be augmented with project-specific debugging tools.

- Computer time availability influences the form of debugging practiced more than any other factor.

1.5.2 Conclusions from Literature Survey

The following general conclusions were drawn from the literature survey:

- Software Management Methodology. For projects of reasonable size, a well structured system of configuration management and development phase milestones, documentation, and acceptance criteria is essential to the establishment of a stable debugging environment. Of particular importance to debugging are software version control and prioritized modification review procedures.

- Tolerance Considerations. The problems concerning a system's inability to meet performance or size requirements will not be considered in the remainder of this study. The solution to such problems is design reiteration, which is different in nature from the debugging of logical problems.

- Language Considerations. The choice of a programming language impacts the debugging effort to a marked extent. The debugging tools associated with a language reflect the behavior of a "virtual machine" implemented by that language. The more abstract the language, the more powerful the automated error detection capability can be.

- Interactive vs Batch Systems. The use of interactive symbolic debugging tools is unquestionably one of the major advances in debugging methodology. These tools will receive detailed consideration in the body of the debugging methodology study.

- System Architecture. The primary impact of computer architecture on a debugging methodology is the availability of sufficient resources to support the overhead implicit in the use of most debugging tools. The debugging methodology should provide guidance on the amount of resources needed for a specific application of debugging tools and techniques. The initial hardware selection criteria should include provisions for this overhead, otherwise the selection of debugging tools will be severely limited. A second architectural consideration is the complexity introduced by distributed or real-time system architectures. The literature survey provided very little insight into a methodology which can deal with this complexity. This area will become a major subject of study and analysis in the remainder of the debugging study.

- Integration Concepts. Advantages and disadvantages exist for both the top-down and bottom-up development approaches. Either can be effectively applied, depending on the nature of the software system being developed. For purposes of defining a debugging methodology, both approaches will be considered.

- Testing Methodologies. The standard Air Force testing processes are suitable as background for the debugging methodology. The processes consist of the CPC code-and-test, CPC incremental-integration testing, CPCI testing, PQT, and FQT testing stages.

- Testing Tools and Techniques. The testing tools and techniques available to aid in discovering the errors dealt with by the debugging process have been found to be very dependent on the type of software system involved. The disparity between large scale systems and microcomputer systems in this regard, for example, appears to require separate consideration for the effect of testing tools and techniques on the debugging methodology.

- Software Errors. The literature survey has shown that definite relationships can be established between testing tools and techniques and software errors. Since the debugging process operates on the symptoms of errors rather than the error themselves, a principal task of the debugging methodology development appears to be establishing the relationship of software errors to error symptoms, and error symptoms to debugging tools and techniques.

- Debugging Aids. Except for new system-specific hardware debugging aids, there appears to be no new or unique debugging tool concepts within the past 10-12 years, when interactive systems began to flourish.

- Debugging Methods. The methods used by the debugger in resolving a software problem can be defined in terms of the tools and techniques employed. This should result in a meaningful input to the debugging methodology development process.

SECTION 2 - SITE SURVEYS

The analysis of current debugging technology is a prerequisite to formulating a methodology applicable to diverse hardware/software environments. While examination of software packages supplied by hardware vendors forms the bulk of information relating to current debugging capabilities, this study has surveyed five selected software development sites. The purpose of the site survey was twofold. First, attitudes of personnel within a specific software development environment were sampled to determine their personal preferences in the usage and adequacy of the vendor-supplied software debugging capabilities. Second, the capabilities provided by vendor-supplied software packages have, in many cases, been augmented by site-specific debugging tools and techniques and these were surveyed to determine how available debugging capabilities were incorporated into a systematic procedure. Both programmer preferences and the integration of testing/debugging capabilities in the development process are necessary components to the derivation of a debugging methodology and a handbook for debugging is the eventual goal of this study. The sites selected to be surveyed include:

- U.S. Army Advanced Research Center (ARC)
- Rome Air Development Center (RADC)
- Space and Missile Test Center (SAMTEC) Telemetry Integrated Processing Subsystem (TIPS)
- USAF 427M Space Computational Center (SCC)
- USAF Satellite Control Facility (AFSCF)

The situation of sites was based in part on the immediate accessibility of the debugging project personnel to personnel within each facility because of current SDC contract work. In addition, each site contained diverse hardware/software characteristics which present differing environments on which testing and debugging activities are performed. Further, each development project survey reflected a different phase of the software life cycle. For example, several projects within RADC and the ARC are phasing down or are in transition. Although the analysis of debugging capabilities for these projects are not greatly detailed, personal hindsight on debugging capabilities that would have significantly aided in software verification contributes to the overall objectives of the debugging study.

Much of the information presented in this section is very detailed in its examination of debugging tools and techniques. The intention of including such detailed information is to provide a firm basis for the information requirements for subsequent tasks associated with this study.*

---

*See Appendix A for bibliographical references of documents used in the preparation of this section.

2.1 ADVANCED RESEARCH CENTER (ARC)

The U.S. Army Advanced Research Center is located in Huntsville, Alabama. This facility is operated by System Development Corporation (SDC) to support the Data Processing Directorate of the Ballistic Missile Defense Advanced Technology Center (BMDATC). The purpose of this directorate is to supply computer resource and technical support to each of the other BMDATC directorates (i.e., Technology Analysis, Missile, Optics, Radar, Discrimination). Each BMDATC directorate is charged with performing research in a different area of technology and each investigation addresses a different phase of the research problem. In addition, each directorate works with a set of contractors, many of which eventually use the computer facility and the technical support supplied by SDC. This center provides a centralized capability for cost-effective research, development, analysis, and demonstration of data processing and hardware technology, and their interrelations.

2.1.1 Hardware/Software Description

The data processors provided in the ARC provide a real-time computer configuration dedicated to advanced research in BMD software and data processing. The machines currently available are configured as shown in Figure 2, with the real-time software in the Texas Instruments Advanced Scientific Computer (TIASC) being driven by the system, environment, and threat simulations on the Control Data Corporation (CDC) 7600 through a realistic hardware computer/radar interface.

A short description of each hardware component follows:

- Control Data Corporation 6400. The CDC 6400 is a medium-capacity computer with 100 nanosecond memory cycle time. It is equipped with 65K words (60 bits) of Central Memory (CM) and 125K words of Extended Core Storage (ECS). The CDC 6400 functions as an input/output system for the CDC 7600, and provides communication for remote batch users. The operating system for the CDC 6400 is SCOPE 3.4, a multiprogramming system with a large complement of languages and utilities. It also serves as an I/O station for the 7600 computer and supports interactive terminals through INTERCOM. The INTERCOM 4 system is used to drive and communicate with all remote terminals. Batch processing and the storage of permanent files is not permitted on the CDC 6400 except for special requirements because the resources are not adequate to both operate as a stand-alone system and support the 7600.

- Control Data Corporation 7600. The 7600 is a powerful scalar processor with 27.5 nanosecond effective memory cycle time. It is equipped with 65K words (60 bits) of Small Core Memory (SCM) and 512K words of Large Core Memory (LCM). In addition, it executes programs in a multi-processing environment.

  The SCOPE 2.1 operating system for the CDC 7600 is compatible with SCOPE 3.4 on the 6400. Job control cards, language processors, and utilities are the same or very similar.

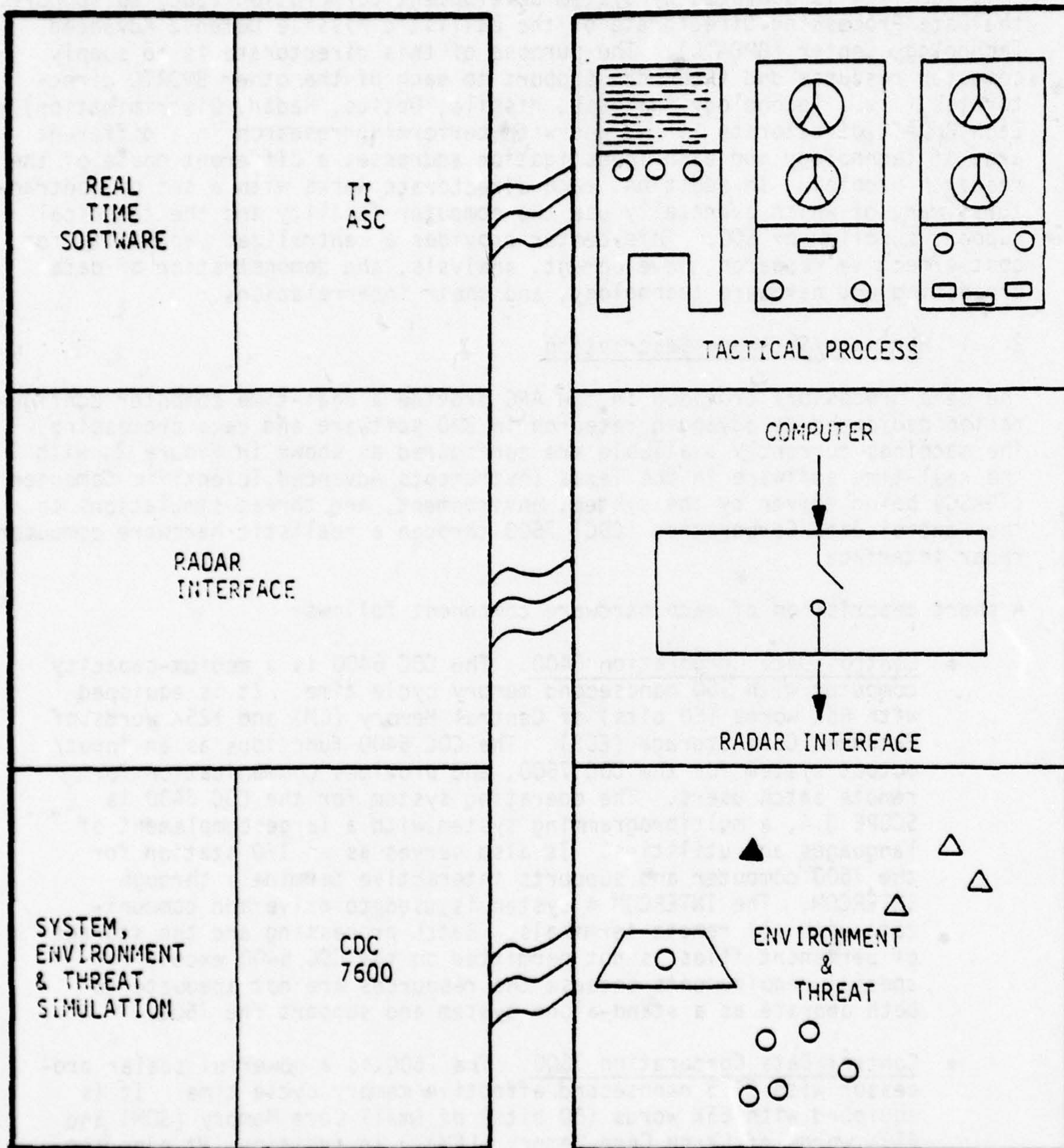


Figure 2. Advanced Research Center Computer Configuration

The 7600 in the ARC does not support unit record I/O or interactive terminals. With the exeption of on-line disk files and tape drivers, all I/O and operator control commands are provided by the 6400.

An input tape can be mounted on the 6400 and staged through the station to the 7600 or can be mounted on the on-line drives and stages. The staging process copies the whole tape to a disk file and releases the tape drive. Thus, reads and writes operate on a disk file rather than on a tape and then, when closed and unloaded, are staged to a tape. Huge files that might, for example, occupy 20 reels of tape are built as a series of smaller files to avoid exhausting the disk capacity on the 7600.

In addition, it is desirable to use the 7600 on-line tapes in preference to the 6400 tapes that must be staged across the channel to the 7600. The former are faster and they do not tie up the 6400/7600 channel. Although seven track drives are available on both the 6400 and 7600, only the 7600 has nine-track capability.

- Texas Instruments ASC. The ASC is a pipeline processor with both scalar and vector instructions, the latter allowing it to perform the same operation on every element of an array with only one instruction. The clock cycle on the 32-bit word machine is 70 nanoseconds. It has 128K words of bipolar memory (140 nanosecond read time) and 512K words of Memory Extension (MES). The ARC ASC has one pipeline (arithmetic unit), but up to four are possible.

  The General Purpose Operating System (GPOS) installed on the ASC is designed to provide the services expected of a large-scale, multiprogramming system. The ASC's central processor has vector and scalar instructions. The operating system has been modified to support the interactive graphics system. Nine and seven-track tapes are both supported by GPOS.

- Parallel Element Processing Ensemble (PEPE). PEPE is a special purpose computer designed to augment conventional sequential computers. PEPE architecture features a highly parallel, associatively addressed ensemble of Processing Elements (PEs) operating under global control units. The ARC PEPE is designed for 288 PEs, however only 11 are implemented. A PEPE instruction-level hardware simulation system has been implemented on the CDC 7600 which executes PEPE programs requiring up to 288 elements. The control units contain a total of 52K words (32 bits) of bipolar memory for program and data storage. In addition each PE contains 2K of bipolar memory for data storage. PEPE was designed by SDC with hardware construction performed by Burroughs. All software was developed by SDC.

The PEPE Real-Time (or Run-Time) Execution (RTE) augments the CDC 7600 SCOPE 2 operating system with 7600 and PEPE resident executive components that allow flexible and dynamic task scheduling. PEPE code may be executed on the 11-element hardware system or on the 288-element emulated system.

2.1.2 Site-Specific Debugging Software

Although little software has been developed within the ARC to support specific debugging capabilities, the following programs are available:

- FORTRAN Analyzer Program (FORAN). This program analyzes the usage of program labels, tags, data variables, constants, subroutines, and other program elements for a main program and its related subroutine components. The information output includes the item name, the FORAN assigned statement numbers in the program or subroutine where the identified item is referenced, and a code indicating how the item is referenced (e.g., assigned, used, input, output, a system external reference, a DO variable, a subroutine CALL, etc). In addition, FORAN can provide a composite cross reference of data usage for separately analyzed program units. FORAN also performs certain consistency checks on the data, including:

  - Symbols defined but not used
  - Variable type and dimension
  - Number and type of parameters in functions and subroutines

  FORAN's primary use in debugging is to determine possible computation or logic errors by providing a static analysis of data usage. It is also valuable in analyzing the effect of a program modification on data usage. FORAN can analyze any dialect of FORTRAN. During the syntax analysis, syntax errors are flagged and the analysis is continued.

  FORAN is easy to use and its output contains more information and is easier to read than the compiler's symbolic reference map. It is limited to 4095 manual items and a total of 24,000 unique references for all named items.

- MERGEL. MERGEL allows replacement of changed subroutines and the addition of new subroutines to an existing binary file. Thus, Central Processing Unit (CPU) time is saved by not requiring a full update and compilation each time a line of code needs to be changed. This program is useful after an error has been found in a subroutine which then is recompiled or reassembled.

2.1.3 Vendor-Supplied Debugging Software

The debugging tools available under the CDC 6400/7600 Scope 3.4/2.1 operating system are provided by Cyber Interactive Debug (CID), FORTRAN Interactive Debug (FID), SCOPE, and special software developed or obtained/modified by SDC. The following capabilities are provided on the CDC 6400/7600:

- Cyber Interactive Debug. CID provides debugging capabilities which can be used to facilitate checkout of programs by allowing an interactive terminal user to monitor and control the execution of programs. It is primarily intended to be used interactively, but may be used in batch. CID provides facilities independent of any particular source language, but it is also possible to interface source-language dependent modules to CID to handle languages such as FORTRAN or BASIC. CID features include the ability to:

  - Set breakpoints at specific locations such that user program execution will be suspended when it reaches one of those points, allowing the user to monitor the state of the program.
  - Set traps such that program execution will be suspended on specific events such as the loading of an overlay.
  - Interrupt and restart the user program from the terminal.
  - Display variables in the user program.
  - Enter data into the user program.
  - Define and save sequences of debug commands to be executed when a breakpoint or trap occurs.

- FORTRAN Interactive Debug. FID commands are used to debug programs written in FORTRAN and compiled by the FORTRAN Extended* compiler in debug mode. FID commands are designed to provide access to the program in terms of the FORTRAN language; its variables, constants, and statement forms are FORTRAN forms. FID commands resemble straightforward FORTRAN statements. The FID commands allow the user to display program variables, assign values to program variables, and execute conditional statements. FID commands include:

COMMAND	PURPOSE
PRINT	Simulates the FORTRAN list directed Print statement, allowing display of many variables with one command.
Assignment	Assigns calculated values to FORTRAN variables. Strictly speaking, assignment is recognized by the presence of an equal sign in the command line.

*FORTRAN Extended (FTN 5) does not support the debugging features described above. CDC is in the process of providing a similar debug package.

COMMAND	PURPOSE
IF	Allows the user to execute other debug commands conditional on relationships between expressions.
GOTO	Causes the user's program to resume executing at at specified point in the program.
TRACEBACK	Displays a trace of the path the program took to get to the current subprogram.

- SCOPE 2.1. The operating system provides two capabilities that can be used for debugging:
  - DMP. DMP is a control card callable routine that may be used to dump selected positions of the user's field length as well as the current status of the operating registers.
  - TDMP. TDMP is a special utility program called by a control card and used to dump disk files. It is primarily used as a debugging aid since it can dump various portions of a file under user control.

- Subroutine Timer. A performance measurement program has been modified to be run on the CDC 7600 that will provide timing information at the subroutine level. No changes are required to the timed program; the timer reads the memory map produced by the loader and traps all user subroutines at entrance and exit to obtain their run times. In addition to information on individual routines, the timer provides a flow analysis showing which routines called which others. This output is printed twice to show everything from the viewpoint of the calling program and from the viewpoint of the called program. The print-out gives the following designated sequence for each of the subroutines which were executed at least once:

DEPTH	Number of subroutines involved.
TIMES EXECUTED	The number of times the subroutines in the sequence were called.
AVERAGE	Average number of calls from one subroutine to another each execution of the sequence.
PERCENT	Percent of calls from one subroutine to another, relative to all calls made from the initial subroutine in the sequence to any other routine.
CALLS MADE	Numer of calls made by a subroutine to other routines.

2.1.4 PEPE Debugging Tools

The debugging tools available under PEPE RTE include:

- PEPE Emulator Trace Output Processor (POST). POST is a postrun processor that selectively prints PEPE instruction trace data. The selection may be based on time periods, instruction types, PEPE processor, and other controls. POST executes on the 7600.

- PEPE Dump Print Program. No material on this program was obtained during the trip to the BMDATC.

2.1.5 Texas Instruments ASC

No information was available on the debugging tools under the TI ASC GPOS.

2.2 ROME AIR DEVELOPMENT FACILITY R&D COMPUTER FACILITY

RADC is a research and development facility charged with the establishment, operation, and maintenance of computer and display facilities for the support of:

- Exploratory development programs in information sciences
- Research and development in advanced computer techniques
- RADC requirements for scientific computational services

The current internal structure of the research and development facility consists of five divisions, including: Intelligence Reconnaisance, Surveillance, Information Sciences, Reliability, and Communications. Each division concentrates on a specific area of technology which evolves according to user needs and the state of current hardware/software technology Accordingly, the specific application area under investigation and the hardware/software environment specific to each group of users is dynamic and cannot easily be characterized.

The computer systems available to RADC personnel and contractors working on RADC projects include:

- Honeywell 6180/Multiplexed Information and Computing System (MULTICS)
- Honeywell 6180/General Comprehensive Operating System (GCOS)
- RADC Associative Processor
- QM-1 Emulation System
- PDP-11/45
- Advanced Research Projects Agency (ARPA) Network

The computer systems which are used most frequently by the user groups are the GCOS and MULTICS systems, each on a Honeywell 6180. FORTRAN and JOVIAL currently appear to be the programming languages most frequently used. The classes of activities that impact the amount and type of computer resources and support software required include:

- Scientific and data handling activities which directly and indirectly support the overall RADC purpose. Both the time-sharing and batch facilities of GCOS are used for standard GCOS production jobs. The primary languages used are FORTRAN, BASIC, and the time-sharing subsystems.

- Software system development activities which tend to be quite large, sophisticated, and highly evolutionary. Software developers require multiple turn-arounds per short intervals of time and small cpu utilization for debugging. Software development requires large amounts of permanent file and removable disk packs and tapes. Consequently, system development may require many more peripheral resources than other activities. Delivery dates generally produce high levels of activity. Languages such as JOVIAL, COBOL, FORTRAN, GMAP, and SIMSCRIPT are used. Both GCOS and MULTICS are used; however, MULTICS may be better suited for R&D work.

- Software system testing activities whose principal characteristics is the need for special conditions and equipment configurations. Standard software services are used in the support roles, such as editing test programs and test tapes. Actual testing usually requires dedicated machine time and sometimes a non-standard or special version of the operating system. Test teams frequently require long elapsed time between machine runs. Some software system testing is done during regular processing hours if the testing can co-exist with the regularly scheduled workload. Frequently the output requirements due to dumps are very large.

- Modeling activities for radar systems, base intrusions defense systems, electromagnetic compatibility analysis, reliability analysis, electronic components, and elements of management information systems. Models are run at variable intervals ranging from monthly to daily. During the preparation stage the needs of model developers are similar to those of software system developers. After reaching the production stage, however, models use large amounts of core space and processor time and produce large outputs. Both GCOS and MULTICS are used for modeling.

2.2.1 Hardware/Software Description

The following discussion describes the hardware/software configurations within RADC.

2.2.1.1 Honeywell 6180

The 6180 computer system consists of:

- Two Honeywell 6180 CPUs (one dedicated to MULTICS and one to GCOS).

- One million words of core storage

- DSU - 451 Multics (6) 156,000,000 bytes (9 bit)
  DSS - 191 GCOS (5) fixed 118,778,880 char (6 bit)

- 3 line printers
- 8 tape drivers
- 2 Datanet 355 communications processors (currently handling about 80 time-sharing users simultaneously)
- One off-line plotter

The MULTICS and GCOS operating systems are briefly described in the following paragraphs.

MULTICS Operating System

The MULTICS operating system was developed by MIT with funding provided by DoD/ARPA. The MULTICS operating system is written in PL/1. Other languages supported are FORTRAN, APL, BASIC, and COBOL 74, but the emphasis is on PL/1, both as an application language and as the system's own implementation language. MULTICS exploits virtual memory hardware and technology to provide:

- Secure Operation. Security of files from undesired access is maintained through the use of a ring-structure. Programs and data, as well as MULTICS itself, are assigned operational rings (domans of operation), and access permission is required for a program assigned to a given ring to corss into another ring. In all there are eight rings of which only the outermost is normally available to the user. The others are reserved for special use and system use.

- Continuous Service. MULTICS provides a hierarchical file structure and automatically protects user's files by performing an incremental dump to tape at regular intervals with the result that a user should never lose more than a half-hour's work due to a system malfunction.

- Growth in Capabilities. Dynamic linking provides the capability of writing programs with conditional calls to large and sophisticated programs or error routines. Unless the call conditions are satisfied, the user will not have to pay the cost of locating, linking, and executing the called program.

- Interactive Operation. MULTICS can be accessed via a multitude of commercially available computer terminals either directly or via the ARPANET. Programs can be prepared, executed, and documented through an interactive keyboard interface which most eliminates the need for punched cards. The system does much of the work for the programmer, enabling useful program execution at the earliest possible point.

The MULTICS features and software include:

- Absentee Capability. Creates an absentee job which is similar to a batch job on other systems. Absentee usage gives the user the ability to execute runs without waiting at the terminal while the run is in progress or while performing other functions.

- APL. IBM-developed engineering-oriented language.

- Archiving. Allows test archiving.

- Binder. Provides preloading and linking for subroutine communication.

- Debug. Interactive source debugging tool.

- CALC. Desk calculator.

- Exec-com. Powerful job control languages

- EDM. Simplified text editor.

- Fortran. Higher order language for math and science applications.

- GCOS Environment Simulator. Supports GCOS batch programming environment.

- GUS. Graphics Users System.

- PL/1. Higher level algebraic programming language.

- QEDX. Programmable text editor.

- Runoff. Text formatting facility.

- Sort/Merge. Sort and merge package.

- MULTICS Relational Data Store. Programmer services for relational type of data base organization.

- MULTICS Integrated Data Store. Programmer services for CODASYL/IDS II type of data base organization.

GCOS Operating System

The GCOS operating system features include source management, job scheduling, data base management, security control, multiprocessing, multiprogramming, and others. It provides both time-sharing and batch operation. RADC's time-sharing system (TSS) can handle up to 64 simultaneous users. Users can access the computer by means of locally available computer terminals. Batch operation can be accessed from time-sharing through a subsystem called CARDIN; however, the standard mode of operation is to submit programs and data on cards or tape.

GCOS comes with a wide variety of programming languages and service subsystems. These include the following time sharing software.

- Abacus. TSS acts as a desk calculator.
- Access. Conversational file space management system.
- Basic. Beginners all purpose symbolic instruction code.
- Cardin. Allows a user to enter a batch job from TSS.
- Data Basic. Terminal-oriented data processing.
- FDUMP. Word-oriented file insepction/maintenance facility. for permanent files regardless of format.
- JOUT. TSS facility for manipulating batch output.
- RBUG. Conversational debugging facility.
- Runoff. Allows user to format a file printed at a terminal.
- SCAN. Provides facility to conversationally examine output of batch jobs saved on permenent files.
- Text Editor. Text entry and manipulation system.

The batch software includes:

- Algol. Higher-order programming languages.
- BMD. Bio-medical statistical programs.
- Bulk Media Conversion. Converts jobs from one storage media to another.
- COBOL. Higher-order language used for business applications.

- ECSS II. Extended Computer Simulation System from the Federal Performance and Simulation Center (FEDSIM).

- File and Record Control.

- FORTRAN. Higher-order language used for math and science

- G-225 Simulator.

- GCS. West Point's device-independent graphics package.

- GMAP. GCOS assembly language.

- Indexed-Sequential Processor. Allows access of direct-access files in either a random or sequential mode.

- Integrated Data Store (IDS). COBOL/FORTRAN-oriented file management system.

- Jovial. Higher-order programming language suitable for system bulding.

- SIMSCRIPT II.5. An extended version of SIMSCRIPT from CALI, Inc.

- Sort/Merge.

- Source and Object Library Editor. Allows on-line creation of and maintenance of user libraries.

- System Library Editor. Allows creation, modification and maintenance of system libraries.

- Transaction Processor. Processes remote transactions.

- Utility. Used for operational and debugging purposes, permits copying, comparing, positioning, and printing storage-device data.

2.2.1.2 RADC Associated Processor (RADCAP) Facility

The RADCAP facility uses the STARAN parallel-array processor to explore the performance of associative computer architecture on real-world, real-time problems with the specific goal to determine the cost-effectiveness of associative/parallel processing. Associative processing has been studied extensively in both theoretical and simulation studies, but no significant practical operating experience with it currently exists. Experimentation is necessary to provide hard data. Practical operating system experience is also required to develop a general purpose associative processor configuration, if results warrant it.

STARAN is a parallel array processor, with four arrays interfaced with the Honeywell 6180 via an I/O channel and a Custom Input/Outout Unit (CIOU). Additional hardware includes a disc storage subsystem, a hardware performance monitor and various peripheral devices. The STARAN system can perform search, arithmetic, and logical operations simultaneously on any or all bit slices or word slices of its associative memory.

The STARAN system software is based upon a disc operating system and has a batch processing capability. In addition, language processing and operational software are available. STARAN software can operate in a stand-alone mode and, via an interface with MULTICS, in an integrated mode. In the STARAN/MULTICS mode, commands to the STARAN disk operating system can originate in MULTICS, the MULTICS storage system is available to STARAN users for program or data storage, and a single task can use both machines to verify its processing requirements.

2.2.1.3 QM-1 Emulation System

The QM-1 emulation system provides a technique for:

- The study of system architecture
- The development of an unique instruction repertoire
- The test and evaluation of new computer designs
- The development and validation of microprocessor firmware and software
- The emulation of obsolete computers

The QM-1 is an extremely flexible machine which can be programmed at two distinct microprogramming levels (Control store level, 18-bit words; Nanostore level, 360-bit words). System components include two magnetic tape units, a 60 M Byte disk unit, card reader printer, cassette tape unit, and a CRT console. The system contains 128K words of core, 16K words of semiconductor control store, and 512 words of semiconductor nanostore.

QM-1 provides a basic operating system for writing, editing, running, and debugging emulations. Assemblers for both levels of user microcode are provided. Once a system is emulated, the operating system for the emulated machine can be loaded. Emulations currently exist for such diverse machines as the IBM 360 and PDP-11/10.

2.2.1.4 PDP-11/45

The PDP-11/45 is a major element of RADC's Pattern Recognition Design Facility. It, and the Applied Dynamics A/D-5 analog computer, program the following capabilities:

- Long Waveform Analysis. An interactive software system designed to digitize and display analog data on a Tektronix 4002A display. It is used to perform spectral analysis on very long waveforms.
- Waveform Processing System (WPS). An interactive graphic system for the evaluation of extraction techniques for waveform pattern recognition problems.
- Feature Extraction Software System (FESS). Generates a data base of features extracted and defined from analog data by WPS.
- On-line Pattern Analysis and Recognition System (OLPARS). An interactive graphics system for the solution of feature data analysis and pattern classification problems.

There are two hardware configurations in the Pattern Design Recognition Facility. One is built around the PDP-11/45 computer and the other around the Honeywell 6180/MULTICS System. There is a direct hardware link between the two computers allowing for interchange of data between the two systems. The user of this facility is also its computer operator. Therefore, it is imperative that potential users become qualified system operators before any utilization begins. The PDP-11/45 configuration is composed of:

- Digital Equipment Corp., PDP-11/45 digital computer with 76K of core and the following options:
  - Hardware floating point processor.
  - Seven/nine-track magnetic tape.
  - DEC tape.
  - Fixed head disk file, 256K words.
  - Cartridge disk file, 1.5 million words.
  - Disk packfile, 10 million words

- Analog to digital converter.
- Card reader.
- Digital plotter
- Paper tape unit.
- Store terminal
- Image dissector camera.

- Vector General Graphics terminal with the following:
  - Three-dimensional rotation, translation, scaling of display image.
  - Light pen
  - Data tablet.
  - Alphanumeric keyboard.
  - Functions keys.
  - Intensity modulation.
- Applied Dynamics analog computer with the following:
  - 196 amplifiers.
  - 24 comparators.
  - Function generators (including since, cosine, dual log, and variable).
  - Digital logic (including flip-flops, gates, etc.)
  - Hybrid control.
  - A-to-D and D-to-A converters.
- Analog instrumentation including the following:
  - Analog tape units
  - Tunable filters
  - Spectrum analyzer
  - Probability and correlation analyzer
  - Strip chart recorder
  - Storage CRT display

The hardware is configured such that the operator utilizes the Vector General terminals as the primary control device for the problem's solution as it progresses. All of the pattern recognition software is written with the Vector General graphics terminal or the storage tube terminal as the controlling device. The anolog computer is connected to the digital computer through a hybrid interface, thus allowing the capability of processing in a hybrid mode. The analog tape units as well as the analog instrumentation allow the operator to begin the raw analog data and perform the required preprocessing as well as analog to digital conversion prior to the actual data analysis and processing within.

2.2.1.5 Advanced Research Projects Agency Network (ARPANET)

ARPANET is a system of geographically separate, independent, heterogeneous computers, linked together in a computer-to-computer communications network. One use of such a computer network is to provide a software programming environment consisting of a number of heterogeneous computers with a large set of program development tools which are administratively and geographically dispersed. An approach to a network system providing this capability is the National Software Works (NSW) available to RADC users via the ARPANET.

The NSW is a DoD research and development project for developing ideas and systems for increasing programmer productivity and reducing the computer costs of software systems by providing programmers with a set of integrated tools (e.g., text editor, compilers, assemblers, and debuggers) which can be used in a software production activity. In addition to software production tools, the NSW provides the manager with a set of tools for monitoring through the use of tool and file access control mechanisms, and collects project status and programmer activity information which are used by the management tools.

NSW consists of: (1) the Front End (FE) through which the users access NSW; (2) the Work Manager (WM), the access granting, resource controlling, central component; (3) the Foreman (FM) modules, which interface tools in the Tool Bearing Hosts (TBHs) with the WM; (4) the File Package (FP) modules, which are responsible for file translation and movement; and (5) communication protocols (MSG) that specify the communication links between the various NSW components.

- Front End. FE provides direct communication between the WM and the user. FE interprets the user's input character stream and directs user requests to the appropriate component. Functions performed include establishing appropriate communication paths which enable the user to access tools and for NSW to report errors, progress, etc., to the user.

- Work Manager. WM services user requests for system resources (e.g., running a tool, opening a file, etc.), and determines if the request is valid. Since WM has access to the description of the characteristics of all tools within NSW, the validation of a work request can be unusually exhaustive. (For example, Does the tool exist? Do the input files exist? Are they of the

right type and status for this tool? Are the files and the tools both of the right status to be used by this user at this time? WM allocates to each valid request the necessary resources. This allocation process involves either the activation of a tool for the user or the movement of a file, which may be either interhost or intrahost.

- <u>Foreman.</u> FM takes a well-defined and fully validated request for tool use and gets the job done. There is an FM at each individual TBH. This component controls user access of the tools and makes NSW resources, such as NSW files, available to the activated tools. Thus, a tool gets NSW resources by making a local call on FM, which then forwards the request to the appropriate NSW component. In essence, FM provides NSW interface for the tool, and tools see, through FM, an extended local system environment.

- <u>File Package</u>. The primary function of the NSW FP is to create a copy of an NSW file which will be suitable as input for a tool, that is, to make the output of one tool (e.g., an editor) acceptable as the input of another (e.g., a language processor or compiler). There is an FP at each TBH.

- <u>Tool Bearing Hosts.</u> TBH is a computer system that hosts software development tools which are made available to users through NSW. To become a TBH, the computer system must have an FM, MSF and an FP. Versions of these modules have been implemented and installed on the 360/91, H6180 (MULTICS) and PDP-10/Tenex.

- <u>Software Tools</u>. Tools are software systems which are available at a TBH and operate in an environment which recognizes the NSW. This environment is provided by FM on the tool's host. From the tool's viewpoint, FM acts as an extension to the local operating system. Although tools may seem to be an integral part of NSW from the user's viewpoint, this is not the case. NSW simply provides a framework in which tools may be installed. NSW software contains only a few specialized tools itself and provides the interface to the rest.

The tools execute on a variety of computer systems. The user does not require expertise in the particular computer used nor must he be concerned with moving or converting files from one machine to another; the NSW takes care of tool access and invocation and file reference, storage, and transfer. The NSW functions as a resource managing network operating system.

The latest version of the NSW involves the use of tools on the IBM 360/91, PDP-10/Tenex, and the Honeywell 6180/MULTICS. The system has a Front End (FE), currently residing on a Tenex, a single WM, (residing on the same or a different Tenex ; and three tool bearing hosts located at UCLA/CCN (360/91), MIT (H6180, and ISI (PDP-10) at Marina del Rey, CA. Geographical location of the FE or the WM is not important. The system runs equally well using an FE or a WM on a Tenex in Cambridge, MA., as contrasted with a Tenex machine at ISI, Marina del Rey, CA., or SRI, Menlo Park, CA.

The RADC interface to ARPANET is a Terminal Interface Message Processor (TIP). This hardware unit, based around the Honeywell 316 processor, will accommodate two computers and 63 terminals. Currently, the H6180 running MULTICS is connected to one computer port on the RADC TIP. Both the associative processor and the On-Line Pattern Analysis and Recognition System Software are assessible through network front-end processor software. TIP supports the NSW Front End package. Nearly half of the 63 terminal ports are in use, being connected to such devices as Execuports, Texas Instruments Silent 700, IMLACS and Terminent 300s. TIP will support asynchronous terminals either directly or modem-connected through a standard EIA connector. TIP hardware will support asynchronous internally clocked speeds (baud rates) from 75-to-2400 baud for input and from 75-to-19,200 baud for output.

2.2.2 Site-Specific Debugging Software

The following capabilities are available at RADC and may contribute to the debugging process, although they are not considered to be debugging software:

- FORTRAN Code Auditor. This program analyzes the syntax of a FORTRAN program and audits each statement for conformance to FORTRAN programming standards and conventions appropriate to the Air Force software development environment. It does not modify code, but it tells the user where in the source program pre-defined standards and conventions have not been followed. It is an effective mechanism for enforcing standards and improving both verification and maintenance activities. The Code Auditor requires the program being audited to be free from FORTRAN syntax errors (it does not detect syntax errors). The Code Auditor provides the following information:

  - A source code listing with Code Auditor assigned card numbers and numeric codes representing standards. Both numbers are used as reference numbers in other reports.

  - A subroutine summary report giving a count of executable statements in the module and tabular listing of standards and conventions violations within the module. The total count of each violation and the card number where the violation occurred is also given.

  - A program summary report giving:

    1. The number of violations of each standard in the entire program and in each subroutine.

    2. The total number of errors, the total card count, and a performance index for the entire program and for each subroutine. The performance index is computed by the following equation:

$$\left(1 - \frac{\text{total number of errors}}{\text{total card count}}\right) \times 100$$

A number of reports are output analyzing the program's adherence to the use of structured programming constructs.

- STRUCTRAN-1. STRUCTRAN-1 is a pre-processor for a structured-programming language called DMATRAN. It translates a program written in DMATRAN into a FORTRAN program that is compatible with the ASA standard FORTRAN X3.9. A structured programming pre-processor such as STRUCTRAN-1 enriches FORTRAN, but eliminates the need to change existing FORTRAN language compilers. Programs written using structured-programming constructs are more reliable, and easier to debug and maintain. DMATRAN replaces FORTRAN control statements with the following five structured programming constructs:

  - IF...THEN...ELSE...END IF. This construct provides block structuring of conditionally executable sequences of statements.

  - DO WHILE...END WHILE. This construct permits iteration of a code segment while a specified condition remains true.

  - CASE OF...CASE...CASE ELSE...END CASE. This construct allows multiple choices for program action selection.

  - DO UNTIL...END UNTIL. This construct permits iteration until a specified condition becomes true.

  - BLOCK NAME...END BLOCK. This construct and corresponding INVOKE <name> statement provide a facility for top-down programming and internal subroutines.

  These statement forms can be intermixed with ordinary FORTRAN non-control statements in the text stream processed by the STRUCTRAN-1 preprocessor.

  A structured DMATRAN program using these constructs has highly visible form which reveals its intended function more readily than a FORTRAN program containing GOTO statements. Well-defined blocks of code are executed in a sequential, top-down manner. The possible sequences of blocks which can be executable are also well-defined.

- STRUCTRAN-2. STRUCTRAN-2 translates programs written in the FORTRAN V programming language into logically equivalent, structured DMATRAN programs. STRUCTRAN-2 replaces FORTRAN control statements with the DMATRAN structured-programming constructs described previously (the CASE statement is not used). The output of STRUCTRAN-2 can be examined manually for

the purpose of making simple improvements to the code and then the DMATRAN program is translated back to a FORTRAN program by STRUCTRAN-1. The output of STRUCTRAN-1 is then compiled by a standard ASA FORTRAN compiler. STRUCTRAN-2 can be useful in the maintenance of existing FORTRAN programs. The resulting DMATRAN program is easier to understand, modify and maintain than the original FORTRAN program.

- JOVIAL Automated Verification System (JAVS). JAVS is a program-path-execution analyzer that operates on successfully-compiled JOCIT JOVIAL (J3) source modules and produces a variety of reports to aid in testing those modules. It is used as a software verification tool to assure that JOVIAL programs have achieved a measurable level of exercise during execution testing. JAVS provides test execution coverage reports showing which modules, decision-to-decision (DD) paths, and statements of a program have been exercised.

  JAVS analyzes up to 250 invokable modules and an unlimited number of JOVIAL statements in a single process job. A module is a JOVIAL main program, CLOSE, or PROC. A START-TERM sequence is a unit of source text, containing one or more modules, which is separately compilable (for the JOCIT JOVIAL compiler).

  The role of JAVS, as a testing tool, is to assure that the software has achieved a measurable level of exercise. JAVS provides execution coverage reports showing which modules, DD-paths, and statements have been exercised. When test cases are input which achieve a high level of DD-path coverage and which match the requirements stated in a functional specification, the tester can be assured of comprehensive verification.

  The dynamic behavior of the program can be studied by requesting JAVS tracing reports. These traces show the invocations and returns of all modules executed during the test. At user option, the tracing can be performed at the DD-path level to determine the dynamic behavior of the program while it is processing the data. In addition, the user can trace "important" events, such as overlay link loading, by invoking one of the JAVS data collection routines. JAVS operates in batch mode only, and in a series of separately-submitted steps it provides the following facilities:

  - It produces various kinds of source program reformattings and analyzes the source to create tables for use in subsequent steps.

  - It identifies the source program's pathways between each of its conditional branches, (DD pathways), creates a version of the source program instrumented for tracing of flow and variable assignments, and produces detailed reports and graph-like lists indicating the degree to which various parts of the program have been tested.

- It prepares reports describing the dynamic relations among the various modules (including library modules) of a multiple-module system .

- From the JAVS library prepared by previous steps, and on execution of the subject program, it prepares additional reports counting a variety of post-mortem statistics on the degree of path execution (for one run or cumulative for all runs), and various other execution-time summaries.

- FORTRAN Automated Verification System (FAVS). FAVS is a software verification tool used to assure that FORTRAN and DMATRAN programs have achieved a measurable level of exercise during execution testing. FAVS provides test execution coverage reports showing which modules, decision-to-decision (DD) paths, and statements of a program that have been exercised. It is also useful for providing syntax analysis of FORTRAN programs compiled on a compiler with minimal checking features. The following set of FAVS commands are input with a source program that is to be verified.

  - LANGUAGE which specifies whether source input is FORTRAN or DMATRAN.

  - LIST which produces a source listing which shows the number of each statement, the levels of indentation, and the DD-paths.

  - SUMMARY which provides an analysis of statements (classified or declaration, executable, decision, or documentation), common blocks, and module (subroutine) dependencies.

  - DOCUMENT which generates a set of five different reports including invocation summaries for a module, a list of READ statements within a module, a symbol cross reference listing for all modules, and a common block symbol matrix for modules.

  - STATIC which causes the source code to be examined to detect errors not normally checked by compilers. These include: mixed mode expressions, improper subroutine calls, variables that may be used before being set, graphically unreachable code, and potential infinite loops.

  - INSTRUMENT which causes a set of probe statements to be inserted into a module. The instrumented form is compiled and used in test execution.

  - TESTBOUND which allows the user to insert special probes into the instrumented code which delineates test cases within the text execution.

  - INPUT/OUTPUT which provides a dynamic tracing of the values of the program variables.

- REACHING which allows the user to determine which DD-path must be executed for a particular statement to be reached.

- RESTRUCTURE which translates existing FORTRAN programs into the structured language DMATRAN.

A variety of coverage analyses can be generated from data collection during execution of a program containing one or more modules that have been instrumented by FAVS. During execution the number of times a DD-path has been traversed is recorded on a trace file. The following FAVS commands generate reports from the trace file:

- FORMODULES which specifies for which modules the reports are to be generated.

- SUMMARY which lists the number of DD-paths for a module traversed by a test case, as well as cumulative coverage by past test cases.

- NOTHIT which lists the number of DD-paths not tranversed for this test case and for all test cases and identifies them.

- DETAILED which presents a display showing the individual DD-path coverage for a test case run on an individual module, and similar information for all test cases run on the module.

2.2.3 Vendor-Supplied Debugging Software

The following debugging capabilities are available via the H6180 MULTICS:

- Debug. The Debug command is an interactive debugging aid that allows the user to look at, or modify, data or code. The concept of breakpoints is implemented and thus makes it possible for the user to gain control during program execution for whatever reason he may have. Symbolic references permit the user to retreat from the machine oriented debugging techniques in common use and to refer to variables of interest directly by name. The Debug command provides the user with the following functions:
  - Look at data or code
  - Modify data or code
  - Set a break
  - Perform (possibly nonlocal) transfers
  - Call procedures
  - Trace the stack being used
  - Look at procedure arguments
  - Control and coordinate breakpoints
  - Continue execution after a breakpoint fault
  - Change the stack reference frame
  - Print machine registers
- Dump. The Dump command prints selected portions of a segment in octal format. It will print out four or eight words per line and can be instructed to print out an edited version of the ASCII representation.
- Page Trace. The Page Trace command prints a recent history of page faults and other system events.

- Probe. The Probe command provides symbolic, interactive debugging facilities for programs compiled with PL/1, FORTRAN, or COBOL. It provides the user with the following functions:

  - Set a breakpoint after a statement
  - Set a breakpoint before a statement
  - Call an external procedures
  - Return from probe
  - Execute a MULTICS command
  - Transfer to a statement
  - Stop the program
  - Execute commands if condition is true
  - Assign a value to a variable
  - Turn brief message mode on or off
  - Stop a program once
  - Examine a specified statement or locate a string in the program
  - Return to command level
  - Delete one or more breakpoints
  - Display source statements
  - Trace the stack
  - Display information about breakpoints
  - Advance one statement and halt
  - Display the attributes of a variable
  - Examine the block specified
  - Display the value of a variable
  - Display the value of probe pointers
  - Execute commands while condition is true

- Profile. The Profile command prints information about the execution of each statement in PL/1 or FORTRAN programs.

- Trace. The Trace command monitors all calls to a specified set of external procedures.

- Trace Stack. The Trace Stack command prints a detailed explanation of the current process's stack history in reverse order (most recent frame first). For each stack frame, all available information about the procedure which establishes the frame (including, if possible, the source statement last executed), the arguments to that (the owning) procedure, and the condition handlers established in the frame are printed. Track Stack is most useful after a fault or other error condition. If the command is invoked after such an error, the machine registers at the time of the fault are also printed, as well as an explanation of the fault and the source line in which it occured if possible.

The following debugging capabilities are available via the H6180 GCOS operating system:

- Conversational Debug Routine (RBUG). RBUG is a conversational debugging tool for batch programs using a teletype terminal as the conversational device. Various functions are available to RBUG to aid the programmer to inspect and modify program instructions, registers, and data parameters while testing the program in the execution environment. RBUG is an effective programmer tool for debugging and testing programs under development when operating from a remote terminal. By using RBUG for program testing, costly batch turn-around time is greatly reduced. RBUG provides the user with the following functions:

  - Print contents of location in ASCII or BCD.

  - Print location in double precision, complex, or decimal integers.

  - Print logical representation of location (T or F).

  - Print octal contents of location.

  - Print effective address.

  - Print registers.

  - Insert Breakpoint at location.

  - Remove Breakpoint at location.

- Modify a register.

- Path contents of location.

- Terminate program (abort).

- Terminate program (normal).

- Terminal Debug Subroutine (TDS). TDS is used to checkout and test a subsystem. It allows the user to gain control from a terminal at selected locations in the subsystem for inspecting patching areas of the subsystems. It also allows the user to run segments of the subsystem for testing purposes. TDS provides the user with the following capabilities:

  - Establish breakpoint at a given location. Breakpoint can be established to execute conditionally or unconditionally.

  - Delete breakpoint at a given location.

  - Modify contents of a register.

  - Set or reset offset values.

  - Patch or replace contents of a location.

  - Return to subsystem from breakpoint.

  - Return to specified locations in subsystem from breakpoint.

  - Snap or display memory.

  - Display registers.

- Time Sharing Debug TRACE Package. The command language includes the TDS language commands. The debugging capabilities are upgraded to include additional commands not found in RBUG and TDS. It also contains a TRACE mechanism making it a highly useful debugging aid. Some of the features include: accepting multiple commands; allowing continguous patching; collecting and displaying data as it steps through the program allowing the user to invoke a selected subsystem; and conducting a search through memory for a selected data pattern. The Time Sharing Debug TRACE package can be useful on highly complex programs used to debug, test, and evaluate a H6000 program. It is superior to RBUG and TDS because it has more debugging options. It uses more core space (6K) where RBUG and TDS use (2K). The DEBUG commands provide the following functions:

- Abort
- Establish breakpoints
- Enable control via break key
- Call subsystem
- Delete breakpoint
- Decimal to octal conversion
- Execute instruction
- Final data patterns in memory
- Local Symdef
- Modify registers
- Octal-to-decimal conversion
- Patch memory
- Resume execute
- Snap memory
- Save current program state
- Restore program state from last save
- Terminate execution and return to location where TRACE package was invoked
- Display Registers

The following TRACE options are available:

- Transfer Trace. Displays every instruction resulting in a transfer of a control within a specified tracing region.
- Operation Code Trace. Displays every occurance of one or more selected operation code within a specified tracing region.
- Modifier Trace. Similar to Operation Code Trace except modifiers will be traced.

- Use Trace. Displays every instruction which uses or references one or more specified registers.

- Change Trace. Similar to Use Trace except, only the instruction that changes the specified registers or memory are displayed.

- Full Trace. Displays all instructions within a specified region.

- Map Trace. Displays a map report showing partitioned segments of core and the number of instructions which were executed in each segment. Used for measuring a programs efficiency.

- Address Modification Trace. Displays every instruction using a specified type of address modification.

- Subroutine Trace. Displays all entries to and exists from up to 10 specified subroutines. The contents of the arguments and registers are shown upon entry and exit.

- Own Code Trace. Allows the user to design his own trace routine.

● Trace Package. This Trace Package is a debugging and information gathering routine which provides information dynamically on the program as it is executed. The Trace Package is useful for finding certain types of program errors quickly. It can also be a valuable tool in redesigning a program or for training since it can show the step-by-step operations of execution. There are many types of trace capabilities in this Trace Package, including a type which allows the user to design his own trace routine. The types of traces available are listed below.

  - Transfer Trace. Traces transfers of control. Conditional transfers are only traced when they transfer control.

  - Change Trace. Every instruction which actually changes a location or register is traced.

  - Use Trace. A location or register being used can be traced.

  - Operation Code Trace. Execution of a specified operation code is traced.

  - Full Trace. Traces every instruction executed including indirect modification changes.

- <u>Subroutine Trace</u>. Traces all entries and exits to subroutine. Displays register and number of entries.

- <u>Map Trace</u>. Provides a map report showing the frequency of instruction (location) being executed.

- <u>Own Code Trace</u>. Allows the user to supply his own code which is performed before the execution of each instruction.

● <u>Editor Subsystem</u>. The Time-Sharing Text Editor is an efficient tool for building programs and modifying programs or data files. When terminals are available, it is a highly desirable method to build and modify files using the Time-Sharing system rather than using the conventional method for inputing data via the card reader. Working within the confines of the Time-Sharing system, the EDITOR subsystem permits the user to perform functions on a text file. These functions include:

- Building a file

- Appending to an existing file

- Edit a text file by additions, deletions and corrections

Text files can be operated on from a terminal keyboard or from paper tape output.

● <u>FDUMP Subsystems</u>. The FDUMP subsystem operates under the Time-Sharing System and is used to inspect and modify mass storage files in the permanent file system of GCOS. FDUMP will display a file in 320-word blocks and permit a word or words in the block as specified to be modified.

2.3 TELEMETRY INTEGRATED PROCESSING SYSTEM (TIPS)

The Telemetry Integrated Processing System (TIPS) is a Space and Missile Test Center (SAMTEC) project, located at Vandenburg Air Force Base. Its purpose is to support space and missile pre-launch, launch, and post-launch requirements through the acquisition, processing, and distribution of telemetry data. Specifically, telemetry data input to the system results from support of space and missile launches, aircraft fly-bys, and orbiting satellites.

The TIPS computers constitute a distributed processor system composed of multiple Systems Engineering Laboratories (SEL) 32/55s and a Control Data Corporation (CDC) CYBER 173 computer system. The TIPS Distributed Operating System (DOS) comprises the operating system for the network of SEL computers and is an extension of SEL's Real-Time Monitor (RTM). The CYBER 173 Network Operating System (NOS) is extended via NOS extensions to support communication with interfacing SEL computers. DOS and NOS extensions each provide operating system features through which the TIPS distributed computers can interact.

The following discussion describes the TIPS debugging effort in terms of:

- Hardware/software
- Site-specific debugging software
- Vendor-supplied debugging tools
- Testing and debugging methodology
- Debugging methods actually used

2.3.1 Hardware/Software Descriptions

The TIPS hardware configuration is shown in Figure 3. The hardware consists of six major processors whose functions are as follows:

- Near Real-Time Batch Processor (NBP). This processor provides batch users with necessary data from telemetry input in near-real-time. In non-real-time, the NBP provides the system's large scale computation capability, interactive time-share user support, scheduling runs, and telemetry compiler execution.

- Telemetry Pre-Processor (TPP). This processor processes telemetry data according to tables/programs supplied by the telemetry compiler and outputs all data to the Mass Storage Controller (MSC).

- Mass Storage Controller (MSC). This processor provides interfaces with the TPPs for history recording as well as the NBP interface for mass-storage equipment. The MSC in turn, provides the NPB interface to TPP-stored data as well as the TPP/NBP interface with the Configuration and Interface Controller (CIC). History data is

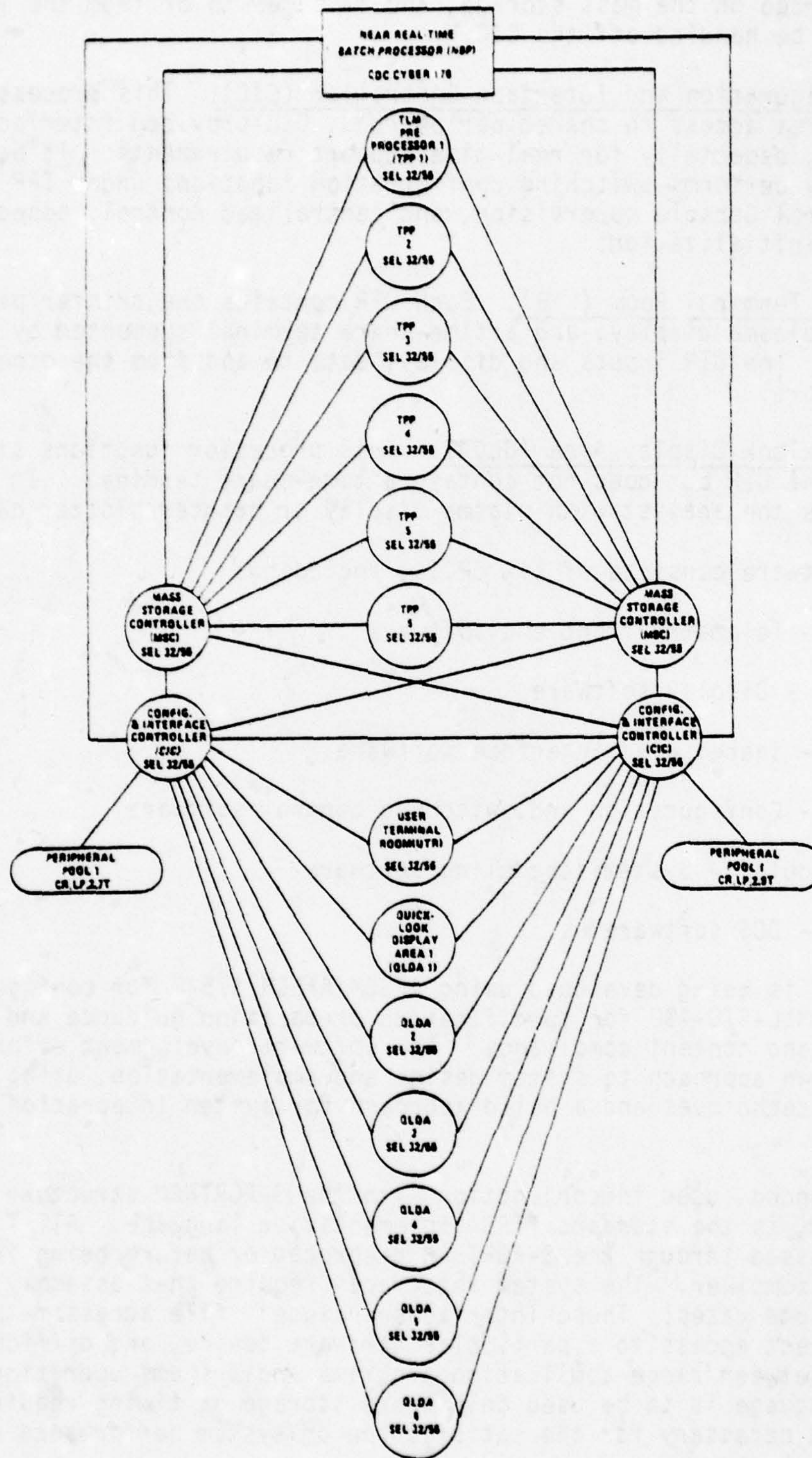


Figure 3. TIPS Hardware Configuration

recorded on the mass storage, and messages to or from the TPP will be handled off the CIC.

- Configuration and Interface Controller (CIC). This processor assures access to shared peripherals, CIC-provided interfaces, etc., especially for real-time support requirements. It basically performs switching configuration functions under TPP Control Console supervision, and centralized control, scheduling, and initialization.

- User Terminal Room (UTR). Each UTR contains one printer/plotter, one plasma display, and a time-share terminal supported by the NBP. The UTR inputs and displays data to and from the other processors.

- Quicklook Display Area (QLDA). This processor functions similar to the UTR but does not contain a time-share terminal. It provides the analyst with plasma display or printer/plotter hardcopy.

The TIPS software consists of six CPCIs, including:

- TPP - Telemetry front end software

- QLDA - Display software

- MSC - Shared disc interface software

- CIC - Configuration and switching control software

- Schedules - System scheduling software

- DOS - DOS software

The software is being developed using AFSCM/AFLCM 375-7 for configuration management, MIL-STD-490 for specification preparation guidance and MIL-STD-483 for formats and content compliance. The software development effort is employing a top-down approach to system design and implementation, using structured programming techniques and a build approach for system integration (see Paragraph 2.3.4).

FORTRAN Extended, used in conjunction with the S-FORTRAN structured FORTRAN preprocessor, is the standard TIPS implementation language. All TIPS programs are passed through the S-FORTRAN preprocessor before being input to the FORTRAN compiler. The system interfaces require that assembly language be used in some cases. These interfaces include: file access methods, specific direct access to a particular hardware device, and difficult system interfaces between range application programs and systems operating software. Assembly language is to be used only where storage or timing requirements are critical and necessary for the satisfaction of system performance requirements.

2.3.2 Site-Specific Debugging Software

At the time of the literature survey, the TIPS project had just completed CPC detailed design activities. In addition, the project had developed very few of its own tools. In a previous internal survey, the programming personnel determined which tools were available and which tools should be developed. The following summarizes the response to the survey:

- SEL Debugging Tools Available:
  - Breakpoint Dump Program. This program provides a command terminal display of general registers and designated dump addresses in hexidecimal when a designated breakpoint in an active task is reached.
- SEL Debugging Tool Capability Desired:
  - Formatted Print of ICD Data. This candidate tool would format dumps of inter-CPCI data consistant with ICD data descriptions such that:
    1. Key interface data is recorded whenever it is read, written or moved.
    2. Data recorded during development (Phase I & Phase II Component Test) can be identically recorded during system integration (Phase III Component Test) and Configuration Item Test.
    3. Data to be listed can be selected by the analyst either through the use of selective recording, the use of reduction routines, or both.
    4. Recording hooks be inserted in such a way that they can be disabled or removed with minimal disruption of the software being tested.
  - Formatted Print of Non-ICD System Data. This candidate tool is essentially the same as the formatted ICD data tool, but the output is critical data which is not defined in the ICD.
- CYBER Debugging Tools Available:
  - FORTRAN COMMON Data Cross-Reference Program. This program provides a set/use cross-reference of all data variables and constants declared via NAMED COMMON statements. Likewise a cross reference of all routines and subroutines on the Program Library is provided. This is useful for interface problem debugging.

- CYBER Debugging Tools Desired:
  - Formatted Print of ICD Data. This candidate tool is the same as that described for the SEL software.
  - Formatted Print of Non-ICD System Data. This candidate tool is the same as that described for the SEL software.

2.3.3. Vendor Supplied Debugging Tools

A number of debugging features are available in the vendor supplied software for the SEL and CYBER computer systems.

- SEL Debugging Tools.
  - Real-Time Monitor (RTM) System Debugging Features. The debugging capabilities provided by SEL RTM are described below in terms of the operator communication, monitor services, debug, and media conversion processors of RTM.

    1. Operator Communication Processor Debugging Features. The Operator Communication processor provides the interface between the computer operator and the RTM by processing operator input commands which direct RTM to perform specific control and inquiry functions. An extensive set of input command verbs are accepted from the console keyboard, allowing the operator to exercise complete system/control from the console. The commands pertaining to debugging are:

COMMAND	DESCRIPTION OF COMMAND
SNAP	The input command, SNAP, dumps the word locations specified by the starting and ending address to the console TTY.
DUMP	The input command, DUMP, outputs to an allocated output file the word locations specified by the starting and ending addresses. Listed output is in side-by-side ASCII-coded hexadecimal with ASCII format.
SEARCH	The input command, SEARCH, causes core memory to be searched within the specified addresses for the specified value under control of the specified mask. A "logical AND" operation is performed between the memory word and the mask word. The result is compared to the value, and if equal, the memory address and contents are printed on the console printer.

COMMAND	DESCRIPTION OF COMMAND
FILL	The input command, FILL, fills the specified memory area with the specified value.
MODIFY	The input command MODIFY permits any core memory word to be reset by the specified value under control of the specified mask. A "logical AND" operation is performed between the specified memory word and the mask word, followed by a "logical OR" operation performed between the result of the "logical AND" and the specified value. This result is stored in the specified memory word.
EXAMINE	The input command, EXAMINE, outputs the contents of the specified memory word to the console TTY.

2. Monitor Services Processor Debugging Features. The RTM provides an extensive set of resident, reentrant services accessible to the user by macro calls, Call Monitor (CALM) instructions, and FORTRAN-callable subroutines. The only macro applicable to debugging is the core dump request. This service provides a dump of the caller's program status word (PSW), general purpose registers, and specified memory limits. The output is to a file in side-by-side ASCII-coded hexadecimal with ASCII format, with the PSW and registers preceeding the specified memory limits. The PSW and registers are extracted from the first level of register push down of the calling task. Optionally, register 5 may specify the address of a nine word block containing the registers 0 through 7 and the PSW to be dumped, respectively. Any task may request a core dump.

3. Debug Processor Debugging Features. The RTM provides capabilities for debugging programs in the background under control of the DEBUG Processor. DEBUG operates as a non-privileged, batch background task, fully protected from degrading the operation of the foreground or the system. The control directives are:

DIRECTIVE	DESCRIPTION OF DIRECTIVE
DISPLAY	The DISPLAY directive results in an immediate dump of the user's PSW, general purpose registers, and specified memory. The dump is hexadecimal and is output to an output file.

DIRECTIVE	DESCRIPTION OF DIRECTIVE
FILL	The FILL directive results in all memory word locations between, and including the specified limits being set to the specified value. The value to which memory is modified is absolute, that is, DEBUG will not assume that the value contains an address which must be biased by the address of the first location available to the user. This directive is useful for zeroing blocks of memory or initializing tables or buffers to a constant value.
MODIFY	The MODIFY directive results in the modification of a general purpose register or series of registers or a memory word or series of memory words with the specified value(s). Modification occurs to successive memory locations or registers until the specification list is exhausted.
RESET	The RESET directive results in the elimination of all prior requests for program breakpoints, snapshot dumps, and trace initiations and terminations. If trace is active, it is terminated. The Linking Loader symbol table is cleared, and the default load origin (address at which the next module will load if an address is not specified) is reset to the first location available to the user. Any programs which may have been loaded under the control of DEBUG remain intact along with the user's registers and PSW.
SNAPSHOT	The SNAPSHOT directive results in the user's instruction at the specified memory location being replaced by a DEBUG Call Monitor (CALM) instruction so that when that location is executed, the user's program is interrupted for the snapshot dump. The dump is hexadecimal and includes the user's PSW, registers, and specified memory locations. Upon completion of the dump to an output file, the instruction that was replaced by the CALM is executed and control is returned to the user program for continuation of normal execution.
START	The START directive results in control being transferred to the user program at the specified location. The address specification on this directive is optional, and if it is not specified, the last transfer address encountered by the Linking Loader is assumed. If the user program was interrupted as a result of a STOP directive and no intervening program loading with a transfer address has occurred,

DIRECTIVE	DESCRIPTION OF DIRECTIVE
START (cont'd)	the user program will resume execution at the point of the interruption. In other words, the priority specifying the location at which the user program will start execution is as follows: • Address specified on the directive. • Transfer address encountered by the Linking Loader • Location at which the program was interrupted as a result of a STOP directive.
STOP	The STOP directive results in the user's instruction(s) at the specified locations(s) being replaced by DEBUG Call Monitor (CALM) instructions which will result in interruption of the user program when any of the specified locations are executed. When the interruption occurs, control is dispatched to DEBUG for directive processing. The START directive will allow execution of the replaced instruction and continuation of processing.
TRACE	This directive results in the user's instructions at the specified locations being replaced by DEBUG Call Monitor (CALM) instructions which will result in initiation or termination of program tracing when the specified locations are executed. The program trace provides a comprehensive listing of all activity occurring within the program segment being traced. The instruction being traced is output along with the instruction mnemonic, all register and memory locations referenced and their contents before execution; all registers and memory locations altered by the execution of the instruction and their values after execution; and the resultant condition codes. Additionally, some instructions are flagged with an asterisk (*) indicating that they have not been executed because they belong to the privileged instruction set, and some instructions are flagged with a question mark (?) indicating a suspicious condition has been encountered, such as a BIB on a non-negative register, address type modified as a result of indexing or indirect addressing, or Load File or Store File on an improper boundary. All values output to the trace are hexadecimal with the exeception of shift counters which are decimal, and condition codes which are binary.

4. Media Conversion Processor. The Media Conversion Processor provides media-to-media conversion, media copying, and media verification capabilities. Functions ranging from file duplication to merging multiple media inputs into single or multiple media outputs are provided by the Media Conversion Processor. The only debugging capability is invoked by the DUMP statement. The DUMP statement causes the file specified by file code 1 to be input record-by-record, converted to ASCII-coded hexadecimal, and output to the file specified by file code 2. The dump is terminated when an end-of-life is encountered on either file, or when the optionally specified number of records have been dumped. File code 2 must be assigned to either the line printer or an output file.

- Terminal Support System (TSS) Debugging Features. The SYSTEMS TSS is a multi-terminal, interactive processing system which operates in conjunction with the SYSTEMS RTM. Through TSS, the user at a terminal can edit text files, compose and initiate batch jobs, and control and monitor the execution of cataloged tasks being debugged. The debugging function of this system is called FORGROUND DEBUG. FORGOUND DEBUG provides facilities to control and monitor the on-line execution of cataloged tasks from terminals. It augments RTM batch debug facilities where program logic can be exercised off-line. Once the integrity of the logic reaches the desired level of confidence, the program may be cataloged as a task and further exercised through FORGROUND DEBUG.

  Tasks being debugged may be cataloged at any task priority level. Their execution are free-standing tasks, independent from FORGROUND DEBUG.

  The debugging commands, and their meaning, for this system are as follows:

COMMAND	DESCRIPTION OF COMMAND
ABORT	This command requests the abnormal termination (abort) of a specified task.
ACTIVATE	This command issues an activation request for the specified task. The task must have been cataloged at a foreground level.
CONTINUE	This command resumes the execution of the specified task which was suspended by the HOLD command or is currently trapped.
DISPLAY	This command requests the display of the contents of an area of core memory. If the area is task relative, the task must be in core memory.

COMMAND	DESCRIPTION OF COMMAND
DUMP	This command requests a listing of an area of core memory on a line printer. If an area is task relative, the task must be in core memory.
EXIT	This command terminates the debug function. All traps set by the using terminal are purged.
FILL	This command modifies an area of core memory to equal a specified value. If the area is task relative, the task must be in core memory.
FREE	This command returns a task protected via the PROTECT command to a nonprotected status.
HOLD	This command suspends execution of the specified tasks. Execution of the tasks can be resumed with the CONTINUE command.
MODIFY	This command changes a word in core memory to equal a specified value. If the address of the word is task relative, the task must be in core memory.
NAME	This command sets the default "task name" for other commands to the task name specified.
PROTECT	This command restricts referencing the specified task from terminals other than the one protecting the task. The task is returned to a nonprotected status by the FREE command or by the EXIT command.
PSW	This command alters the current execution address of a task by changing the contents of the task's PSW. Only the address portion of the PSW is changed. The privileged bit and condition code bits in the PSW are unchanged. The specified task must be suspended by a HOLD command or by an activation or internal trap. The task cannot be in a termination trap.
PURGE	This command deletes traps set from this terminal via the TRAP command.
RO-7	This command sets the contents of a general purpose register to the value specified. The specified task must be in core memory and suspended by a HOLD command or by an activation or internal trap. The task cannot be in a termination trap.

COMMAND	DESCRIPTION OF COMMAND
SEARCH	This command initiates a search of a specified area of core memory under control of a mask for words containing a given value.
STATUS	This command provides a display of the last batch job processed and all active tasks except RTM core resident tasks.
TASK	This command provides a display of a specified active tasks's status, registers, and traps.
TRAP	This command sets traps for the selective suspension of a tasks' execution. There are three kinds of traps:
	Activation Traps: This type results in the suspension of a task each time the task is placed in execution in response to an ACTIVATE request.
	Termination Traps: This type results in the suspension of a task upon its termination.
	Internal Traps: This type results in the suspension of a task during its execution. One or more traps may be set at locations internal to a task via a common trap table previously set up by the user.
TRAPS	This command provides a display of all currently active traps set by this user and their status.

- User's Group Software Library Debugging Features. The following debugging programs are available to the user via the SEL software library:

PROGRAM	DESCRIPTION OF PROGRAM
COREDUMP	Stand alone routine to dump the contents of specified memory locations to the line printer.
DEBUGGER	This processor or subroutine gives the user control of his program at the console teletype or system input device, and offers a wide variety of debug commands.

The following commands are available:

AD Display current base and high memory

AR Perform arithmetic add, subtract, multiply, divide, shift left, shift right, logical AND, or logical OR.

PROGRAM	DESCRIPTION OF PROGRAMS
	BA Set base to absolute value
	BR Set base to relative value
	CC Change control input to console teletype
	CL Clear DEBUGGER
	CM Change memory
	CR Change register
	CT Continue execution and breakpoint
	DM Display memory
	DU Dump memory to LO device
	EX Set breakpoint and execute
	GO Start execution
	IF Set IF request providing for conditional execution of DEBUGGER commands
	OV Load overlay
	PS Display user PSWR
	SI Change control input to SID
	SN Set snapshot request with multiple dump ranges and execution count
	TE Terminate execution
DUMPMAP	This routine provides the on-line capability to binarily dump the disc allocation maps.
DUMPSMD	This routine provides the on-line capability to hexadecimally dump the System Master Directory file.
FDP (FORTRAN DEBUG)	FDP acts as a preprocessor to provide a complete or selective trace of FORTRAN programs. Clean, compiled FORTRAN is scanned and calls to print routines are added to the code. At run time the trace is output via logical stream DO. Optionally at run time, tracing can be selectively started and stopped at locations within the program. Selected variables may be omitted or included in the trace.

PROGRAM	DESCRIPTION OF PROGRAMS
	Commands are optionally inserted at run time. The following commands can be entered:

COMMAND	DESCRIPTION OF COMMAND
STARTRACE	Begin tracing when label is reached.
STOPTRACE	Stop tracing when label is reached.
VARIABLE	Print only specified variables.
NOTVARIABLE	Print all variables and labels excent that specified.
CLEAR	Clear debug commands.
EXIT	Stop execution.
END	End of debug commands

FINDCALL	To find all locations in the resident RTM which call a specific system module.
LIST	This routine lists any blocked or unblocked file on disc to the line printer.
TIME	Provides timing information for job steps by printing the current system time as recorded by the interrupt counter to the operator's console.

- CYBER Debugging Tools. The following debugging features are provided by the CYBER systems software.

  - FORTRAN Extended Debugging Facility. The FORTRAN Extended Compiler for the CYBER is equipped with a comprehensive and time-saving debugging facility. This facility permits the FORTRAN programmer to debug a faulty program without assistance from core dumps or assembly language listings. With this debugging facility, the programmer debugs a program within the context of the FORTRAN language itself.

    The debugging facility permits the programmer to check for one or more types of possible errors. These checks are made by options placed on special cards which are either inserted in the program deck to be debugged or are placed externally to it. These cards can be compiled or ignored (treated as comments) via a control specification in the JOB deck. The following options can be specified.

OPTION	DESCRIPTION OF DEBUGGING CAPABILITY
ARRAYS	Checks subscript bounds on dimensioned variables.
TRACE	Checks transfers of control within a program.

OPTION	DESCRIPTION OF DEBUGGING CAPABILITY
STORES	Records changes of values of specified variables whenever they are encountered in arithmetic replacement statements.
FUNCS	Checks values returned by library functions and function subprograms.
CALLS	Checks calls to and returns from subroutines.
GOTOS	Checks the validity of statement labels which appear in assigned GOTO statements.
OFF	Turns off one or more of the above options after they have been turned on: effective only for those options activated in the interspersed mode.
NOGO	Suppresses execution if a fatal error occurs during compilation: in the absence of a NOGO card any program compiled with a debug control specification will execute until the point of a fatal error, if one is present.
DEBUG	Used only with internal and external packets and external debug files.
AREA	Used only in conjunction with the DEBUG card: the area statement allows regions within a program to be debugged on a selective basis.

- CYBER Network Operating System (NOS) Debugging Features. The reference manual for NOS indicates the only debugging aids available in this operating system are central memory dumps. The main portion of the dump is in octal form. Also displayed, is the exchange package for the user, a symbolic reference map to equate the machine code of the dump to a FORTRAN source listing, and a load map.

2.3.4 Testing and Debugging Methodology

The testing methodology for the TIPS project is based exclusively on code execution testing. The purpose of this testing is to validate that the requirements specified in the Development (Part I) Specifications are satisfied. There is no static code analysis testing involving code analysis tools in this project (e.g., path flow analyzers, equation generators, or proof-of-correctness tools).

The TIPS project has an independent test group called the Software Integration Group (SIG). This group is used for integration testing after the production of the Part I and Part II specifications. However, the group has no influence on the testability of the design via participation in the requirements or specifications analysis phases.

2.3.4.1 Software Integration

SIG has the responsibility for integrating and testing the software through a succession of builds*. The SIG is comprised of a small cadre of personnel and is temporarily supported in the integration process by the develooment programmer(s) responsible for a particular integration test procedure. Also assisting the SIG and forming an integral part of it are personnel assigned by the SAMTEC Project Office.

The Software Build Plan identifies and selects those functions and required CPCs which will be developed and integrated into sequential builds, until the final build contains the capabilities identified for the Initial Operational Capability (IOC). Each successive build will use the previous build as a building block and the new software test data base contains the CPCs released for the new build, as well as all the software released in previous builds. The software contained in each new build can then use system capabilities of previous builds and the total system is gradually integrated in a planned manner. Regression testing is an ongoing process in this activity and takes place as dead-end drivers, stubs, simulation code, etc., are removed, and as errors are found and corrected.

The following criteria have been used to establish a candidate set of functional capabilities to be incorporated in the first and successive builds.

- The function is in the critical path in terms of a dependency for continued system development.
- The function will minimize the need for construction of dead-end drivers and stubs.
- The function is important to validate system design.
- The function has a feasible completion date.
- The function will provide tangible visibility of progress.
- The function uses available hardware.

A separate build schedule has been developed and reflects in detail the functions and CPCs being incorporated in each successive build. Test planning and scheduling by SIG is based upon this schedule.

The CPCs comprising a particular construct are released to SIG as Phase II Component Testing (see Paragraph 2.3.4.2) is completed according to the established schedule for each build. It is at this point that the software is placed under internal configuration control. The test procedures and test reports are also maintained and controlled, and available for customer review.

*CPCs logically linked together to form constructs, which are then logically linked together to form builds.

Integration testing begins with the release of the first build and continues until the testing of the IOC build is complete. Each successive build is added to the previous build and integration testing continues using the total software incorporated in the test file. Regression testing is performed as required when new features are added to CPCs, design changes are incorporated, or errors are corrected. Regression testing could occur at any phase of component testing.

As successive builds are incorporated into the test file and meaningful strings of CPCs are linked together, two sets of test data are used to verify system operation. The first set of test data is a small amount of inputs representing data from a relatively simple vehicle. The primary purpose of this first set of test data is to provide a manageable amount of data to ensure gross system operation by verifying software-to-software and software-to-hardware interfaces.

The second set of test input data represents a more complex vehicle and is introduced to the integration process as system performance increases. This set of test data approaches the capability to demonstrate a total system operation prior to formal configuration item testing.

It is anticipated that a large percentage of system requirements will be verified during module testing; i.e., during individual CPC testing and during integration testing. However, verification of system requirements during component testing does not necessarily preclude formal verification of selected requirements during configuration item and subsystem/system testing.

2.3.4.2 Test Levels

Overall testing of the TIPS requirements will be performed in three levels of testing as defined below. The levels of testing also represent the sequence in which testing will take place.

- Level 1 - Component Testing. Encompasses Computer Program Test and Evaluation (CPT&E) and Preliminary Qualification Test (PQT).
- Level 2 - Configuration Item Testing. Encompasses Formal Qualification Test (FQT).
- Level 3 - Subsystem/System Testing. Formerly Category II Testing.

Level 1, or component testing, is the only level applicable to the debugging study because it includes the development integration phase. There are three phases of component testing. These phases of testing represent the stages in the developmental process from initial coding of CPCs through total system integration. Component testing is internal to the development contractor with no formal customer participation. However, all testing procedures, test reports and related documentation will be available for review. The three phases of component testing are:

- Phase I. Phase I component testing represents the testing of small sections of code (a subset of a CPC) to provide confidence and demonstrate functional capability. The amount of testing required is the responsibility of the programmer and will depend upon the size and complexity of the CPC. No written procedures are required.

- Phase II. Phase II component testing represents the functional testing of one or more CPC's (CONSTRUCTS). Written test procedures and test reports following a contractor standard format are required. Although these tests are the responsibility of the programmer, the procedures and reports will be controlled and maintained by SIG.

- Phase III. Phase III component testing represents that testing accomplished during the system integration process and is the test phase pertinent to this study. It is the functional testing of two or more CPCs beginning with the release of the initial build until final integration testing is accomplished with the release of the IOC build. Written test procedures and reports following a contractor standard format are required. The software is under internal configuration control. This requires that any errors discovered will be documented via a Discrepancy Report (DR) and corrections released via a Corrections Report (CR). DR/CR procedures are to be published as a TIPS Operational/Administrative Directive (TOAD) and will be applicable to the entire TIPS project. Test procedures and test reports are also controlled by SIG. This phase of testing is the responsibility of SIG with support from the developmental programmers as required. Individual CPCI requirements will be verified as a result of component testing of the several builds culminating with the testing against the IOC build.

2.3.4.3 Phase III Testing and Associated Debugging Methods

When Phase II testing is run to satisfaction and the code for a construct is released to SIG, the test procedures are rerun as part of Phase III testing by the test group (with help from the programmer as required). This is done for two reasons. First it verifies that the code under configuration control is indeed the same code used during Phase II component testing and that no errors were introduced during the process of obtaining the code and placing it under configuration control. Second, the files that the SIG will use for testing are the total accumulated TIPS software. Phase III component testing is intended to reveal any interface problems with constructs/builds at the earliest possible time and demonstrate the compatability of the system. Two types of errors are expected as a result of integration testing. First, relatively simple program logic errors will be found because different inputs are used and more interfaces are exercised, or the system is operated in a different manner or sequence. The second type of error is more complex and could involve CPCI interfaces, software/hardware interfaces, throughput, design problems, etc.

Testing and debugging within TIPS is almost entirely interactive (i.e., a test is executed via an interactive display/input consoles and if the cause is isolated and the correction is found, the source code is modified, recompiled, and the test is rerun to verify the correction). For problems with complex corrections, the programmer can discontinue operation on the console, devise the correction separately, return to the console, and continue interactive testing/debugging.

The actual isolation of problem causes through the accumulation of symptoms, elimination of factors, and determination of what really should occur, is entirely at the discretion and imagination of the individual debugger in the TIPS system. No precise or imprecise definitions of when one tool should be used as opposed to another has been formulated.

2.3.5 Debugging Methods Actually Used

Interviews with the programmers of the TIPS project revealed that relatively few of the debugging tools discussed in Paragraphs 2.3.2 and 2.3.3 will actually be employed during the Phase III period. The following tools are likely to be used:

- SEL Breakpoint Dump Program
- Formatted Print Of ICD Data (candidate tool for CYBER and SEL)
- Formatted Print Of Critical Non-ICD Data (candidate tool for CYBER and SEL)
- FORTRAN COMMON Data Cross Reference Program
- FORTRAN COMMON Data List
- FORTRAN Extended Debugging Facility

2.4 SPACE COMPUTATION CENTER (427M)

The Space Computation Center (SCC) is a joint USAF/SDC/FACC effort to develop a catalog of earth orbiting objects and maintain current orbital characteristics. The SCC software executes on an HIS 6080 computer. There are currently over 3 million lines of source code in support of this objective. The system is intended to replace an existing Philco 2000-based system. The 427M software development effort has been planned in incremental deliveries of increasing capability. The first four releases of the system have been delivered. The software has been implemented in a mix of FORTRAN, JOVIAL, and GMAP assembly language.

The SCC software development is divided into three major areas: application programs, GCOS extensions, and the man-machine display interface. The application programs are almost entirely written in either FORTRAN or JOVIAL. The size of the system is such that substantial extensions to GCOS have been required. A sub-executive, called Space Computation Center Executive (SCCEX) has been implemented to provide this support. SCCEX is implemented in the GMAP assembly language. The man-machine display interface software resides in Data General NOVA minicomputers and is coded entirely in assembly language.

2.4.1 Hardware/Software Description

The SCC is implemented as a distributed system consisting of a dual processor HIS 6080 and a set of Data General NOVA minicomputers which support interactive displays. Figure 4 depicts a simplified view of the system architecture.

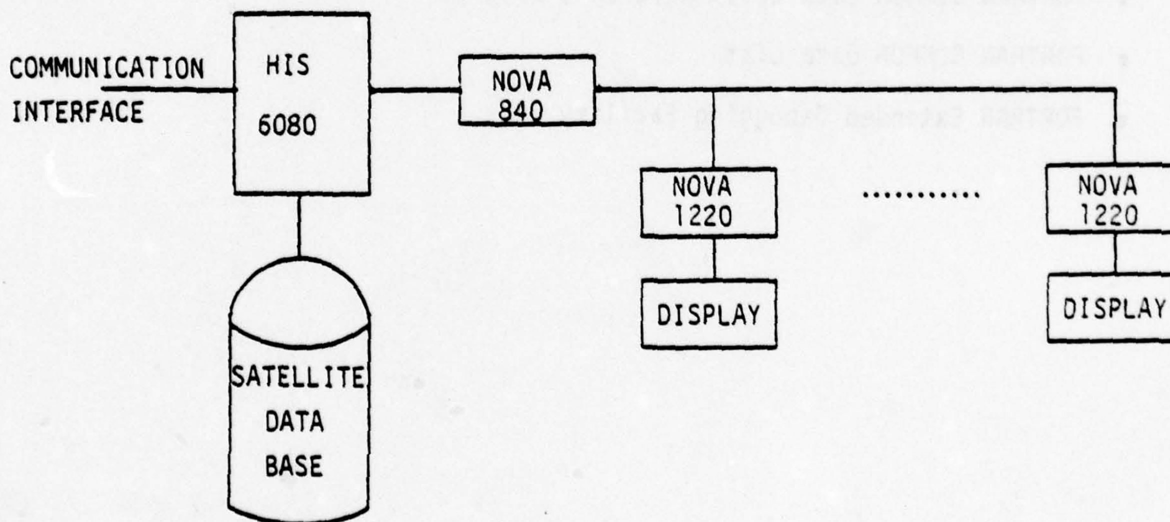


Figure 4. Space Computation Center System Architecture

The hardware processors consist of the following:

- Honeywell Information Systems 6080. The HIS 6080 utilized by the SCC is a standard Worldwide Military Command Control System (WWMCCS) - configured dual processor computer operating under the GCOS operating system. The HIS 6080 is configured with the maximum memory complement of 512K 36-bit words. The orbital data base resides on 10 HIS 227 disk packs.

- Data General NOVA 840. The NOVA 840. The NOVA 840 is a standard Data General processor configured with 16K 16-bit words. It is used as a communications interface between the orbital applications and the man-machine display subsystem. In its operational configuration, the NOVA 840 is operated without the vendor-supplied operating system.

- Data General NOVA 1220. The NOVA 1220 is a standard Data General processor configured with 8K 16-bit words. It is used as a display driver for the Raytheon display consoles. It also is operated without the operating system.

2.4.2 Site Specific Debugging Software

The SCC project has evolved a set of software testing and debugging tools which fall into two categories, (1) extensions to vendor-supplied capabilities and (2) environment simulation/data capture tools, as follows:

- Extensions to vendor capabilities
  - A formatted dump capability which allows the display of system control blocks local to the SCC implemented GCOS extensions.
  - A cross-reference capability to display the use of program-defined symbols between separately assembled routines. This processor displays the symbol, the routines in which it is referred, the source line number of the reference, and the text of the source line.
  - A set of conventions and standard JCL adapted to the use of GCOS file-system capabilities which allow management of multiple releases by maintaining separate libraries of modules (source code, object code, and alterations) for each release. These conventions can be viewed as implementing a manual version of a Programming Support Library.

- Environment Simulation/Data Capture Tools

  - A software implemented virtual display console which allows development and debugging of application programs in the absence of the minicomputer-based, man-machine interface display system. This test driver allows testing of application programs from a time sharing station. This system allows development to continue when the display subsystem is unavailable due to scheduling constraints or, earlier in the project, while the display subsystem is under development.

  - A software implemented data capture system developed to record message traffic in both directions between the HIS 6080 and the display subsystem. All data is captured on the HIS 6080 side of the interface.

2.4.3 Vendor Supplied Debugging Tools

The following vendor-supplied debugging tools were available for use on the SCC project:

- HIS 6080. The WWMCCS-configured HIS 6080 is common to several of the sites surveyed. A description of Honeywell supported debug tools is included in Paragraph 2.2.1.1.

- Data General NOVA 840 and NOVA 1220. No capabilities beyond an octal system dump were employed.

2.4.4 Testing and Debugging Methodology

The SCC identifies and tracks software errors in a conventional fashion in which DRs are prepared by a testing organization and software corrections to the system are tracked by a configuration management process. The module-level testing is performed by programmers and is not subject to configuration management. Integration testing is performed by an organizationally distinct test group which reports at the same level as the development group managers. The testing approach used is based upon demonstration of functional capabilities and is not readily describable as either top down or bottom up developments. No data reduction or data analysis tools are available.

The actual debugging methodology exhibits a primary reliance on GCOS and locally developed dumps or programmer-inserted code which prints information about program data and control flow. Some use of GCOS-supported interactive debugging tools was reported for application programs in the early development phases, but the size of the total system prevents inclusion of the GCOS time-sharing system in the system generation for integration test runs. As has been the norm in other site visits, no widespread methodology of debugging was evident. Debugging style is apparently a very individual matter. The highest level of interest was exhibited in the development of more powerful interactive debugging capabilities.

2.5 SATELLITE CONTROL FACILITY

The Air Force Satellite Control Facility (AFSCF) is a SAMSO funded project. It is composed of the Satellite Test Center (STC) in Sunnyvale, California, and several remote tracking stations (RTSs) located around the world. The AFSCF's purpose is to support unmanned satellites during the on-orbit and reentry phases of a mission. This support consists basically of transmitting data to a satellite (command), receiving and processing data (telemetry) from a satellite, and keeping precise track of the location of a satellite (tracking). Intercomputer communication between the STC and external agencies also occures.

The SCF computers comprise a varied range of systems, including:

- Seven batch-processing, non-real-time, large-scale flight support computers (CDC 3800s) located at the STC.
- Twelve near-real-time buffer computers (Varian 73 mini-computers) which emulate a previously used buffer computer (CDC 160A) at the STC.
- Two real-time multiprocessing computers (Univac 1230s) located at each RTS.

Figure 5 presents a generalized view of the AFSCF total computer system. The system depicted, together with the general purpose real-time and non-real-time systems support software, is shared by different satellite projects. The individual projects (or users) have their own unique software to produce commands and evaluate the telemetry from their satellites. The satellite projects have their own software development contractors called Computer Program Associate Contractors (CPACs) to develop and maintain this unique software. The AFSCF computer-system general-purpose support software is developed by a separate group of CPACs. To insure that all of the software works together, the AFSCF employs a unique contractor, called the Computer Program Integration Contractor (CPIC), to integrate and test all new or changed software being used on the AFSCF system. The other unique contractor for the AFSCF is the General System Engineering/Technical Director (GSE/TD) Contractor (Aerospace Corporation) whose function is to specify the development requirements for the new or modified software for the system. Figure 6 summarizes the responsibilities of the AFSCF software community.

2.5.1 Hardware/Software Description

The AFSCF uses three basic computer systems, the Flight Support Computer System (FSCS), a buffer computer system, and the RTS computer system.

The FSCS consists of the CDC 3800 computer, the System IIB operating system, and the satellite project-specific application software. Only the computer and the operating system are described here since the application software's characteristics vary considerably and contribute little to the background for the debugging study. A description of the FSCS follows:

- CDC 3800. The CDC 3800 systems consist of a central computer (computational module, input/output section, 65K magnetic core, and a console), eight magnetic tape drives, two CDC 844 disc drives, a card reader, card punch, printer, and the CDC 3811 relocation unit. There are seven 3800s located at the STC (of which five are used operationally), two 3800s located at SDC's facility in Santa Monica, and one at the SAMSO facility in El Segundo, California.

- System IIB. System IIB is a modular, disc-oriented, executive-control, operating and support system designed primarily for the AFSCF CDC 3800 computer configuration. It encompasses the executive (SYMON), utility programs, and the Parameter Test Subsystem. The debugging capabilities of these subsystems are described in Paragraph 2.5.2.

  The purpose of System IIB is to control and support flight operations and to provide the tools to produce, modify, and maintain the operational system programs required for all phases of AFSCF flight-support activities.

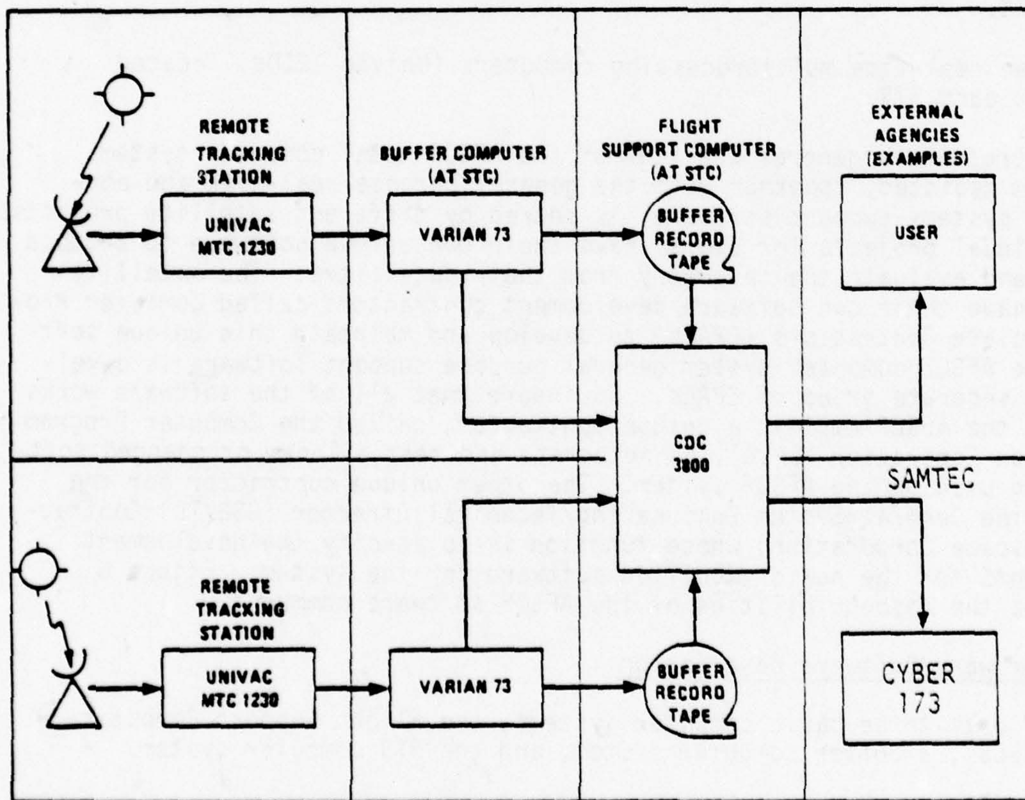


Figure 5. AFSCF Computer System Overview

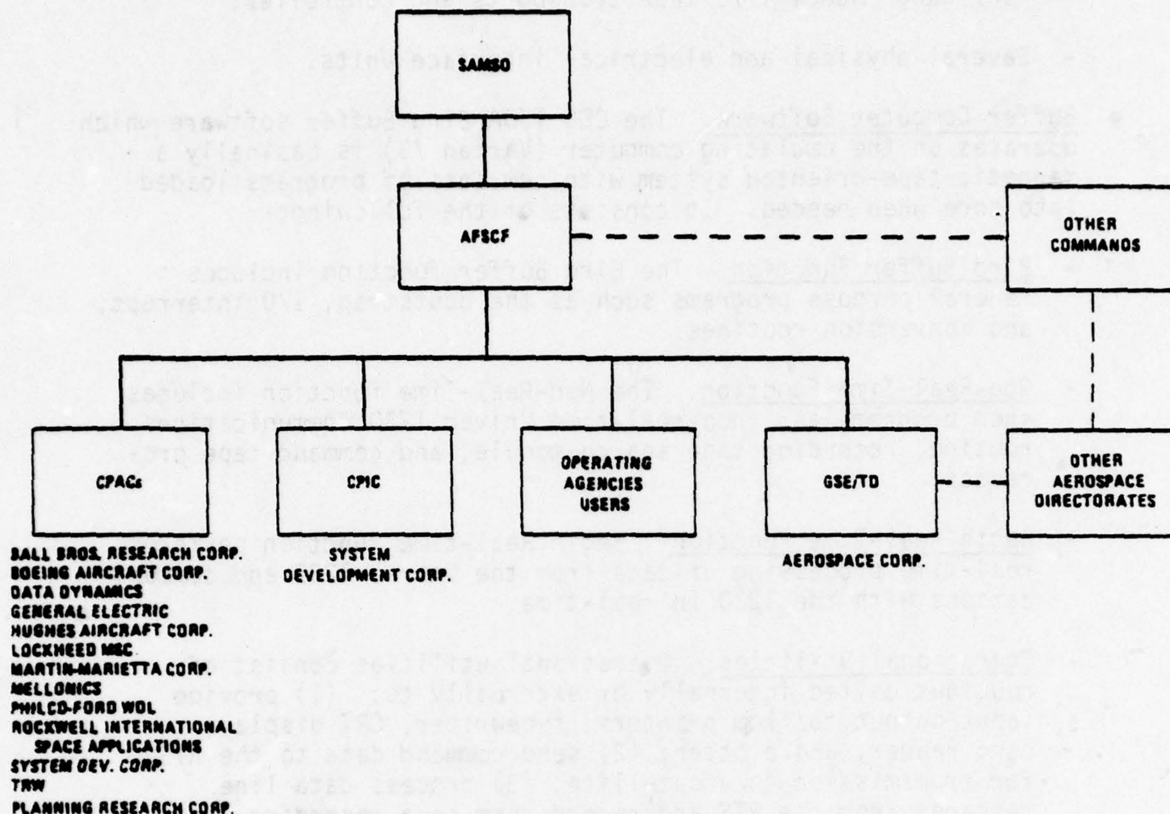


Figure 6. SCF Software Community

The buffer computer system acts as an interface between the non-real-time CDC 3800 systems and the real-time Univac 1230 system. It consists of the following hardware:

- Emulated Buffer Computer Complex (EBCC). The EBCC replaced the CDC 160A Bird Buffer computer and some of the peripheral equipment associated with it. The EBCC consists of the following components:

  - A Varian 73 computer which uses microcode, or firmware, to analyze each CDC 160A instruction and perform the equivalent function in the EBCC.

  - An operator's control terminal (OCT) which includes an EBCC control panel (a functional simulation of the 160A computer console), a Lear-Siegler ADM-1 alpha-numeric keyboard and cathode ray tube display terminal (ANK/CRT) and controller, a Data Products Model 2230 line printer and controller, and a Documation Model M200 card reader and controller.

- Four Wang Model 1175 tape transports and controller;

- Several physical and electrical interface units.

- Buffer Computer Software. The CDC 160A Bird Buffer software which operates on the emulating computer (Varian 73) is basically a magnetic tape-oriented system with new sets of programs loaded into core when needed. It consists of the following:

  - Bird Buffer Function. The Bird Buffer function includes general purpose programs such as the bootstrap, I/O interrupt, and conversion routines.

  - Non-Real-Time Function. The Non-Real-Time function includes such programs as: non-real-time Univac 1230 communications routine, recording tape search module, and command tape processor.

  - Begin Real-Time Function. Begin Real-time function performs real-time processing of data from the Univac 1230 and communications with the 1230 in real-time.

  - Operational Utilities. Operational utilities consist of routines called internally or externally to: (1) provide input/output to/from printers, typewriter, CRT display, card reader, and plotter; (2) send command data to the RTS for transmission to a satellite; (3) process data line messages from the RTS and record them on a recording tape; (4) process and reduce a recording tape; (5) perform telemetry limit compare function.

  - Non-Operational Utilities. These utilities include non-operational functions such as a tape-dump routine.

Each RTS has a single UNIVAC 1230 Military Tactical Computer (MTC) and associated peripherals. The UNIVAC MTC computer is a medium-scale, general-purpose, digital machine with multiprogramming and dual processing capabilities. Logical, semi-independent computer units (each with independent internal timing controls) share functional operations to provide asynchronous processing in systems involving complex and massive amounts of data in real-time or scheduled activities. The computer accepts STC-originated acquisitions, slave tracking, and command data from the Communications Buffer (COMB), stores the data on a magnetic disc storage unit, and processes the data so that the Input/Output Buffer (IOB-II), and the Command Buffer (CMDB) can receive it in the proper sequence. The RTS computer equipment configuration includes:

- UNIVAC 1230 MTC computers
- UNIVAC 1232-04 I/O consoles
- Anelex 414 line printers
- UNIVAC 1531 mass storage adapters

- CDC 3254 disc controllers
- CDC 854 disc drives
- Philco display and control subsystem
- IOB-II
- COMB
- Telemetry Data Processor Buffer (TDPB)
- Telemetry Data Processor, Model II (TDP-II)
- CMDB

Each AFSCF RTS performs multi-satellite programs support, using a data handling system composed of a UNIVAC 1230 MTC Dual Control Processor computer, with a set of peripheral equipments. RTS support encompasses tracking, commanding, and processing of the various telemetry data formats. Except for command and telemetry processing, the AFSCF RTS computers use the same basic software programs in such areas as data input/output for disc storage, output to local printers, data transfer to the STC Bird Buffer via the communication system, status checking of peripheral equipment, etc. A unique software program, called MADCOM, is used for command processing. Each of the satellite programs uses a distinct set of telemetry processing programs.

RTS software is divided into five major modules that perform a specific function or operate during a particular phase of operation. The five major modules are:

- Operating System Executive and I/O Subsystem
- The System Prepass Programs
- The System Preacquisition Programs
- Pass and Postpass Telemetry and Tracking Processing
- The Commanding Subsystem Programs

2.5.2 Site-Specific Debugging Software

The debugging software available to the AFSCF includes capabilities for the Flight Support, Buffer, and RTS Systems. This software is described in the following paragraphs.

2.5.2.1 Flight Support Computer System Debugging Tools

The debugging tools available under the CDC 3800 System IIB operating system are provided by the executive (SYMON), utility, and parameter test subsystems. A description of the Flight Support debugging software follows:

- Executive Software (SYMON) Debugging Tools. SYMON provides the following debugging capabilities which can be applied to any operation request executed during a job:

- Interrupt On Arithmetic Faults Option. This option allows the user to interrupt the program if arithmetic faults occur, and output a message for each fault detected. Interrupts for the following fault types can be requested:

  1. All faults
  2. Divide
  3. Underflow
  4. Overflow
  5. Shift
  6. Fixed-point overflow

- Memory Map Option. A memory map is a list of all elements (routines, data blocks, alternate entrances) in core, with their corresponding virtual address, length, starting page, segment, and physical address.

- Corrector List Option. The correctors for the requested program and its environment are listed.

- General Timing Information Option. The start, stop, and duration times for execution of the requested program are listed.

- Trace On Jumps Option. A trace of all jump instructions executed by the requested program and its environment is produced.

- Parameter Test Option. The debugging capabilities of the Parameter Test Subsystem are invoked during execution of the requested program.

- Load, Do Not Execute Option. The requested program and its environment is loaded into core but not operated. This option allows core load problems to be debugged more easily.

- Dump Option. A core dump of the requested program and its environment is produced. The dump can be requested:

  1. After execution of the requested program.
  2. If requested program terminates abnormally.
  3. After execution of a program which is called and loaded in core from disc by the requested program.

  The dump provides the following for each element in core:

  1. Virtual address of element (i.e., relative to the beginning of a segment).
  2. A memory map.

3. A console scoop (contents of computer registers at time of dump.

4. List of program correctors for all elements.

5. A core listing of each element in either hollerith, decimal, floating-point, mnemonic op-code, octal, or symbolic format.

- Set Pseudo Switch Options. Pseudo jump, sense, or stop switches can be set which have the same effect on the software as manually setting the switches. This feature is used together with programmed-in aids to provide debug code in a program which can be switched-on only when desired via this pseudo switch option.

- Specific Timing Information Option. Timing on a program level can be obtained. Output includes: The number of disc accesses and disc data handler calls, the total times of execution in milliseconds of the disc accesses and data handler calls, and a total SYMON module time in milliseconds. The output also includes a tabulation of the number of requests for each I/O unit (e.g., typewriter, tape, card punch) and the time spent servicing those requests.

- Segmentation Diagnostic Information Option. Messages warning of segmentation problems and/or diagnostic table information can be provided.

- Check Data Base and Data Handler Directors Option. The data base related directories are validated prior to the operation of the requested program.

- Loop Trace Option. This is a manual interrupt request which will cause a jump trace to be produced for approximately ten seconds of program operation. Upon termination of the trace, a core dump and memory map is output.

- Disc Data Handler Diagnostics Option. For each call to the operating systems disc data handler software by the requested program, or any program in its core environment, diagnostic information can be provided. Information includes:

1. Calling routine name.
2. Type of disc access.
3. Operation (read, write, position, etc.)
4. Name of file.
5. Record number.
6. Buffer address.
7. Buffer size.
8. Starting pack and sector used.
9. Open table entry for the elements.

- Utility System Debugging Tools. The utility program subsystem of System IIB provides both external user requested debugging aids and programmer requested aids specified within the JOVIAL code of a program. The following debugging tools can be requested by the System IIB user:

  - System List Master Tape (SLMT) Program. This program is especially useful for debugging interface problems for an integrated set of routines. The master tapes in System IIB contain all of the operating system and user support programs used on the CDC 3800 system for a particular satellite project. The master tapes are loaded onto disc and the routines are then loaded into core by SYMON as requested. The master tapes, then, can be thought of as a library of routines for a project. SLMT provides the following information useful for debugging:

    1. A list of all compool elements (routines and data blocks) on the master tapes and a list of all other routines which refer to (set, use, both) each element.
    2. A list of the environmental elements (other routines and data blocks referenced directly or indirectly by each compool-defined routine), and how an environmental element is referenced (set/used/both) by the routines.
    3. A list of all compool-defined routines on the master tapes which reference a compool-defined table.
    4. A list of all compool-defined routines which reference a compool-defined table item.
    5. A list of all compool-defined routines which reference a compool-defined simple item.
    6. A list of the compool-defined arrays, tables, table items, and simple items of a compool-defined routine and how they are referenced (set, used, both).

  - System Disc Dump (SDD) Program. This program provides a symbolic or octal listing of specific data blocks, routines, compools, directories, correctors, areas, or sectors of disc.

  - System Flowcharter (SFLOW) Program. This program produces flowcharts of J4 JOVIAL programs which are useful in analysis. The level of detail of the flowchart is controlled by the user.

  - System Master Tape Comparison (SMATCH) Program. This program compares master tapes in their entirety or any subset of elements common to both tapes. All differences found are listed with an appropriate description.

  - Disassemble Compool (LOOPMOC) Program. This program produces a formatted listing of one or more compools from disc or tape. The format is especially applicable to debugging compool-related interface problems.

- JOVIAL Reformatter (RFORMAT) Program. This JOVIAL preprocessor reformats a program's source code. Especially useful to debugging is the identification of BEGIN and END locps via identation and asterisk-charting to the right of the listable output.

- System Data Base Directories Check (SDBCHEK) Program. This program checks for errors in the data base directory and disc data handler directory and for correspondence between them. Checks are made for duplicate entries or addresses in either directory and conflicting data in matching entries. This is useful in debugging data base-related interface problems.

- System Data Base Compare and Verification (SDBCOM) Program. This program compares data bases or any subset of elements on either data base. This information is useful in debugging data base-related problems.

- System Data Base Display (SDBD) Programs. These programs display elements of a data base in a user provided format enabling debugging on a higher language level.

The following service routines can be requested within a JOVIAL-coded program to provide debugging information during the program's execution:

- Instruction Time On (CLOCKON) Routine. This procedure sets an item with the current value of the computer clock. When this routine is used in conjunction with the CLOCKOFF routine, the time required for a sequence of instructions to operate may be determined.

- Instruction Time Off (CLOCKOFF) Routine. This procedure computes the elapsed time for a given sequence of instructions within a program and its environment.

- Arithmetic Fault Selection (FAULT) Routine. This procedure will select interrupt on all or specific arithmetic interrupts. Specific interrupts are divide, exponent overflow, exponent underflow, shift, and fixed-point overflow faults.

- System Data Block List (SDBL) Routine. This procedure will format the contents of a compool-defined data block for listing.

- Core Memory Dump (SNAP) Routine. This procedure provides an octal dump of a specified area in virtual core memory.

- Select Or Deselect Trace On Jumps (TRACE) Routine. This procedure provides the capability to select or deselect a trace on all jump instructions.

- Parameter Test Debugging Tools. The Parameter Test Subsystem of System IIB is dedicated totally to debugging. It is invoked at the time a user requests a program to be operated under SYMON and provides three major capabilities (the user may combine any or all of these capabilities in any one run) as follows:

  - Record Option. The user may get recording and console scoops anytime during the operation of the tested program. The user has the following options concerning when to record:

    1. Before, after, or during execution of a program.
    2. If a time limit is reached.
    3. If the program aborts.

    The user has the following options concerning what to record:

    1. Record all segmented elements in core.
    2. Record selected portions of core (tables, arrays, items, or portions of programs) in symbolic, octal, floating point, decimal, hollerith, or mnemonic format.
    3. Record selected portions of core which have changed during execution of a program.
    4. Record console settings.
    5. Record all the data from one location to a second and inclusive location.

    Additional user options are:

    1. Option to terminate or continue execution of a program after recording has been completed.
    2. Option to perform multiple passes of a program and to perform different recording on difference passes.
    3. Delay reduction of recording tape until later, or reduce immediately.
    4. Record within a program starting at certain JOVIAL statement label, switch, close, procedure, or function, plus or minus an increment, or at a certain octal location.

  - Set Parameters and Data Option. The user may set parameter and data values anytime during the execution of his program. The user has the following capabilities for setting values:

    1. Set Parameter Values. If an internal procedure or a compool defined routine is to be executed, the user may set the parameter values prior to executing the routine or procedure. Setting options are:

       a. Setting call-by-value parameters

b. Setting call-by-name parameters

c. Setting value into space allocated for call-by-name parameter

2. Set Data Values. The user may set values into compool defined or internally defined tables, arrays, or items. Setting options are:

   a. Setting NENT (number of entries) of either an internal or compool defined table.

   b. Setting either an item, complete table, all entries of a table item, or a complete array to zeros.

   c. Setting a simple item or octal location including specification of units for input value.

   d. Setting table item or array including specifying value's units, selective entries, and repeating values.

The user has the following options available:

   a. Set values either before or during execution of a program.

   b. Perform multiple passes of a program, and set different data on different passes.

   c. Setting during the execution of a program where setting shall take place immediately before the execution of JOVIAL statement label, switch, close, procedure or function.

- Trace Option. The user may follow the operation of a program on an instruction-by-instruction basis. There are three different forms of traces.

  1. Regular Trace. The regular trace will trace the normal sequence of program instructions.

  2. Reference Trace. A reference trace will trace only instructions which reference locations within a designated range. Range references are arrays, tables, table items, simple item, or data block.

  3. Jump Trace. A jump trace will trace only jump instructions.

  The user has the option to:

  1. Trace more than one area of a program (the traced areas may not overlap, however).

  2. Perform multiple passes of a program, and perform different traces on different passes.

3. Trace within a program starting and ending at certain JOVIAL statement labels, switches, closes, procedures, or functions, plus or minus an increment, or at certain octal locations. Also, trace within upper or lower blocks within the starting and ending locations.

2.5.2.2 Buffer and RTS Computer System Debugging Tools

Varian 73 and Univac 1230 computer systems of the AFSCF share one debugging tool, the Station Ground Environment Simulation (STAGES) system, and also possess unique tools of their own. These capabilities are described below as follows:

- STAGES. STAGES provides a test bed facility at the STC for the development and integration of the real-time computer programs for AFSCF. It simulates the components of the Buffer Computer System and the RTS System, as well as the command responses, telemetry inputs, and tracking data of the various satellites controlled by the system. STAGES is comprised of both hardware and software elements. It provides a repeatable, dynamically controlled real-time environment, and other features that are ideal for real-time debugging. The hardware components of STAGES are:

  - The Varian Emulated Buffer Computer.
  - The Univac 1230 RTS Computer.
  - The actual interface equipment between the EBC and the 1230.
  - An XDS SIGMA 2 computer, also referred to as the Environment Controller (EC)
  - The Interface Simulation Device (ISD).

  The first three items in the above list are exactly the same as the operational components. The last two represent the unique simulation functions of STAGES.

  The software components of STAGES are divided into three categories. The first category contains the utility programs which perform basic developmental needs such as tape manipulation, program assembly, and master generation. The second category contains all of the programs which generate real-time simulation inputs and diagnostic outputs. These programs are called real-time support programs. The third category contains all the programs that operate in the SIGMA 2 during real-time simulation. These programs are called the System Environment Simulator (SES) programs.

  Stages is primarily a testing tool but its capabilities make it an ideal system in which to debug the real-time software. The following debugging capabilities are provided by this test bed:

- Interface Simulation Device (ISD). This hardware device provides the necessary interfaces to simulate an RTS environment within STAGES. It simulates the hardware which provides interrupts, status words, system time, satellite tracking data, and command accept/reject verification data. The ISD allows the user to completely simulate or recreate conditions or sequences of events likely to be encountered by the Univac 1230 system at an operational RTS. This ability to recreate conditions is especially useful in debugging. A software problem can be duplicated and recreated over and over until enough symptoms have been gathered to isolate the cause of a problem.

- Breaklining. The breaklining feature is very important in debugging real-time software problems associated with interface timing. It is important that any recording provided for debugging not cause timing to change to the point where a problem is no longer being duplicated. When STAGES is used as an RTS real-time environment simulator, the 1230 MTC, the Varian 73, and the EC operate asynchronously. Certain tasks assigned to the EC, such as the recording of much of the 1230 MTC's input/output actions, require the EC to accomplish more in a period of system time than is possible in real-time. The ISD employs a BREAKLINE or clock-stop circuit which allows the EC to satisfy this requirement. When the EC's workload surpasses that possible in real time, the EC, or the ISD, sets the BREAKLINE which effectively stops system time until the EC has caught up with its processing. The EC then releases the BREAKLINE and system time continues from where it stopped when the BREAKLINE was set. In this way system time integrity is maintained.

  In addition to stopping and restarting various timing clocks within the ISD, the BREAKLINE signal is also transmitted via cables to the 1230 MTC and the Varian 73. This signal is used by the CPUs of these computers to halt the main timing chains at the completion of the instruction currently being executed. In this way the programs operating in these computers are kept in sync with system time and are unaware that system time is running slower than real time.

- SCRIPT Tape. The STAGES SCRIPT tape provides the user with a time-ordered, repeatable sequence of events for environment control. The tape, input to the Environment Controller (SIGMA 2) executive during real-time simulation, directs specific operations to be performed in support of the user's environment characteristics. These operations include development tools, complex telemetry stream modification, satellite tracking data simulation, and all of the switch actions. Thus, the actions are exactly repeatable if a test is rerun using the same SCRIPT tape.

- Recording Tape. The STAGES recording tape is generated by the SIGMA executive during a real-time simulation run. The tape contains chronologically ordered, detailed information about data transfers that take place between the 1230 and the interface and simulation devices within the ISD, as well as between the 1230 and EBC computers. After a real-time simulation run in which a recording tape is generated, the user can selectively list information from the tape for debugging purposes.

- Date Line Control. The user may control the data rate of data lines in the real-time system and initiate resync of the data. Also, all data line messages can be monitored, recorded, modified, and delayed by user control.

- Input/Output Buffer Control. The Input/Output Buffer (IOB) at an RTS is a hardware device which acts both as a timing source for the 1230 MTC and as an interface between the 1230 MTC and various hardware units at the RTS. Simulation of the IOB by the ISD provides the user with the capability to control the system time and station time used by the RTS and Buffer computer systems, either stopping them or changing their value. Also, the IOB contains many general purpose control and status registers to interface the 1230 MTC with various RTS equipment. Simulation of the IOB by the ISD makes these registers accessible to the SIGMA software during real-time simulation and allows the user to selectively modify them. The control of timing and general-purpose registers provide an effective means of debugging real-time interface problems in the real-time system.

- Telemetry Data Control. Control functions are available through STAGES software to stop the simulation of telemetry data streams, change data rates, introduce overriding telemetry values, and control of the source of discrete verification responses to commanding. These features are especially useful in debugging satellite-unique, telemetry-related software problems.

- Command Buffer Control. Control functions are available through STAGES to initialize or change Command Buffer status bits during the course of real-time simulation.

- WAIT Function. STAGES has a WAIT function which effectively freezes the RTS environment being simulated, thereby enabling the user to study and diagnose problems as they arise.

• Buffer Computer-Unique Debugging Tools. The following tools are available for debugging the Varian 73 EBC software:

  - DUMP Request. The DUMP function provides the capability to specify which core data are to be dumped (bank, first word address, last word address), when to dump (before, after, or before and after a subroutine executes), the format (listable or binary), and where the data is output (recording tape, printer, or CRT).

- MSTUD Routine. This is an in-house routine of the development contractor used to provide an EBC symbol tape set/use listing.

- RTS Computer-Unique Debugging Tools. The following tools are available for debugging the Univac 1230 RTS software:

  - CORE DUMP Request. The CORE DUMP function provides the capability to dump core onto disc and then route specific areas to the printer or magnetic tape. The disc data may also be transferred over data lines to a Varian 73 computer at the STC, or may be output to the printer or tape via another dump call. Automatic core dumps may be generated when high priority interrupts occur, and with the proper switch settings a core dump is initiated when a program breakpoint match interrupt occurs. (Breakpoint address is entered in a hardware breakpoint register).

  - LOOK Request. The LOOK request provides the capability to examine the contents of a specified core address. Only one address per request may be specified.

  - MLID Routine. This is an in-house routine of the development contractor used to provide an RTS data base set/use listing.

  - MFLOW Routine. This is an in-house routine of the development contractor used to generate a flow-chart from a program's source code.

  - MINCE Routine. This is an in-house routine of the development contractor used to compile a list of symbolic correctors currently applied to the RTS software.

2.5.3 Vendor-Supplied Debugging Software

No vendor-supplied debugging software is used. All of the operating systems and utility subsystems were developed specifically for the AFSCF.

2.5.4 Testing and Debugging Methodology

The AFSCF has independent, development contractor test groups and a CPIC. The test groups and CPIC participate in the system requirements and design review process and influence testability of the software design.

The testing methodology for the AFSCF is based exclusively on code execution testing. There is no static code analysis testing involving code analysis tools in this project, although such tools do exist [e.g., the JOVIAL flowcharter (SFLOW), the 3800 system cross referenced and interface consistency analyzer tool (SLMT), and the code comparator tool (SMATCH)].

The typical SCF software development cycle is depicted in Figure 7. The phases pertinent to this debugging study are the coding and checkout phase and the Category I testing phase. The software management standard used for AFSCF software is Exhibit 61-47B for established software, and a tailored version of AFSCM/AFLCM 375-7 for more recently developed software. Figure 7 reflects the standard for newer software. Military standards are also followed for documentation and configuration control policies within the AFSCF.

2.5.4.1 Software Integration

Since there are more than 15 separate software development contractors for the AFSCF, and the systems for which they are responsible are in different states of development, the methods of integrating different programs within a software subsystem vary considerably. New software development usually is current with the state-of-the-art management and development methodology. For example, newer software projects are employing the build concept, top-down design and structured-programming techniques. On the other hand, software projects which experience only minor modifications due to error correction and minor design changes follow no set procedures for design, coding, and integration.

2.5.4.2 Testing Levels

The categorization of the different testing stages performed for the AFSCF also is difficult because of the multiplicity of users and developers, and the new or changed characteristics of the software being developed. The following is a generalized categorization of the levels of AFSCF testing which indicate the testing methodology used for the system:

- Program Testing. Computer program testing is testing performed primarily to support the design and development process of the CPCs. These tests are comparable to engineering tests and evaluations performed on equipment Configuration Items (CIs). These computer program tests are conducted informally at the development contractor's test facility without specific approval of procedures by the Air Force. Tests may occur whenever they are required to support the design and development process of CPCs. Testing of software in computer program production is a sequential process that may consist of the following stages:

    - Pre-implementation Design Tests. If the program is sufficiently complex, tests may be run on trial designs prior to establishing an initial design approach. Typically, these tests indicate performance characteristics, computational accuracies, storage limitations, etc. Tests of this type may continue throughout the design process.

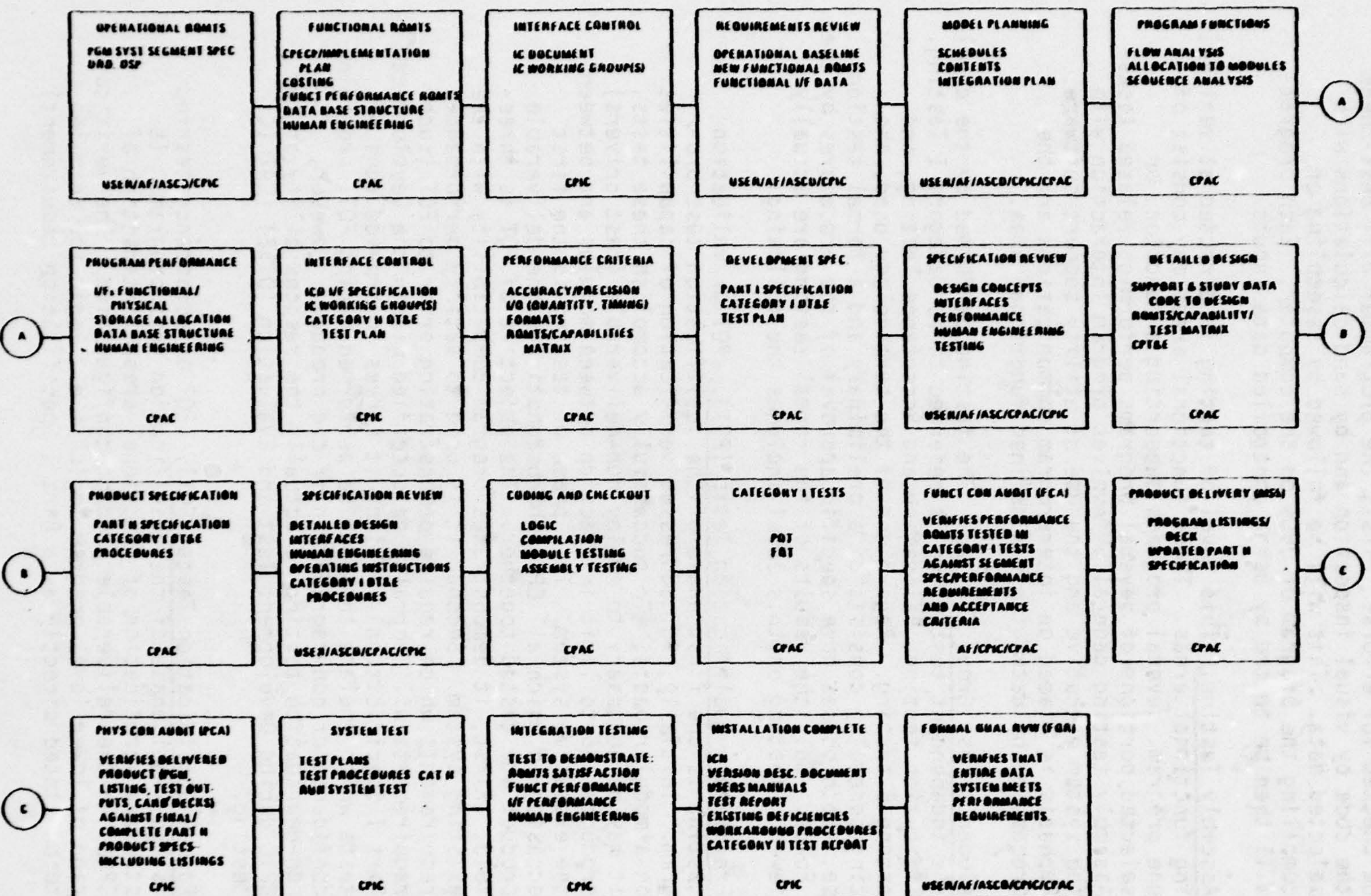


Figure 7. AFSCF Software Development Cycle

- Module Testing. This level of testing is directed at the program or CPC level. As each individual CPC is coded, it will be checked for errors. First, the programmer will desk-check the code by visual inspection and by hand calculations with selected data. This will be followed by assembling or compiling the program or CPCs on the computer. Each program will then be tested by using controlled data inputs.

- Assembly Testing. This level of testing is directed at verifying functional areas. These functional areas may consist of one program, several programs supporting a function, or selected portions of several programs performing related tasks. Assembly testing generally involves program interaction with the system executive and the use of utility support software. Emphasis is placed on interprogram communications and the processing of data within a defined functional area.

• Developer's Test Group Testing. The testing performed by the developer's independent test group is referred to as Category I Testing, Qualification testing, Validation and Acceptance Testing, and Milestone 5 testing. Regardless of the terminology used, the testing generally consists of a preliminary and a formal testing phase. Both phases have specific approval of the procedures by the Air Force. Only the results of the formal testing are actually reviewed. Testing on this level includes the following:

  - Preliminary Qualification Tests (PQT). PQT or validation testing is the first phase of the qualification test program. Initially, PQT addresses verification of capabilities on a modular basis. To successfully accomplish these tests, it may be necessary to employ non-deliverable (test drivers) products and to limit interaction between modules and between the executive system. This phase of testing is the first occasion in which all CPCs which constitute the deliverable product are tested together. The objective of PQT is threefold. First, it demonstrates program compatibility with the existing system. Second, it is used to satisfy performance requirements which require demonstration prior to FQT (such requirements will normally be specified within the Development Part I Specification). Third, it serves to provide runs of tests which emulate those to be performed during FQT, providing confidence of non-degradation of the product. However, non-degradation testing is actually the responsibility of the CPIC during Development Test and Evaluation (DT&E) (CAT II) testing.

  - Formal Qualification Tests (FQT). FQT or acceptance testing is the final phase of the qualification test program. It stresses verification of functional areas (interaction of CPCs) in the deliverable product configuration. The verification of lower level program logic (i.e., logic too detailed to be stated directly as a Part I Specification requirement)

is not addressed during FQT nor is an attempt made to demonstrate all capabilities. Hopefully, all system requirements are demonstrated in PQT, and FQT takes the form of a demonstration and reconfirmation. Those system requirements to be satisfied during PQT are identified within a formal test plan.

PQT/FQT is conducted by the development test team at their test facility. The tests use artificial and real input types and formats and include functional as well as subfunctional-level test procedures. FQT provides the evidence necessary to authenticate a statement of readiness to begin DT&E testing.

- Independent System Integration Testing. Upon completion of FQT and product turnover, the CPIC performs system integration testing or DT&E. DT&E demonstrates that the software fulfills its operational requirements, interfacing with the entire AFSCF computational environment. In these tests, specific testing against requirements from the Development (Part I) Specification are addressed again. In testing certain project-unique software, the CPIC performs detailed testing to insure that all inputs, outputs, error conditions, restrictions, program paths, mathematical algorithms, and interfaces are correct when the system is tested via data input in a normal, operational fashion. This testing may be as detailed as the programmer's module level testing, but is performed within the operational environment.

The testing phases applicable to the debugging study are the programmer's assembly testing, PQT, and FQT. This testing spans the integration of software modules during development. It is during PQT that the software comes under internal configuration control. At this time, any errors discovered are documented via a Software Problem Report (SPR) or Discrepancy Report Form (DRF), and corrections are released via a Modification Transmittal Memorandum (MTM). Also, all compool, data base, and document/specification changes resulting from problem corrections are formally controlled using the Compool Change Request (CCR), Data Base Change Request (DBCR) or Data Modification Request (DMR), and Document Update Transmittal (DUT) forms.

2.5.4.3 Debugging Methods Employed During Development Integration

Development integration phase debugging for the AFSCF software involves both batch and interactive methods. The debugging of problems encountered with CDC 3800 software is almost exclusively batch in nature. Tests are usually not attended by the test personnel, so when a problem is encountered, another batch-type job is later run to duplicate or isolate the problem, or to show it cannot be repeated (e.g., a machine-problem). These production type runs might require many repetitions before a problem is isolated and fixed. If the problem is critical in nature, the debugger will usually request "attended time" where a more interactive type of debugging will take place. This implies that the debugger inputs changes to the job on-line via the typewriter or card reader until enough symptoms have been gathered or a "fix" eliminates the problem.

The debugging activities for the real-time software (Varian 73 and Univac 1230 software) of the AFSCF is more interactive because of computer availability. There are many more "attended time" jobs where the programmer debugs the software on-line, as was described for the critical 3800 problems. Also, since the Univac 1230 software involves console switch action inputs and CRT outputs the real-time system calls for more interaction.

Whether a problem is the result of a batch or interactive type test, the initial encounter of the problem has a predefined set of computer operator procedures associated with it. For example, if it is apparent that a program is experiencing an infinite loop during a 3800 run, a manual interrupt is made and a loop trace is requested. Likewise, a core dump and console scoop is taken if a 3800 program aborts because of an illegal instruction. These immediate debugging actions are usually all that is needed for the programmer to isolate most problems.

For CDC 3800 software debugging, certain features associated with the AFSCF's JOVIAL/J4 compiler can be employed at the preliminary stage of problem duplication and isolation. This includes using the JOVIAL reformatter output for more easily read source code, and using the JOVIAL cross-reference output for data/routine set/use determination. The JOVIAL flowcharter, while available, is never used for debugging a problem.

For Varian 73 and Univac 1230 software debugging, STAGES is the primary tool for problem duplication and isolation. As opposed to 3800 debugging, language-associated debugging features are not important here. If a problem solution is not immediately apparent, the execution-approach to debugging is much more common than the off-line code analysis approach required for the less available 3800 computers. Computer availability, then, seems to be a definite characteristic of a system's debugging methodology, and a definite contributor to the types of debugging tools developed or required.

The AFSCF employs the following methods for debugging which do not involve specific debugging tools:

- Employ program correctors to cause a program to halt at a certain location so a manual dump can be requested.
- Turn on the programmed-in diagnostics via manual sense switch settings, data base settings, or program correctors.
- Use another computer when machine problems are suspected.
- Utilize previous version of operating system if application-versus-system software incompatibility is suspected.
- Use different master tape and data base tape copies if input problem is suspected.

While the actual isolation of problem causes is entirely at the discretion and imagination of the individual debugger in the AFSCF, there are patterns for each process. The following methods are used to eliminate various factors from consideration as problem contributers:

- Trying other input options or work-arounds
- Rerunning on other equipment (machine problem elimination)
- Rerunning on other software configurations
- Bypassing operation of certain software code via program correction or recompilation

Since the AFSCF involves so many software development contractors whose software interacts with one another, the first attempt toward symptom accumulation is determining which program really is at fault. Once a specific program is suspected, the next step is the determination of programmer responsibility. Since, much contractor interaction is possible, a status report form for software problems, called the Software Analysis Report (SAR), is frequently employed for the assigned agency to report the suspected cause of the problem or suspected area of the problem as the debugging process develops.

One of the main problems a debugger is faced with is determining precisely what should happen under the input and configuration situation involved with a problem. The following means are used for this purpose:

- Program Documentation. The software system design is well documented and a debugger new to the system has detailed Part II Specifications at hand to determine the intent of the original designer/programmer.
- Software Analysis Reports (SARs). These interim status reports on a problem can be used to convey preliminary ideas on its cause, or reasons why it is not thought of as a problem.
- Configuration Control Board (CCB) Meetings. CCB meetings provide a forum for non-critical discrepancy report status and responsibility assignments.

2.5.5 Debugging Methods Actually Used

Interviews with the programmers of the AFSCF project indicate that most of the debugging tools or features discussed in Paragraph 2.5.2 are employed at some time or another during development integration testing. The following major debugging capabilities are used:

- SYMON debugging tools (all options used)

- System IIB utility system debugging tools, with the exception of the system flowcharter (SFLOW), the disassemble compool program (LOOPMOC), and the arithmetic fault selector (FAULT) routine.

- STAGES (all features used)

- The Buffer computer-unique tools (all options used)

- The RTS computer-unique tools (all options used)

It is notable that the CDC 3800 Parameter Test subsystem which was specifically created for debugging purposes is not used.

SECTION 3 - LITERATURE SURVEY RESULTS

This section presents a survey of current literature on debugging. The following bibliographies and abstracts were used to generate over 500 references for the survey:

- Bibliography supplied by RADC
- SDC's Software Technology Department bibliography
- NTIS abstracts
- Computing Reviews abstracts
- Computer & Control abstracts
- Computer abstracts
- Computer conference proceedings

A review of this data and further research resulted in the working bibliography presented in Appendix B. This bibliography covers the following four major debugging related topics:

- Debugging environment considerations (i.e., those elements of the development process which impact debugging)
- Software errors
- Debugging aids
- Debugging methods

The remainder of this section discusses the results of the literature survey in terms of these topics. The topics and their subtopics are, in turn, related to one another as the discussion progresses.

3.1 Debugging Environment Considerations

The first step taken in defining the debugging methodology is to identify those elements of the development process which impact debugging. It is expected that the components of the debugging process model will be based on the following environmental considerations:

- Software management methodology
- Software tolerance requirements
- Language considerations

AD-A069 541 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2
SOFTWARE DEBUGGING METHODOLOGY. VOLUME III. LITERATURE AND SITE--ETC(U)
APR 79 M FINFER, J FELLOWS, D CASEY F30602-77-C-0165
UNCLASSIFIED RADC-TR-79-57-VOL-3 NL

2 OF 3
AD
A069541

- Interactive versus batch systems
- System hardware/software architecture
- Integration concepts
- Testing methodology
- Testing tools and techniques

Each consideration, as well as its interrelationships, is discussed in the following paragraphs.

3.1.1 Software Management Methodology

A software management methodology establishes the guidelines for software acquisitions. It greatly influences the debugging methodology by providing the basic framework within which debugging operations will occur. The more developed the software management methodology for a given system is, the more planning can be made for the debugging activities and capabilities of the system. This methodology also establishes the procedures to be followed for the processing of software problems.

The literature survey indicates that software management methodologies for computer systems range from practically non-existent for some small commercial projects to fully developed, sophisticated systems for large-scale military projects. The software management methodology used in this study as background for the development of a debugging methodology reflects that used in large-scale systems, for the following reasons:

- Large-scale system software management methodologies contain the classical management functions. While the exact policies of the large-scale management concepts might not apply to all types of computer systems, the critical management functions that they address are applicable to any computer system. What is important to this study is defining all of the functions which are critical to debugging.

- The main objective of this study is the development of a methodology for defining a software debugging environment to be used by Air Force software development engineers during the integration phase of software modules. This requires that Air Force management concepts be included in any background established for the debugging methodology.

- The nature of debugging tools and techniques is such that all facets can only be explained for applications in which all facets are used. The management methodologies required to support the development of large-scale computer systems generally fulfill this requirement whereas those required to support smaller, less developed computer systems do not.

These reasons form the basis for the following discussion of the effect of large-scale system software management concepts on debugging methodology. The discussion is organized in terms of:

- Regulations and standards related to software management.
- Computer program verification, validation, and certification.
- Internal configuration control.
- Independent verification and validation.

3.1.1.1 Government Software Management Policies

Several Government software management policies are available for assisting in the selection of appropriate project milestones. These policies contain general guidelines for the acquisition life cycle, documentation, reviews, audits, testing, debugging, and configuration management requirements of software development. The management policies presently in use include those governed by the following Government regulations and standards: AFR 800-14, AFSCM/AFLCM 375-7, DoD 4120.17M, WS 8506, Exhibit 61-47B, SECNAVINST 3560.1, and Viking '75. Each of these regulations and standards possesses unique benefits applicable to some types of computer systems and drawbacks for others. For example, AFR 800-14 was issued primarily for software embedded in major systems [80,97]. SECNAVINST 3560.1 attempts to involve the user in the acquisition process [114]. Exhibit (Standard) 61-47B reflects the peculiarities of a fairly stable hardware environment, a large number of users of the system with varying needs, a continuing necessity to modify the application software to meet these changing needs, and a requirement to maintain a stable system configuration while modifications are being implemented [9].

Likewise, the detailed standards applied to a computer system can vary. The military standards which usually are applied with AFR 800-14, for example, include MIL-STD-490 for documentation guidelines, MIL-STD-480 for engineering changes, MIL-STD-483 for configuration management practices, and MIL-STD-1521A for reviews and audits, but for a particular application some may or may not exist and others may be added.

Furthermore, the limitations of these regulations and standards require that for most projects, the user amplify, augment, or tailor the fundamental precepts of the plan utilized. This is usually accomplished via supplements.

The following paragraphs describe the direct impact these regulations and standards have on the development of a debugging methodology. Subjects discussed include:

- Acquisition life cycle

- Key documents and events
- Configuration management
- Testing/debugging requirements

Acquisition Life Cycle.

The acquisition life cycle defined by Government regulations and standards establishes the basic framework within which the debugging operations will occur. Figure 8, according to R.E. Berri [9] presents "the life cycle as currently portrayed in Air Force management regulations. The AFR 800-14 depiction of the various activities of the development phase is also shown."

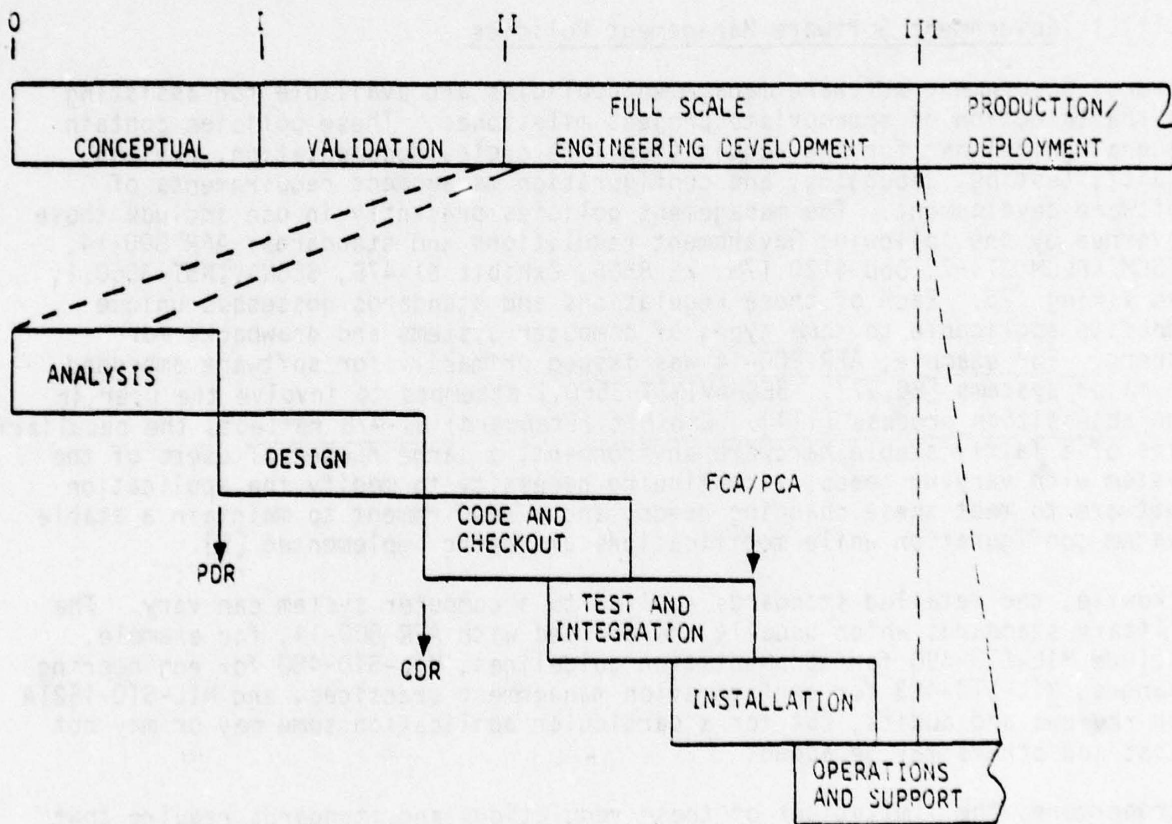


Figure 8. Software Life Cycle Phases and Development Activities [4].

Preliminary Design Review (PDR) and Critical Design Review (CDR) are the general design and detailed design reviews. Functional Configuration Audit/ Physical Configuration Audit (FCA/PCA) represent the formal government audits of the software products [110]. A comparison with other management concepts reveals no major differences which affect debugging. The life cycle depicted, therefore, represents a reasonable basis for the debugging study.

Key Documents and Events

The documentation requirements defined in the management regulations and standards directly relate to the debugging process. Key documents* include:

- System Specification (Type A, MIL-STD-490)
- Development Specification (Type B5, MIL-STD-490 Part I, MIL-STD-483)
- Product Specification (Type C5, MIL-STD-490 Part II, MIL-STD-483)
- Interface Control Document (MIL-STD-483, Appendix II)
- Test Plan/Procedures (DI-T-3703)
- Test Report (DI-T-3717)
- Users Manual (DI-M-3410)
- Programming Manual (DI-M-3411)

The Air Force data items selected are consistent with the general Air Force viewpoint [9]. The documentation required by the various management concepts differ greatly, but none of the documents excluded from the above list influence debugging. The types of information defined by these documents provide the debugging process with data for two of the major steps of debugging: (1) determining the exact performance requirement; and (2) determining the detailed testing requirement.

The key events which relate to the development life cycle and the above documents are:

- System Design Review (SDR)
- Preliminary Design Review (PDR)
- Critical Design Review (CDR)
- Preliminary Qualification Tests (PQT)
- Formal Qualification Tests (FQT)
- Functional Configuration Audit (FCA)**
- Physical Configuration Audit (PCA)**

These events are described and their requirements specified in MIL-STD-1521A.

*The specifications and the Interface Control Document (ICD) are described in related military standards; the testing documents and manuals are specified in Air Force Data Item Descriptions (DIDs).
**Sometimes extended to Formal Qualification Review (FQR).

Configuration Management

Configuration management is the discipline of applying technical and administrative direction and surveillance to (1) identify and document the functional and physical characteristics of systems and configuration items; (2) control changes to those characteristics; and (3) record and report change processing and the implementation status [97]. Government regulations provide guidance on this discipline. For example, AFR 800-14 and associated standards call for identification of software by Computer Program Components (CPCs) and Computer Program Configuration Items (CPCIs), control of changes via Engineering Change Proposals (ECPs), and recording and reporting implementation status via FCA and PCA. These monitoring and control functions affect the debugging process by attempting to establish guarantees that the software developed and tested is the same software being delivered, and that changes made to the software are visible. The regulations also call for a Configuration Control Board (CCB) which is empowered to act on all changes. In some systems, the CCB also monitors software problem status and resolves debugging conflicts.

Testing/Debugging Requirements

The Government regulations establish the general testing and debugging requirements to be followed within the acquisition process. The guidelines provide for accomplishing a test program and for testing and debugging tools and capabilities. (These guidelines are further discussed in Paragraphs 3.1.7, 3.1.8, and 3.3.)

3.1.1.2 Computer Program Verification, Validation, and Certification

Verification and validation refer to the review and testing process which take place within the computer program life cycle of large scale systems. The process may be performed by the development contractor or an independent contractor. AFR 800-14 provides guidelines for the process, outlining the tasks to be performed during the analysis, design, code and checkout, test and integration, installation, and operations and support phases of the software life cycle (see Figure 9). The process is primarily used during the development of software embedded in major defense systems. The following discussion defining verification, validation, and certification is based on a report [13] prepared by System Development Corporation under the direction of the Computer Systems Engineering Directorate (MCI) of the Electronic Systems Division (ESD), Air Force Systems Command (AFSC). This report is one of a series of Software Acquisition Management (SAM) guidebooks intended to help ESD Program Office personnel in the acquisition of embedded software for command, control, and communications systems.

The terms verification and validation currently have many different meanings. The terms as used in connection with certification mean one thing, and as separate testing stages mean another (see Paragraph 3.1.7). The verification-validation-certification usage applicable to the debugging study is defined as follows:

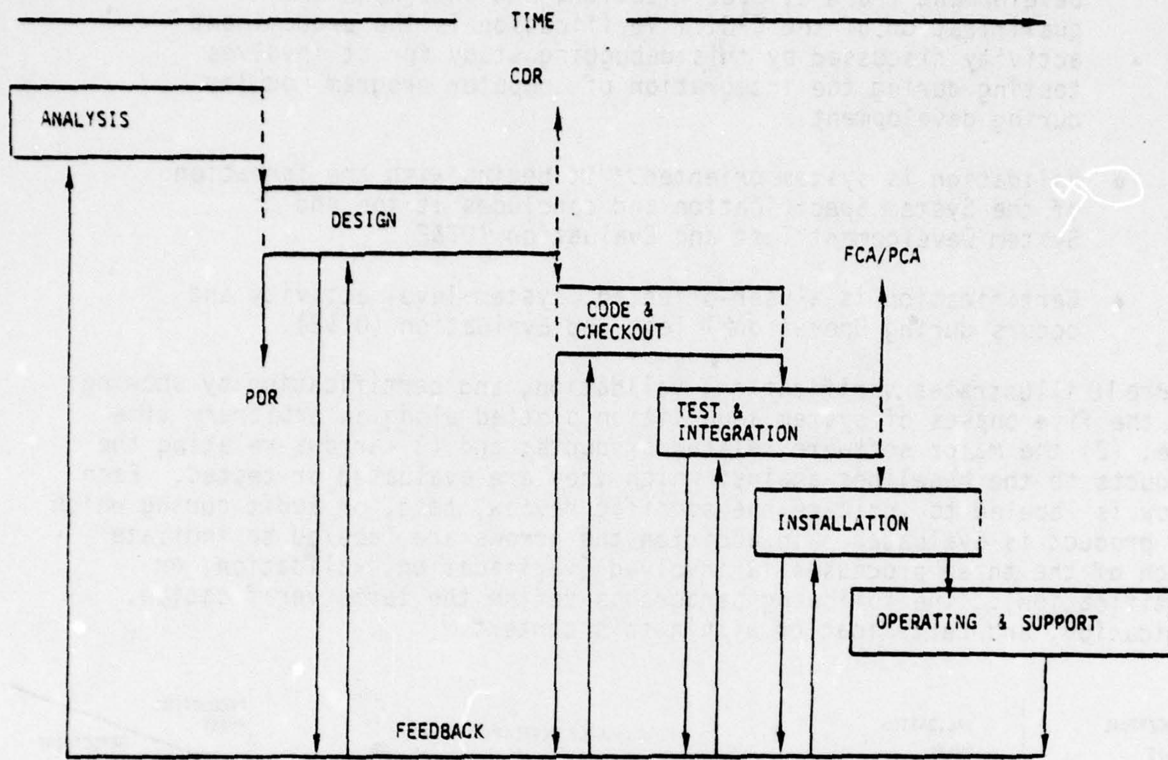


Figure 9. Software Development Life Cycle

- Verification is CPCI oriented. It begins with the system and software engineering activities, which lead first to CPCI definitions, continues to production of the CPCI Development (Part I) Specification, and ends with the qualification of the CPCI. Verification is the predominant activity discussed by this debugging study for it involves testing during the integration of computer program modules during development.

- Validation is system oriented. It begins with the formation of the System Specification and concludes at the end of System Development Test and Evaluation (DT&E).

- Certification is a user-oriented, system-level activity and occurs during Operational Test and Evaluation (OT&E).

Figure 10 illustrates verification, validation, and certification by showing: (1) the five phases of system acquisition plotted along an arbitrary time line; (2) the major software related products; and (3) arrows relating the products to the baselines against which they are evaluated or tested. Each arrow is labeled to indicate the specific review, test, or audit during which the product is evaluated. In addition the arrows are labeled to indicate which of the three processes is involved (verification, validation, or certification). The following paragraphs define the terms verification, validation, and certification within this context.

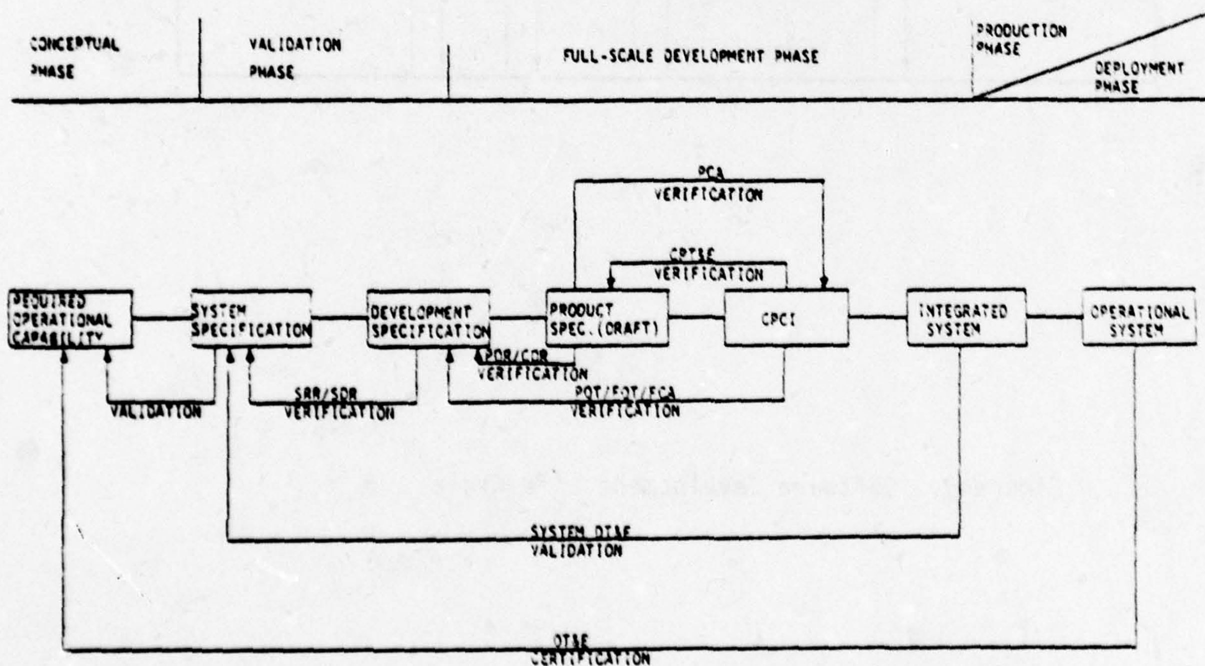


Figure 10. The Scope of Verification, Validation, and Certification [13].

Verification is the iterative process of determining whether the product of selected steps of the CPCI-development process fulfills the requirements levied by the previous step. Specific tasks comprising the CPCI verification process include:

- System engineering analytical activities carried out to ensure that the CPCI Development (Part I) Specifications reflect the requirements allocated from the System Specification (i.e., verifying the Development Specification).
- Design evaluation activities carried out to ensure that the CPCI design continues to meet the requirements of the Development Specification as the design proceeds to greater levels of detail (i.e., PDR and CDR).
- Informal testing of the CPCI and its components [Computer Program Test and Evaluation (CPT&E)] carried out by the contractor at his discretion to assist in development, provide visibility of progress, and prepare for formal testing.
- Formal testing of the CPCI carried out by the contractor in accordance with Air Force-approved test plans and procedures to verify that the CPCI fulfills the requirements of the Development Specification and to provide the basis for CPCI acceptance by the Air Force (i.e , PQT, FQT).

It is the last two task areas which fall within the scope of this debugging study.

Validation comprises those evaluation, integration, and test activities carried out at the system level to ensure that the system being developed satisfies the requirements of the System Specification. While the validation process, distinct from the system validation process, cannot be isolated since all evaluation and test activities comprising validation are focused at the system-requirements level.

Certification refers to the user's agreement, at the conclusion of OT&E, that the acquired system satisfies its intended operational mission. During OT&E, the system undergoes test and evaluation aimed at assuring operational effectiveness and suitability.

3.1.1.3 Internal Configuration Control

Internal configuration control refers to the development contractor's internal policies for managing the development of the software. Some of the policies which affect debugging include (1) standards and conventions for design and development, and (2) software problem policies for reporting, monitoring, and resolving software problems.

Standards and Conventions

Standards and conventions as used here refer to the internal guidelines for the design and development of software products. These guidelines, usually consolidated in the Software Development Standards and Conventions document and made available to all development personnel, are generally enforced by the developer's independent quality assurance team [115,119]. This document is usually project-specific and provides the following:

- Design standards (e.g., top-down, structured programming, modularity, and interface procedures.)
- Coding standards (e.g., structured program language, use of common routine for I/O error messages, console interface standards, module size standards, program organization, naming conventions.)
- Testing standards (e.g., build*, testing).
- Documentation standards (e.g., Development and Product Specification requirements, CPC listing requirements.)
- Program support tools (e.g., code auditors for standards, program library systems, system generation.)
- Error correction standards (e.g., software problem reports, software modification transmittals.)
- Debugging standards (e.g., programmed debug statement requirements, error message requirements.)

The standardized software which results from these internal guidelines should aid the debuggers by providing less complex coding, more controlled testing, and valuable debugging aids.

Software Problem Policies

Policies are established for reporting, monitoring, and resolving software problems. A number of reporting forms are used for this purpose, including:

- Discrepancy Report. This form is used to report a software problem. Ideally, it presents all information necessary to duplicate the problem. This includes hardware configuration, software configuration, test configuration, and exact description of inputs and operations involved [38,99].

*See Paragraph 3.1.6

- Resolution Report. This form is used by the programmer to report the changes made, if any, to resolve the problem. This is especially important for re-testing the area of code in error. The form generally contains a discrepancy report number, the program element changed, and a description of the change or an explanation of why no change was necessary. The form can also contain information useful for debugging data collection/analysis by indicating how the problem was isolated and what debugging tools were used. Also, the form might describe the tests performed to verify the fix, and data base, compool*, and document changes required [38 & 99].

- Software Analysis Report. This form may be part of the discrepancy or resolution reports, or a separate report altogether. It reports the status of a software problem based on preliminary and ongoing analysis by the debugger. It is useful for airing the debugger's view on whether a problem really exists or not, where the problem seems to reside, whether it should be fixed, work-arounds or alternative execution patterns, and the effects on the operational usage of the software [99].

- Design Change Request. This form is used when a problem is known to be a design problem as opposed to an implementation problem. It is used as a preliminary to the Engineering Change Proposal, (ECP), which is an external configuration management form [38,99].

- Data Base Change Request. This form is used to request a change to the data base. If the change is viewed as a software problem, it might be associated with a discrepancy report form [99].

- Compool Change Request. This form is used to request a change to the compool*. If the change is viewed as a software problem it might be associated with a discrepancy report form [99].

All of these forms are related to the debugging process because they initiate the process, describe its status, aid problem repeatability, and report the results of the process.

Associated with these forms is the Problem Status Report tool, a computer program which maintains a file of problems found and associated information on the status of problem resolution. This tool helps verify that all problems found are eventually corrected and maintains a history of the debugging process for analysis purposes (e.g., tool evaluation).

*A compool is a computer system-level definition of common data items, tables, arrays, data blocks, and routines. It is used in large-scale integrated systems to provide unique definition of data common to more than one subsystem.

3.1.1.4 Independent* Verification and Validation (IV&V)

IV&V testing, as opposed to the development contractor's independent test group's testing (see Paragraph 3.1.7), does not include debugging activities, except for those activities which are necessary to insure a suspected problem is, indeed, a problem. The IV&V testing contractor performs testing at the same time as the development contractor during the module integration phase. This usually is performed at an independent facility using independent tools and techniques. The literature survey revealed no information on the debugging tools and methods employed, if any, during the conduct of IV&V testing.

3.1.2 Software Tolerance Requirements

Tolerance requirements** refer to the class of software requirements which deal with processing measurement criteria.

- Program and interface timing requirements
- Storage requirements
- Accuracy requirements
- Throughput requirements
- Reliability requirements

A characteristic of tolerance requirements is that often the software has to be coded, loaded, and, in most cases, operated before major problems with the requirements' design or implementation are found. Code analysis generally does not prove these requirements are satisfied. Tolerance requirements may impact debugging capabilities because basic conflicts between meeting these requirements and providing debugging features in a system may exist. The impact of software tolerance requirements on debugging is discussed in the following paragraphs in terms of:

- Design deficiency resolution
- Debugging methodology
- Software development methodology

*The "independent" aspect usually pertains to separate contracts which aid the program manager in the review or testing process, or both. V&V is usually tailored to the need.

**For example, program and interface timing, storage, accuracy, throughout, and reliability requirements.

3.1.2.1 Role of Debugging in Design Deficiency Resolution

As with the translation of any system requirement into software, problems can be encountered in two separate phases: design and implementation. Design problems are the incorrect translation of the functional/performance requirement into a computerized process which will fulfill the requirement. Implementation is the coding of that computerized process into a form which, when executed on the computer, will fulfill the requirement. Debugging of implementation errors associated with tolerance requirements can be accomplished through the normal debugging tools and methods (see Paragraphs 3.3 and 3.4). The debugging of software which involves tolerance requirements design errors raises the following question:

> If the program does not meet the requirements for speed, size, accuracy, synchrony or reliability, but has been coded to specification, what is the role of debugging?

The position of this study is that the resolution of tolerance requirements design problems is not a debugging function, but a function of the verification and validation process, described in Paragraph 3.1.1.2. This process evaluates the growth of the design associated with each operational capability. To do this for tolerance requirements, it may employ the following tools:

- System Performance Simulation Model. This is a model of system hardware/software used to predict system performance over time. It is not necessary to examine all the design details of a system, so a simplified model can be used to gather information directly pertaining to that system's functional performance. The result of the modeling studies should provide sufficient data to verify the specific system concept. However, the simulation models are too simplified to be meaningful; or the structure of required data is complex, and a large quantity is required. Obtaining sufficient data to cover the time and conditions to be simulated can be costly, time consuming, and may complicate the analysis. However, once the strengths and weaknesses of a specific simulation are recognized, it can be a very efficient tool to aid the hardware selection evaluation, software performance projection, and performance monitoring [55, 7 & 60].

- Data System Analyzer. This tool helps the analyst to interpret the interactions of the system as a whole. The data throughput and loading of all interfaces of each software package are analyzed to ensure that the integration of systems behaves as required. A data system analyzer provides dynamic and statistical outputs relating to system data throughput and loading to enable the analyst to verify that the Development (Part I) Specification satisfies the system requirements. Automation of this analysis guarantees a rapid, accurate completion of the task [51].

- Verification Simulator. The design specifications of software are analyzed for logic and accuracy of mathematical algorithms. The algorithms are programmed on a general-purpose computer to determine that their computations are equivalent to those of the target computer. Core limitations of the target computer should be reflected in the program sizes, as determined by the Development Specification.

- Requirements Tracer. This tool records and traces system requirements to software design representations to software code. It helps the analyst in determining the completeness consistency, and compatibility of requirements [81 & 34].

- Module Interface Analyzer. This tool audits the definitions of functional module and data interfaces for compatibility and consistency. A human analyst usually extracts and formats interface information, and an automated tool audits their consistence [51]. The tools used to detect and resolve problems concerning tolerance requirements in the software design at one stage of the development process are not always used again in another and later stage. As the system software representation approaches the machine state, a different level of detail is necessary to find and resolve tolerance requirements problems. For that reason these tools and processes discussed above will not be considered in the debugging process.

3.1.2.2 Effect on Debugging Methodology

Developing software which must consider tolerance requirements may affect the debugging methodology of a system. Tolerance requirements may affect debugging tools and methods principally because the use of debugging tools and methods perturbs software execution in some manner. For example, tracing tools degrade program timing; dumping tools can affect storage; computer simulators can affect accuracy; data recording tools can affect throughput; and tools, in general, can perturb reliability evaluations. The debugging methodology should consider trade-offs in outlining alternatives in cases of conflicts or when adapting tools and methods for alleviating conflicts.

3.1.2.3 Relation to Tolerance Requirements to Software Management Methodology

The specific software management methodology used is pertinent to the development of a debugging methodology as follows:

- Trade Studies. Government regulations call for trade studies to occur early in the acquisition life cycle so that problems such as tolerance requirements design failures have less chance of occurring (i.e., hardware/software trade-offs, computer capacity and growth trade-offs). The use of trade studies early in the development process will eliminate many of the design errors which are discovered during test and debug activities. The debugging process model will assume that trade studies have been made according to AFR 800-14.

- Life Cycle Event Reviews. The SDR, PDR, and CDR of the acquisition life cycle provide checkpoint stages for evaluating the tolerance requirements design progress. For example, AFR 800-14 calls for the SDR to review the results and progress of hardware/software studies; for PDR to review implementation design of available storage, timing, and sizing, analysis of critical timing requirements and estimated runtimes; and CDR to review algorithms, storage allocation charts, computer loading, iteration rates, processing time and memory estimates. The completeness of these reviews and the quality of the inputs to these reviews are assumed to conform to AFR 800-14 for purposes of the debugging process model.

- IV&V Contractor Activities. When an IV&V process is part of the software management methodology, a number of functions are allocated to this process by Government regulations. During requirement analysis a gross functional simulation of new or critical aspects of the design may be performed to study system design, system-level trade-offs and functional interfaces. Timing and sizing studies are advised. During the design phase, it might be desirable to independently derive equations to insure correctness. A scientific simulation of the system may be produced; (i.e., algorithms are coded in a higher order language and run on a general purpose computer). However, since the IV&V contractor is not primarily concerned with debugging, the debugging process model will not address these activities.

3.1.3 Programming Language Considerations

Many of the debugging tools and techniques addressed in this report are closely tied to programming languages. These languages provide the medium through which software problems are described. Machine Oriented Languages (MOLs) provide complete control over physical machine features; the debugging tools associated with them provide information about the static and dynamic behavior of the physical machine. Higher Order Languages (HOLs) support the creation of operational and data abstractions which can be applied to data processing problems; the debugging tools associated with these languages provide information about the static and dynamic behavior of these abstractions.

In recent years a discipline of software engineering has evolved which is concerned with tools and techniques for the specification, design, and implementation of reliable software. One of the primary areas of interest has been the development of formal languages which support software engineering concepts and allow automated enforcement of these concepts. These language developments have impacted the entire software development life cycle. There are several reasons why this is so:

- The choice of a programming language strongly interacts with the effectiveness of development methodologies, such as levels of abstraction and structured programming.

- Higher order languages mask awarement of machine detail and restrict allowable control flow, allowing less complex and more compact code to be written.

- The use of higher order languages allows detection of many static errors at compile time. The compiler's run time mechanism can provide immediate detection and confinement of dynamic language violations. The degree to which compilers can detect logical and syntactic errors is dependent upon the definition and implementation of the language.

- Many stand-alone tools, which are developed to support a debugging methodology, implement capabilities which could be supported in the implementation of one of the modern programming languages. The fact that commonly used programming languages do not support these capabilities forces the development (and redevelopment) of these tools in a project specific manner.

Each of these areas will be developed more fully, with examples, in the following discussion.

3.1.3.1 <u>Languages and Development Methodologies</u>

The choice of programming language used in a development effort has a dramatic impact upon the effectiveness with which modern software engineering concepts can be implemented. While there is no implicit restriction on an assembly language programmer which prevents him from utilizing the principles of structured programming, modularity, top-down design, or levels of abstraction, the machine-level programmer receives no support from his tools to achieve these goals. Further, he receives no mechanical enforcement of the discipline needed to adhere to them. The question is not whether one can achieve the goal of reliable software using low level tools, but what is the degree of support the tools provide in attaining these goals.

The fundamental reason for the development of HOLs is to provide abstract means of describing data and operations on that data. These abstractions support the programmer's logical view of his problem and allow powerful automatic checks on the correctness of their use. As a result, many software errors can be detected during design and code activities which would otherwise escape detection until testing or operational phases. Several studies [12 & 27] have indicated that early detection of errors minimizes the cost and design impact of error correction.

The power of a compiler's capability to detect errors is dependent on the level of abstraction supported and the amount of redundancy required in the expression of a problem. Figure 11 shows a selection of programming languages and their corresponding abstract qualities.

LANGUAGE	ABSTRACTIONS
Machine	Bit patterns Machine addresses Machine operations
Macro assembler	Mnemonic low level operations Symbolic addresses Subroutines
FORTRAN, JOVIAL, ALGOL	Arithmetic expressions Primitive data types Symbolic control flow
PL/I, PASCAL	Complex data types Arithmetic expressions Closed control flow structures
CONCURRENT PASCAL, CLU, MODULA	Abstract data types Abstract operations Closed control flow structures

Figure 11. Languages and their Abstractions

It might be proper to assume that the use of the highest level language would contribute to the development of more error free programs and the early localization of errors which do occur. Experience indicates that this ideal actually is approached in practice. There remain three major problems which prevent problem solution at the level of the abstract machine.

The first problem is that compilers are implemented in software. They contain errors which cause incorrect translation from the abstract machine to the physical machine. The localization and correction of such bugs is a task requiring knowledge of the program, the abstract machine, and the physical machine.

The second problem can be summarized as follows, the behavior of a program written in an abstract language should always be explainable in terms of the concepts of that language and should never require insight into the details of compilers and computers. Otherwise, an abstract notation has no significant value in reducing complexity [15].

Although much progress has been made in the development of debugging tools which reflect problem symptoms in the concepts of the abstract language, the use of physical machine-level tools is often still required in error resolution. An outstanding example of a compiler which has been designed to permit debugging at the abstract machine level is the IBM PL/1 Checkout Compiler [26 & 105]. The Checkout Compiler offers a wide range of debugging features including:

- Display and change of PL/1 variables by name.

- Correction of source language syntax errors.
- Messages with explanations in source language terms (including variable names) in either succinct or detailed forms.
- Immediate execution of PL/1 statements entered from a terminal (except for those that change declarations or block structure).
- Temporary changes to program logic (useful for checking specific logic paths).
- Specification of the number of statements to be executed before control is returned to the terminal.
- Reformatting of source statements for readability.

The MULTICS PL/1 compiler [90] offers some of the above capabilities by retaining symbol table data at run time. It also contains limited path execution analysis capabilities.

The last problem is that the abstractions of the language as implemented by a compiler must exactly match the abstraction in the mind of a human programmer. There is a substantial possibility that the actual behavior of the computer does not match our assumptions about it. Many of the most difficult errors to detect occur when a programmer has used a language feature which actually does more (or less) than the programmer intended, or even something entirely different. This problem is best addressed within a context in which the programming language is formally defined. Such definition allows more careful analysis of what a programming construct should do, and whether it is doing it correctly. The outstanding examples of formal definitions of higher order languages are PASCAL [53], ALGOL 68 [84], and PL/1 [90].

One of the foremost new development methodologies having implications for debugging and language choice is formal verification. The problems encountered in an attempt to formally prove a program correct are in some ways similar to the problems encountered in debugging. A programmer faced with the refusal of an automated verification system to verify his program must ask himself (among other questions): "How could this program fail to achieve its desired intent?" This is basically a debugging question. Formal verification has been successfully demonstrated for small programs. The majority of the current research effort to extend the size of verifiable programs includes the development of high level languages designed for formal verification, such as EUCLID, CLU, and ALPHARD. Successful application of formal verification techniques to production programs would aid substantially in debugging logical errors in pre-integration development phases. However, the current state-of-the-art of formal verification does not support this approach.

3.1.3.2 Languages and Program Complexity

The implementation of a software system can exhibit two kinds of complexity: first is the intrinsic complexity of the problem being solved; second is the complexity of expressions required to implement the features of the chosen algorithm. General purpose programming languages have little impact on intrinsic problem complexity, but they can contribute substantially to reducing the second kind of complexity. It is common to view higher order languages, together with operating system features, as implementing an abstract, or virtual, machine. This means that the programmers' view of the computing environment is an idealized one, free of concern for details of physical machine architecture, storage management, or process scheduling. The solution to a problem is formulated with the features of this ideal machine; the management of machine specific detail is assumed by the system software. The most advanced of the current generation of programming languages allow the programmer to, in effect, create his own abstract machine by defining abstract data types and abstract operations on these types. Instances of these types are then manipulable by procedures with appropriately defined scope privileges. Some programming languages (e.g., CONCURRENT PASCAL, CLU, and SIMULA) allow programmer definition of data types and procedures which manipulate these types. The data and procedure definitions are termed abstract data types and are bound together in a module called a class or a monitor. Instances of these types may then be declared in a program, and the only legal access to the data of the abstract data type is by invocation of one of the associated procedures. An example of an abstract data type might be a buffer used to communicate data between two processes. The only operations defined on the buffer might then be send and receive. The use of abstract data types to define an abstract machine can contribute to diminished complexity in the expression of problem solutions.

The contribution of HOLs to the lessening of the complexity of debugging is also important. In machine language, the smallest mistake can cause any instruction to destroy any other instruction or variable. Here, the whole memory can be the interface between any two instructions. This was made only too clear in the past by the practice of printing the contents of the entire memory just to locate a single programming error. Programs written in languages such as FORTRAN, ALGOL, and PASCAL are unable to modify themselves. Instead, they will have broad interfaces in the form of global variables that can be changed by every statement, either by intention or mistake.

3.1.3.3 Languages and Error Detection

HOLs provide an opportunity to identify errors in programs which might otherwise escape detection until the integration and operation phases of program development. These capabilities are notably absent from assembly language programs, and are not widely implemented in currently used HOLs. The basic problem of compiler design is that an implementation must correctly implement legal programs and report errors in illegal programs. In the past, most efforts concentrated on the translation of legal programs. The problem of detecting and reporting illegal program constructs has been more elusive, both because of greater complexity and the lack of formal definition of legal language features.

Two types of error detection techniques can be used by a language system; predictive and confining. A predictive technique allows a property of a program to be determined without reference to the exact data upon which it will operate. These static checks are commonly performed at compile time.

Examples of predictive error detection techniques include strong type checking (as in PASCAL and ALGOL 68), compile time checking of formal versus actual parameter lists, enforcement of variable scope rules, and built in synchronization primitives for abstract data types. Although not performed at compile time, formal program verification can be viewed as a predictive technique which proves that a program has certain properties by using the static properties of its description. Predictive techniques detect errors during the coding phase of program development; they can 'filter' many interface problems that would otherwise escape detection until testing or operations. This allows concentration of the integration effort upon surfacing logical or performance problems.

A confinement technique prevents a program from employing a machine in such a way that the machine does not legally implement the language. Using a confinement technique for a part of a system guarantees either that the part will always function as specified or that a malfunction will be detected. Confining techniques are usually implemented as dynamic checks which are performed at run time. Modern run time confinement techniques, such as array bounds checks and capability based protection systems, may not surface a problem until integration or operations, but they will suspend operation at the program state in which the error occurred. This prevents delayed recognition of the problem and destruction of symptoms after it has occurred, which greatly aids debugging.

3.1.3.4 Languages and Debugging Tools

Many of the debugging tools which have been developed and described in the literature represent installation specific extensions to vendor supplied capabilities. Typical examples of such tools include extended cross reference tables (module level set/use data), interface consistency analyses, formatted dumps, and interactive debug packages. (Description of the general capabilities of such tools will be found in Paragraph 3.3.) The development of debug tools in a project specific environment suffers from several major drawbacks, including:

- Project schedule and resource limitations limit the scope of such efforts.

- Many of the most powerful debug aids, such as interactive symbolic trace and dump packages, have substantial impact on compiler design. Many of the most powerful debug compilers produce interpretive code.

- No standardized programmer interface exists, forcing the learning of new protocols for each project.

Careful review of the literature suggests that there is very little substantive variation in debug package capability. Most articles describe similar sets of capabilities implemented for a specific language, computer, or operating system. A software engineering discipline should include debugging assistance and the following points should be considered:

- Debug tools should be a factor in computer selection analysis.
- Debug tools should be considered at a high organizational level, allowing the costs of tool development to be spread over many projects.
- Debug tools should be a factor in language design analysis.

An example of an integrated language/tool environment has been developed at General Research Corporation [75]. This system accepts programs written in dialects of FORTRAN or PASCAL. It includes tools for assertion checking, interface and units consistency checking across multiple compilations, path execution analysis, and limited support of formal verification in an integrated package.

3.1.4 Interactive vs Batch Systems

All manufacturers of large scale computers, and many manufacturers of minicomputers and microcomputers, now support some form of interactive assistance to the debugging process. This support may extend to all levels of programming, from MOLs to HOLs. Most of the tools discussed in Paragraph 3.3 are available in some form as interactive tools. Reports of current industrial practice indicate that the use of interactive tools decreases the turnaround of debug jobs, and thus increases the number of debug attempts per day. It has been noted that this increased availability of computer resources can encourage what has been termed laziness on the part of debugging personnel, i.e., throw it at the machine and see if it breaks. F. Brooks notes [17]:

> "Return to the instant-turnaround capability of on-machine debugging has not yet brought a return to the preplanning of debugging sessions. In a sense such preplanning is not so necessary as before, since machine time doesn't waste away while one sits and thinks.
>
> Nevertheless, three times as much progress in interactive debugging is made on the first interaction of each session as on subsequent interactions. This strongly suggests that we are not realizing the potential of interaction due to lack of session planning. The time has come to dust off the old on-machine techniques."

Interactive debugging systems are well developed for program development; however many systems, particularly those with real-time constraints, can not be designed so as to provide interactive debugging of a system undergoing integration testing. Unless the system is explicitly designed with provisions for debugging-oriented interactive use, most problems encountered in integration testing must be debugged with post-mortem data, or special runs reconstructed to display errors in the interactive environment.

3.1.5 System Architecture

This discussion evaluates the impact of system architecture on the debugging process, including an evaluation of the tools and techniques available for various size hardware systems. The impact of system architecture in the areas of real-time systems and distributed networks of processors will also be evaluated.

The rapid pace of technical development by computer manufacturers has blurred the physical performance margins between large scale computers and mini-computers; and between minicomputers and microcomputers. The level of support provided by system software of various size hardware systems greatly impacts the debugging process. There is an estimated 10-year gap in the level of sophistication between the software support offered by manufacturers of large scale machines and that offered for minicomputers. A similar gap separates mini and microcomputers. The reasons for these gaps are due to both economic and technical factors, as discussed below:

- Vendors of large scale machines have had a longer period of time to evolve a dialogue with and meet the needs of their user communities. The users of large machines expect and receive a high degree of vendor support. Vendor investment in sophisticated system software, including diagnostic features for compilers and debug tools, is substantial and not easily duplicated by mini and microcomputer vendors.

- Debugging tools and compilers generate system overhead which can normally be tolerated in a large multiprogrammed environment. This overhead may not be acceptable in a small machine environment, due to timing or memory constraints.

- Users of minicomputers tend to be technically sophisticated and willing to develop their own extensions to vendor-supplied software. The lower cost of minicomputers allows programmers to operate their programs in a hands on environment. Since minicomputers have begun to approach large scale machines in computing power, market forces have begun to support high level system software features for minicomputers.

3.1.5.1 Large Scale Machines

Due to their longer period of development history and to the size of their resources, large scale machines generally offer the most powerful debugging tools available. These tools require substantial investments of time, effort, and money for both development and proficiency in use. They generally induce inefficiencies in utilization of time and storage space, but these limitations have minimum impact on large systems. Although many military systems, which are hosted by large machines, have resource constraints, these machines can generally support the inefficiencies of powerful debugging tools during program development.

3.1.5.2 Minicomputers

The range of debugging tool capability is limited for minicomputer users by economic and resource issues. The tools which are available tend more towards the traditional set of machine level dumps, traces, and breakpoints.

3.1.5.3 Microcomputers

Self resident debugging tools for microcomputers are limited to primitive store inspection capabilities. Microcomputer program development systems have developed test and debugging techniques in which a separate processor (micro or mini) is used to provide monitored data of microprogram operation. Extensive use is also made of interpretive computer simulation in which the execution of a program under development is simulated on a large scale machine. This technique allows collection of a wide variety of debugging information.

3.1.5.4 Distributed Systems

This discussion is limited to multiple CPU networds in which control is architecturally distributed. Multiprocessor systems operating from shared memory with a common operating system do not differ from other large-scale systems insofar as debugging is concerned. The fundamental debugging concern for distributed systems is the collection of interprocessor information flow in such a manner as to be able to diagnose or reproduce faults.

3.1.5.5 Real-Time Systems

A major objective of this study is to identify the debugging requirements of real-time systems. Real-time systems are defined to be those computer-based systems which satisfy either or both of the following constraints.

- The data processing system exercises no control over the rate at which its input data arrives.

- The data processing system is required to produce a response to conditions in the input data within a fixed, relatively short, period of time.

The basic characteristic of real-time systems is that they directly interact with real world phenomena. The amount of additional functional and performance complexity in real-time systems operation is considerable. In addition, one must often deal with concurrent processes which do not exhibit reproducible behavior. The major debugging problems introduced in the implementation of real-time systems include:

- Increased complexity
- Problem symptom data collection
- Error reproduction

The major tools and techniques which have been developed to meet the above problems are:

- Data capture tools
- Simulation
- Design techniques

3.1.6 Integration Concepts

Since this debugging study is primarily directed at the integration phase of software development, the concepts of integration are an essential element of the debugging environment. This discussion describes those concepts.

In beginning, it is important to distinguish between design practices and development practices. The terms "top-down design" and "top-down development", for example, pertain to two totally separate activities. The design activities are concerned with planning how the software will work, while the development activity deals with implementing this design. In addition, the design approach used does not necessarily imply a given implementation approach. A software system, for example, might be designed in a top-down manner, but developed in a bottom-up manner. The primary emphasis of the literature survey associated with this study was devoted to the development concepts which affect integration activities, rather than design concepts. There are some important debugging considerations associated with the design concepts, however, so these will be discussed in the following paragraphs.

With this understanding, the various concepts of integration are discussed in the following paragraphs in terms of:

- Top-down design and structured programming
- Bottom-up development
- Top-down development

3.1.6.1 Top-Down Design and Structured Programming

Top-down design and structured programming are related to debugging in the integration phase for several reasons, including:

- The discipline and organizational aspects of these practices are intended to lead to more simple or basic interface relationships [92 & 5].

- Top-down design is particularly suited to establishing program module interfaces [63].

- These practices are widely regarded as contributing strongly to system understandability. Top-down design and structured programming are powerful tools which aid in learning, or relearning, the intended purpose and logic of a program. This learning process is a fundamental component of debugging. The fact that these practices contribute to early an deeper insight into program operation lessens the possibility of a program fix introducing new errors into the software [14 & 22].

A bottom-up design approach probably has never really been performed in practice. The reason this approach has been such a topic of discussion is that it probably has not been distinguished from bottom-up development.

3.1.6.2 Bottom-Up Development

The bottom-up development approach codes the lowest level processing programs first. Each unit is tested and then made ready for integration. A unit test is a test of a single module in an isolated environment. Figure 12 depicts this development process [46].

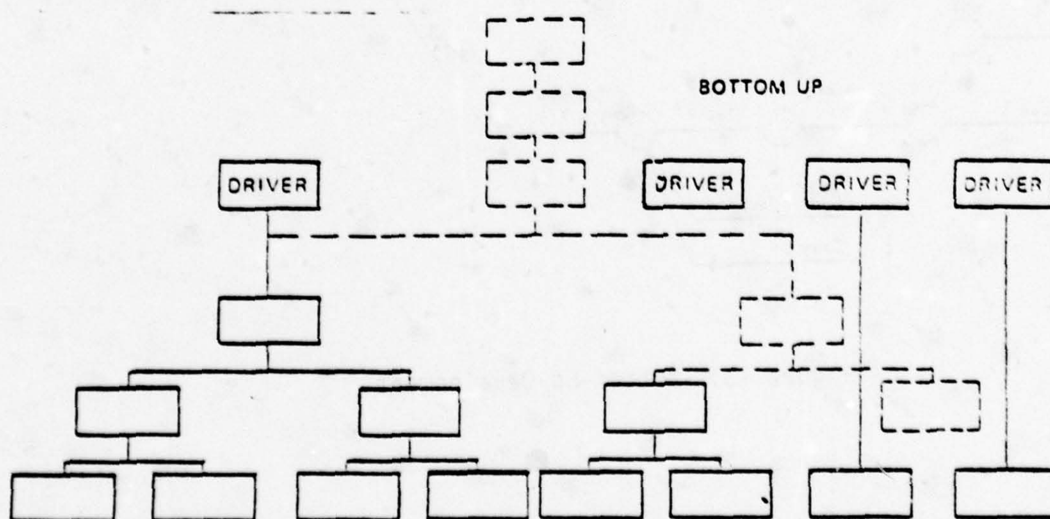


Figure 12. Bottom-Up Development (Worst case)

Bottom-up development requires that code in the form of driver programs is needed to perform unit test and, usually, CPC integration testing. In practice, the bottom-up approach can be implemented via scheduling techniques to reduce the need for drivers as much as possible. Figure 13 illustrates an improved use of the approach, showing the design tree and a PERT chart of a typical bottom-up development. However, in this process, the only modules that are strictly unit tested are the modules with no subordinates. Modules that have subordinates are not tested alone; they are tested with their subordinates [63].

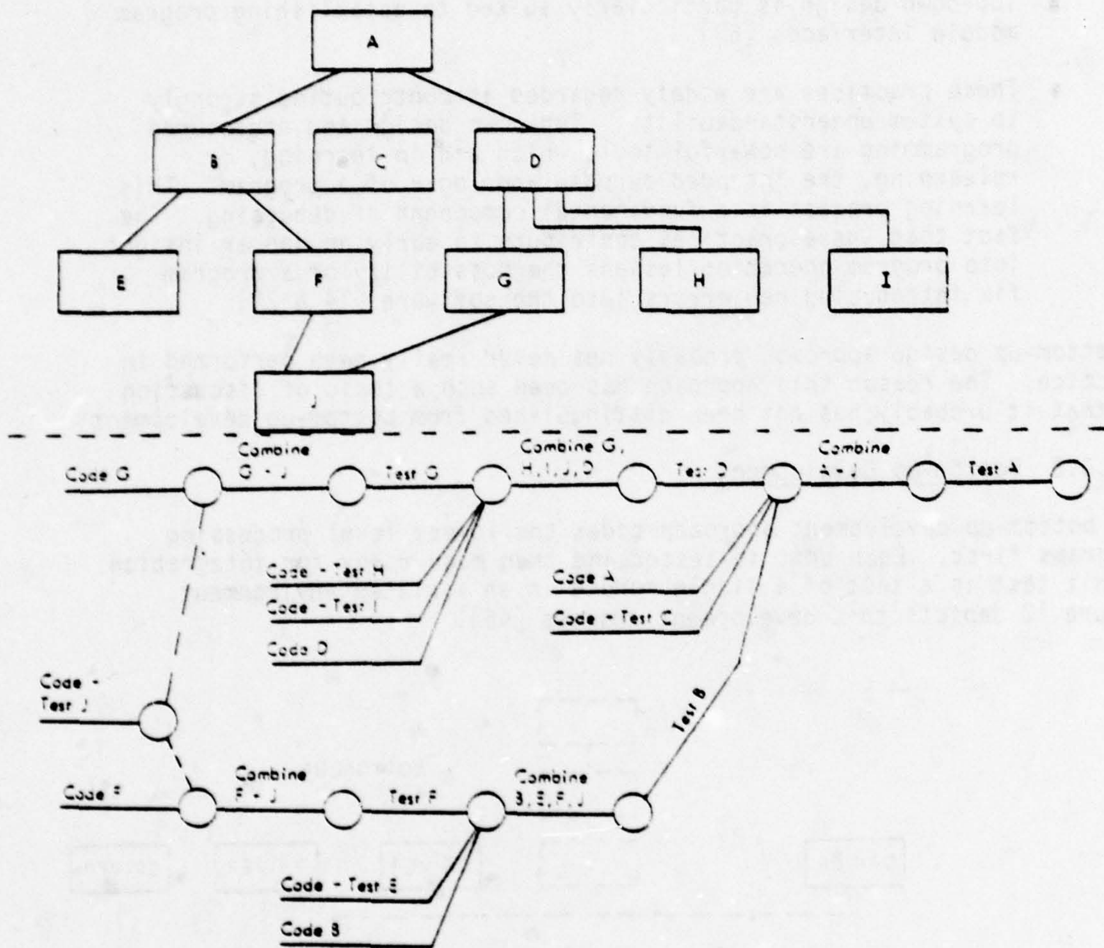


Figure 13. Bottom-Up Development

3.1.6.3 Top-Down Development

In top-down development, coding is performed top-down, in execution sequence. That is, the module at the top of the structure is coded first. Then, modules subordinate to this module are coded, combined, and tested together. The next level is coded, combined, and tested, etc. Figure 14 illustrates this approach [89].

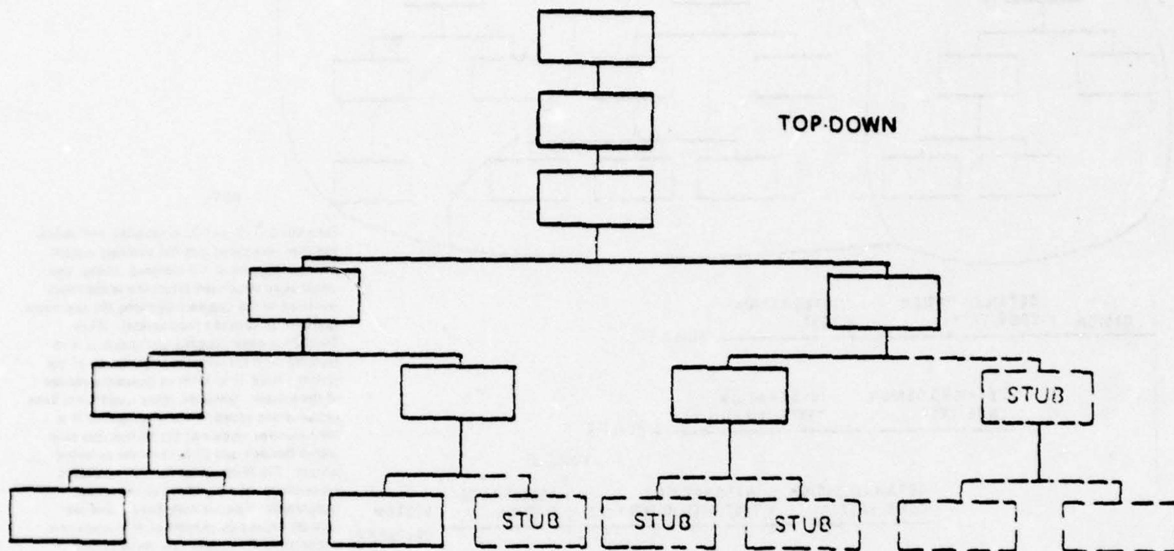
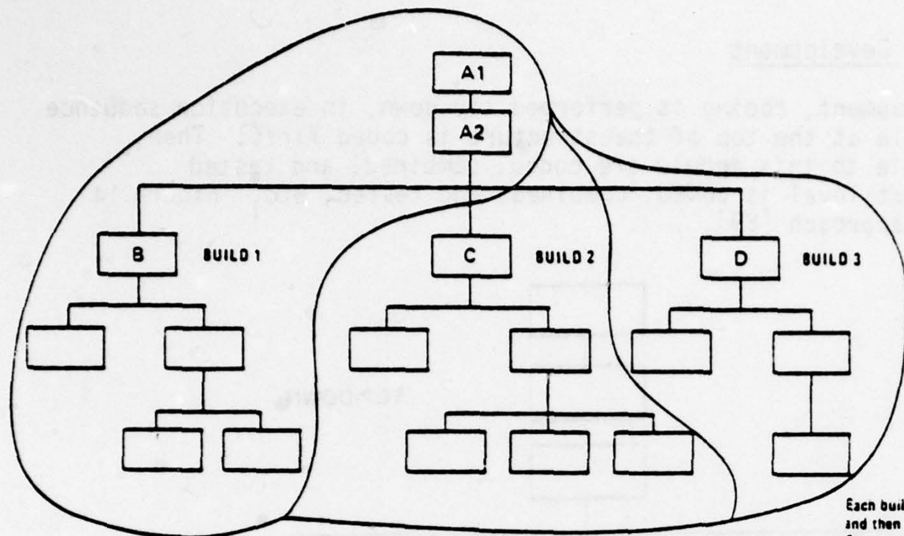
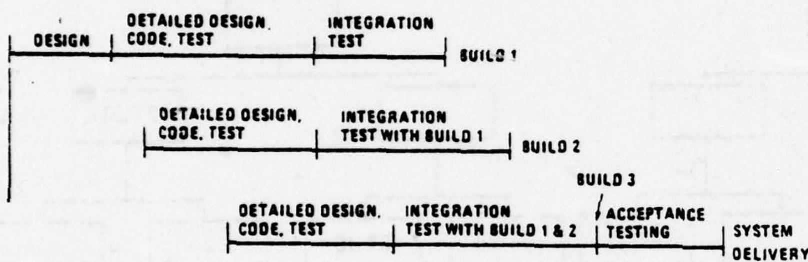


Figure 14. Top-Down Development - Worst Case.

The top-down approach often requires code in the form of stub programs because routines which are called might themselves require lower level routines which are not yet coded. In the optimum top-down development, CPC testing is performed within the developing total system in a continuous integration fashion. This process, known as the "build concept," simplifies the test materials problem since the need for stubs is reduced and most inputs will come from other processes. The build approach involves the successive implementation and demonstration of logically complete subsets (builds) of the total system requirements, each build serving as a basis for the next. Although implementation proceeds from a single starting point, it does not imply that the implementation must proceed down the hierarchy in parallel. Some branches are intentionally developed earlier than other branches. For example, user or other external interfaces might be implemented first to permit early demonstration of software capabilities, partial software evaluation, training, or even incremental software acceptance [46]. Figure 15 illustrates the build approach. Some overlap in implementing successive builds is usually necessary to meet schedule requirements.





NOTE

Each build (1, 2, and 3) is designed and coded and then integrated into the evolving system. For example, Build 1 is designed, coded, and tested as an entity and forms the initial representation of the system (following the top-down approach to system development). When Build 2 has been designed and coded, it is integrated with the initial representation of the system (Build 1) to form an expanded version of the system. Similarly, when Build 3 has been designed and coded, it too is integrated into the expanded version of the system (the integrated Builds 1 and 2) to form the complete system. The three builds may be developed sequentially or in parallel, but the system representation begins with Build 1 and subsequent expansion consists of the integration of the other builds into this initial system representation.

Figure 15. Implementation by Builds

The basis for grouping requirements into builds may be one or more of the following:

- Successive levels of capability in every area of the total system requirement.
- Provision of functions to support a certain increment of hardware or environmental capability.
- A meaningful set of functions which becomes so well defined during the requirement specification process that it can be designed and implemented while the analysis of the remaining requirements continues.
- Each build contains some increment of demonstrable functional capability [115].

3.1.6.4 Relation of Integration Concepts to other Debugging Considerations

The effect that the previously discussed debugging considerations have on integration concepts pertinent to the debugging methodology development are as follows:

- Software Management Methodology. This consideration has no unusual effect on the integration concepts, aside from Government regulation of internal standards and conventions.

- Software Tolerance Requirements. This consideration has no unique relation to the integration concepts.

- Language Considerations. This consideration has no unique relation to the integration concepts.

- Interactive vs Batch Systems. This consideration has no unique relation to the integration concepts.

- System Hardware/Software Architecture. The bottom-up and top-down development approaches are applicable to large-scale, real-time, minicomputer, and distributed systems. Microcomputer systems development is unique in that simultaneous hardware and software development occurs. Figure 16 shows a typical microcomputer system integration cycle for a small system (the numbers represent duration in days of each activity).

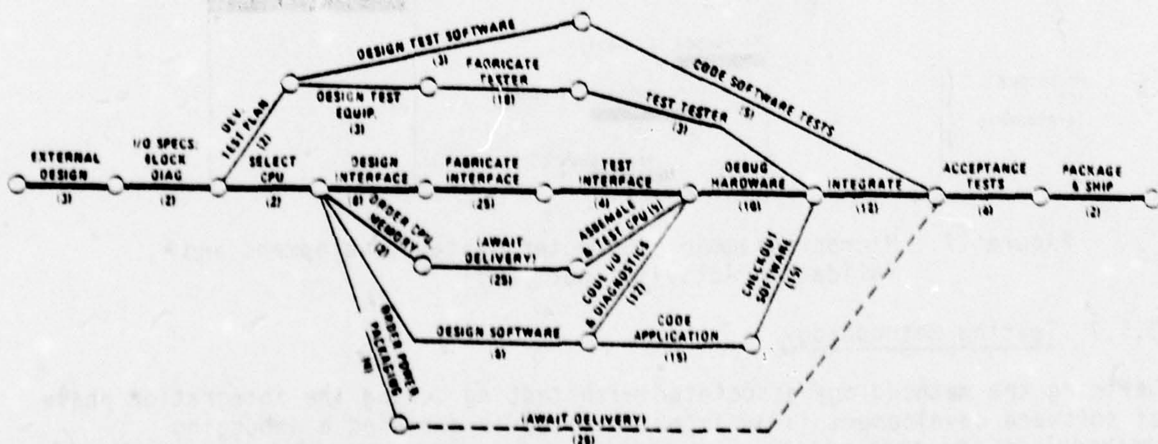


Figure 16. Typical Development Cycle for Small Microcomputer System [64].

Simultaneous hardware and software development is possible through the use of such software testing and debugging tools and techniques as the Interpretive Computer Simulation (ICS). In this case, the microcode is developed, integrated, and tested on a simulated version of the microcomputer before hardware fabrication is complete (see Paragraph 3.1.8.3 for a description of the ICS technique). For example, Figure 17 shows the development and validation activity chart for Logicon's validation of the microprograms of the Brassboard Fault-Tolerant Spaceborne Computer (BFTSC) build for the Air Force Space and Missile Systems Organization (SAMSO) [88].

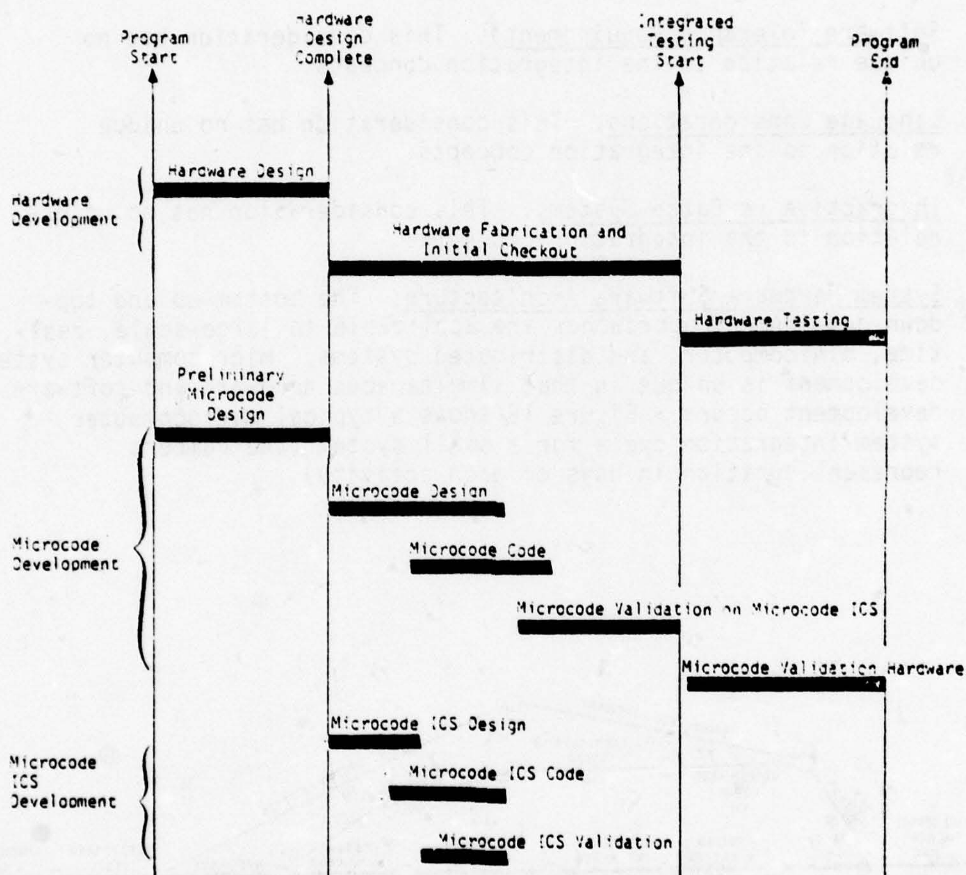


Figure 17. Microprogrammable Computer System Development and Validation Activity Chart [88]

3.1.7 Testing Methodology

Defining the methodology associated with testing during the integration phase of software development is an important step in deriving a debugging methodology for that phase. The testing methodology provides the background on which debugging is conducted. It influences the kinds of errors encountered, the point-in-time within the development process when they are uncovered, and the personnel involved in the debugging process.

The chief difficulty in attempting to define a debugging methodology is the diversity of approaches and terminology currently in use in the field of testing. Figure 18 shows the overlap of terms applying to testing methodology. An "X" opposite two terms means the terms are used to describe the same, related, or overlapping testing activities. Although the list of terms in Figure 18 is not comprehensive, it points out the difficulty of trying to equate the various test approaches and terms concurrently in use.

The remainder of this discussion presents a consolidated description* of the testing methodologies found in the literature survey. The following discussion is compatible with AFR 800-14, and is presented in terms of:

- Computer program verification
- Independent computer program verification
- The relation of testing methodology to other debugging considerations

The methodology discussed does not attempt to satisfy every existing software system but is intended to provide a meaningful and substantial background for development of a debugging methodology.

3.1.7.1 Computer Program Verification

Of the three testing and evaluation functions described in Paragraph 3.1.1.2, the function which applies to the debugging study (i.e., the development integration phase) is verification. Within the verification function, the following activities occur involving the integration process:

- Informal Testing of the CPCI and Its Components. CPT&E is carried out by the contractor, at his discretion, to support his development activities, provide visibility of progress, and prepare for formal testing.
- Formal Testing of the CPCI. PQT and FQT is carried out by the contractor in accordance with Air Force-approved test plans and procedures to verify that the CPCI fulfills the requirements of the Development (Part I) Specification and to provide the basis for CPCI acceptance by the Air Force.

*The testing methodology is based primarily on a guidebook for the verification process, prepared by System Development Corporation [13].

	VERIFICATION	VALIDATION	UNIT	CPC	MODULE	CPT&E	COMPONENT	ASSEMBLY	CONSTRUCT	FUNCTION	INTEGRATION	CPCI	BUILD	CAT I	PRELIMINARY QUAL	FO
Verification	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X
Validation		X														X
Unit	X		X	X	X	X	X									
CPC	X		X	X	X	X	X									
Module	X		X	X	X	X	X									
CPT&E	X		X	X	X	X	X									
Component	X		X	X	X	X	X									
Assembly	X							X	X	X						
Construct	X							X	X	X						
Function	X							X	X	X						
Integration	X										X	X	X			
CPCI	X										X	X	X			
Build	X										X	X	X	X	X	X
CAT I	X												X	X	X	X
Preliminary Qualification (PQT)	X	X											X	X	X	
Formal Qualification (FQT)	X												X	X		X
Acceptance	X												X	X		X
Milestone 5 Testing	X												X	X		X
CAT II		X														
System DT&E		X														
System Integration		X														
Performance		X														
Environment		X														
OT&E																
Operational Demonstration																
Maintenance Testing																
Static Code Analysis	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X
Code Execution Testing	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 18. Overlap of Terms Applying to Testing Methodology

INTEGRATION	CPCI	BUILD	CAT I	PRELIMINARY QUALIFICATION (PQT)	FORMAL QUALIFICATION (FQT)	ACCEPTANCE	MILESTONE 5 TESTING	CAT II	SYSTEM OT&E	SYSTEM INTEGRATION	PERFORMANCE	ENVIRONMENT	OT&E	OPERATIONAL DEMONSTRATION	MAINTENANCE TESTING	STATIC CODE ANALYSIS	CODE EXECUTION TESTING
	X	X	X	X	X	X	X									X	X
					X			X	X	X		X	X				X
																X	X
																X	X
																X	X
																X	X
																X	X
																X	X
																X	X
																X	X
	X	X														X	X
	X	X														X	X
	X	X	X	X	X	X	X										
		X	X	X	X	X	X										
		X	X	X													
		X	X		X	X	X										
		X	X		X	X	X										
		X	X		X	X	X										
								X	X	X	X	X					
								X	X	X	X	X					
								X	X	X	X	X					
								X	X	X	X	X					
								X	X	X	X	X					
													X	X			
													X	X	X		
	X	X	X	X	X	X	X									X	
	X	X	X	X	X	X	X										X

The entire process of CPCI verification is the reverse of design where analysts start from a global definition of the system and proceed with successive layers of detail, finally resulting in a detailed CPCI design from which coding activities may be initiated. CPCI verification, on the other hand, usually proceeds from (1) the detailed-CPC level in a simulated environment, to (2) the execution of a small increment of functionally-related CPCs, to (3) the operation of all CPCs, together in a live, or nearly live, environment. Top-down programming calls for a variation of this method whereby key control and input handling programs are developed and tested first.

The top-down philosophy calls for CPC implementation to be planned to avoid simulated inputs, where possible. The structure of the entire CPCI is initially represented by stubs which (1) contain very brief non-functional code or (2) may simulate each CPC's operation by performing abbreviated functions. The stubs are replaced as each coded CPC becomes available. The following two test processes, CPT&E and preparation for PQT and FQT are described in detail below. These two testing processes form the basis on which the Software Debugging Process will be structured. This material has been adapted from the ESD Verification Guidebook and other current sources [13,47,10,62 & 70].

Computer Program Test and Evaluation (CPT&E)

CPT&E consists of CPC code and test, CPC incremental-integration testing and CPCI testing. CPT&E is the contractor's CPC/CPCI-design shakedown testing. The incremental coding and testing activities of CPT&E may span nearly the entire Full-Scale Development Phase, overlapping with PQTs and terminating when the contractor has completed his internal CPCI testing and is ready for FQT.

The process of translating the software design into executable programs is a multi-step operation using many implementation test tools and techniques. The emphasis on the contractor's work during CPT&E is not immediately directed at verification of performance criteria, but instead, at implementation of the software design. At this point, the design has been shown to meet the specified performance standards. Since performance criteria are a result of analyses of operational requirements and the proposed design has been correlated with the performance criteria, CPT&E verification activities are primarily directed at determining that the programmed instructions are accurate, consistent, and compatible with the detailed computer program draft Product (Part II) Specification.

The first activity of CPT&E is CPC code and test. This activity consists of CPC coding and CPC testing. CPC coding is the translation of the technical solution of a particular problem into a set of machine-readable instructions for the performance of specific computer operations. CPC testing (also referred to as unit, module, component, subprogram, or parameter testing) is that testing performed by contractor personnel and directed at assuring the internal accuracy and consistency of each CPC before integration with other functionally-related CPCs. CPC testing begins with each module or unit of code and continues until the entire CPC is developed and tested. Specifically, each CPC must be tested as a unit to verify that:

- All possible inputs to the CPC are correctly interpreted.
- Arithmetic and logical functions assigned to the CPC are correctly processed.
- Coding conventions and standards are incorporated in the implementation of the CPC.
- Outputs are correct and consistent with the input data.

C testing activities include:

- Preparing test data
- Compiling or assembling the CPC and reviewing the outputs
- Running the test data
- Examining test run results
- Identifying and correcting errors
- Repeating each CPC-level testing step until the CPC operates as the programmer's design intended.

PC testing may also identify requirements for modifications to increase fficiency and maintainability, to meet coding conventions and standards, and o change the program when testing or the contractor's internal audit rocedures indicate that program quality is unacceptable.

PC debugging consists of extracting syntax and logic errors, or "bugs", from he software. During CPC testing, each area of code is tested with sample, xtreme, and illegal (out of range) data values to ensure that the code perates as it was designed. Early stages of debugging rely heavily upon the rogrammer's desk-checking of computer-produced listings, despite tool vailability. One essential quality of debugging tools at this stage is hat they assist the testing process without requiring insertions of large mounts of code. Debugging code insertion may:

- Significantly alter CPC performance
- Generate additional errors because of the additional code
- Hide pre-existing bugs until the added code is removed

The programming methodology used by the contractor impacts the selection of test tools used. For example, top-down development reduces the need for test drivers, whereas bottom-up development generally requires more test drivers.

Testing on a CPC level is complete when all necessary tests have been executed without error and there is demonstrable evidence that the problem/solution algorithms, data set/use, and subprogram interfaces are complete and correct. The quantity and quality of the tests used for CPC testing are highly dependent upon the contractor's internal test effort and upon the individual programmer's approach and habits. The test data used for CPC testing are derived from analyses of internal design specifications and simulation of the CPCI's environment. Sometimes a contractor uses an independent programmer or test team for the testing of each CPC. This approach is more often used in later stages of testing, specifically CPCI testing in preparation for PQT and FQT.

CPC incremental-integration testing begins after the successful completion of CPC testing. CPC incremental-integration testing (also known as construct or assembly testing) is directed at resolving design, logic, data definition, and interface errors existing in the combined operation of two or more CPCs. CPC incremental-integration testing focuses on:

- A sequential integration of functionally-related CPCs
- Using outputs of one CPC as inputs to the next
- Verifying that CPCs operate as designed and according to performance requirements
- Conducting dry runs in preparation for PQTs

Generally, the CPCs are integrated by combining functionally-related CPCs. In that way, meaningful test case data relating to a specific function can be generated by the CPCs instead of test drivers. CPC integration proceeds by incrementing large numbers of CPCs to provide the input, processing, and output functions needed for complete testing. This approach results in efficient test-case generation.

In top-down implementation, CPC code and test are accomplished together with CPC incremental-integration testing. Using this approach, the programmer tests each CPC by linking it to the previously developed higher-level CPCI structure. Necessary lower-level logic may be provided by the use of stubs. In this manner, outputs from a tested CPC are used to provide test inputs for the CPC undergoing test. This method increases the testing applied to key CPCs in an environment which closely approximates the intended operational environment.

Individual CPC testing should verify that each function was executed correctly with input data. These data are first set to single values, followed by a wide range of values. Similarly, the integrated CPCs should be tested to verify that all functions perform correctly for both single and multiple data values.

CPCI testing or integration testing is a contractor dry run of an FQT. It is a necessary part of CPT&E because it completes the iterative process of testing, correcting, and retesting. If individual CPC and CPC incremental-integration testing has tested to design limits, CPCI testing enhances management's confidence that the FQT will be passed successfully. Initially, CPCI testing focuses on verifying the total CPCI design. However, the major effort shifts toward verifying that the CPCI, including all its related components, satisfies the requirements of the Development (Part I) Specification.

The purpose of CPCI testing is to verify that all the components of the CPCI interface together to perform their required functions, while not exceeding the limits of tolerances and qualification criteria. The test plans, procedures, and data used during the contractor's CPCI testing should relate to those submitted for PQTs and the FQT. However, the test plans, procedures, and related data used in CPT&E are usually not deliverable or available for scrutiny, unless contractually specified.

The test tools used during the contractor's CPCI tests are the same as those used during CPC incremental-integration testing. However, the use of such tools should be limited since the intent of CPCI testing is to ensure CPCI performance in the operational environment. It may still be necessary to use simulated input data generated by tools and processors to record, analyze, and reduce output data. It may also be necessary to use hardware, firmware, or software emulation techniques for replicating machine or software functions currently unavailable to the CPCI. It is important that CPCI testing be directed at verifying CPCI performance, not perturbed by the overhead of test tools.

Qualification Testing

Qualification testing is the formal*, contractor CPCI testing witnessed by the customer. It consists of the preparation and execution of PQT(s) and FQT(s).

Preliminary Qualification Tests

PQTs (also referred to as verification tests) are planned, scheduled, and performed by the contractor at his development facility. They are intended to provide visibility into work progress and to demonstrate that the design meets its performance requirements. PQTs are conducted in accordance with approved test plans and procedures, and test reports submitted in accordance with the Contract Data Requirements List (CDRL). PQTs are generally scheduled during contractor CPT&E on a sequential basis, often corresponding in sequence to the reviews of design and implementation of builds. Each PQT is designed to demonstrate the performance capabilities of a group (increment or build) of functionally-related CPCs.

*"Formal" testing is that portion of CPCI testing which is conducted in accordance with approved test plans to verify that the CPCI fulfills requirements of the Development Specification.

PQTs are intended to provide management visibility into performance critical CPCs, not for their qualification. Overly detailed testing, reflecting CPCI design, structure, and internal operation does not generally provide visibility and may even obscure appraisal of technical adequacy.

The PQT differs from the FQT in two major areas, as follows:

- PQT test coverage may be more detailed and the test results may include intermediate processing data, i.e., data communicated between CPCs, but not required output of the CPCI. An entire range of data values may be used for a specific parameter to demonstrate functional/error processing for illegal values.
- PQTs are conducted at the contractor's development site and may include only minimal hardware/software interface testing. They may also use the contractor's CPT&E test tools and techniques (i.e., simulated input data, emulated hardware or software, and output data processors).

Formal Qualification Test

FQT (also referred to as acceptance testing) is a comprehensive test of the integrated CPCI. It is performed by the contractor to verify that the CPCI meets the performance requirements as stated in the Development (Part I) Specification. FQTs are conducted in accordance with approved test plans and procedures, generally with qualified operationally-configured equipment. FQT normally takes place at a location providing the required equipment capability or at the system DT&E site. It should be complete prior to the beginning of system DT&E. If the required equipment configuration is not available, or if performance requirements cannot be verified in the CPCI DT&E environment, it is usually so stated in the Development (Part I) Specification. The requirements are then qualified in the system DT&E environment and the FCA for this CPCI is supplemented by Formal Qualification Review (FQR) of the CPCI. For CPCIs that are not dependent upon total system availability (e.g., support packages), qualification testing is usually conducted at the contractor's site. However, the computer configuration used for qualification at the contractor's site should be sufficiently similar to the operational configuration that no doubts remain about CPCI qualification.

3.1.7.2 Independent Computer Program Verification

Another important aspect of testing methodology is the possible existence of an IV&V whose testing parallels the contractor's internal and qualification testing. The essential difference between IV&V and a development contractor's formal test group is managerial. IV&V is performed by independent personnel employing independent test tools and techniques performed at independent sites or facilities [47]. These parallel testing activities can span the CPC incremental-integration, CPCI, PQT, and FQT testing [10]. It usually involves the use of the static code analysis, dynamic monitoring, and code execution test tools described in Pargraph 3.1.8. An IV&V contractor may affect debugging as follows:

- The IV&V contractor (especially with regards to testing tools, sites, and facilities) can add complexity to the debugging process. Problem recreation and resolution might be more difficult because of communication problems between the IV&V and the development contractors.

- An IV&V contractor is primarily concerned with finding problems rather than debugging them.

- The IV&V contractor is isolated from the development contractor. He is usually not aware of the most current subtle design changes taking place, or of the most current software configurations. Often, reported problems have already been fixed, or a problem is non-existent because the design has been subtly changed.

3.1.7.3 Relation of Testing Methodology to other Debugging Considerations

The effect that other debugging considerations have on the testing methodology is described in the following paragraphs.

Software Management Methodology

A software management methodology influences the testing and debugging process. It establishes the formal structure of the testing to be conducted. For example, AFR 800-14 calls for the CPC, CPCI, PQT, and FQT levels of testing and CPCI audits. The absence of this usually results in the absence of some of the components of development testing just described. This, of course, affects the debugging methodology in relation to when, within the development cycle, certain problems are encountered. In addition, software management methodology also establishes the internal and external configuration control procedures which are applied during the CPC code and test, the CPC Incremental-Integration, CPCI, PQT, and FQT testing periods. This affects the identification and communication of problems to be debugged, the ability to guarantee the proper environment for problem duplication, and the monitoring of problem status. The contractor's internal-change control procedures are important to disseminate problem status accounting to personnel. Furthermore, the requirements for test documentation and audits result in a more controlled testing methodology upon which debugging planning steps (such as tool development) can be based.

Software Tolerance Requirements

Timing and sizing tolerance requirements, in particular, are applicable to CPC integration testing. Although timing and sizing analyses provide estimated values early in the Validation Phase, the CPC integration testing activities provide the first opportunity for the contractor to collect functional performance data related to timing and sizing. Sizing data, at this time, should be fairly accurate. However, timing data are still rudimentary because test inputs are often generated by the computer and timing does not yet reflect the operational environment. The contractor can compare these results with his earlier analyses predictions to confirm accuracy or to identify potential problem areas. Since timing and sizing problems often require expensive and time consuming hardware or software redesign solutions, they need to be identified as early as possible [13].

Language Considerations

In most software development efforts, the language compiler used is a mature compiler (i.e., reliable) which does not contribute to debugging difficulties by producing subtle, language-to-machine translation problems. However, for some of the newer avionics systems, containing small innovative computers [35] or new microcomputer systems [31], the use of a cross-compile on a larger machine is required. A new cross-compiler's translation of a programming language to machine code instructions for the new computer can easily have the most dreaded subtle kinds of errors. In cases like these, the testing methodology must establish a framework for testing that results in the detection of support software problems well before testing of the new application software begins. Testing should include testing tools that use object code, as well as source code. The debugging methodology on the other hand, must anticipate the short-comings of the testing methodology and consider the tools and techniques needed (i.e., a debugging tool that produces machine language output in an easily readable format for this case).

Interactive vs Batch Systems

It is essential that the pre-planning and test material preparation attributes of the batch approach be applied to interactive testing. This is necessary for problem reproduction, a necessary debugging task. It also appears that the organized methods of the batch testing result in software test optimization which needs to be applied to interactive testing.

System Hardware/Software Architecture

The testing methodology defined above applies to any type of computerized systems. AFR 800-14, for example, makes no distinction regarding the type of system in advocating the CPC, CPCI, PQT, and FQT levels of testing. The literature survey also revealed no new developments in this area for a specific type of computer system.

Integration Concepts

Testing methodology depends to some extent on the development methodology, specifically the integration techniques used. Traditional developments employ bottom-up testing in which increasing aggregates of small programs are tested in succession, generally using program drivers. Top-down developments replace program "stubs" with analytical data to simulate the effect of having executed the program element that replaced the stub. The "build" concept requires that stimulus-response patterns be determined to test out constructs of program elements that implement particular stimulus-response paths. All of these methods employ simulation software which itself has to be debugged [47].

Also, unique integration situations have unique effects on testing methodology. For example, when new software is to be integrated with existing, established software, drivers or stubs might not be necessary. Existing, proven tests can be used to provide inputs to new code and show that the old code is not perturbed, and more operationally-oriented testing can be conducted. The debugging activity, in this case, is eased by the existence of proven software and tests. For the case where integration involves general-purpose routines developed by a separate contractor, either that developer's drivers or stubs can be employed or, if scheduled properly, the routines themselves may be used. Unless these routines are thoroughly checked-out prior to their use, however, testing may inadvertently become a test vehicle for them and greatly perturb debugging. For the case where new software is integrated with new or established interfacing hardware, beside the debugging associated with the hardware simulators which are akin to those of drivers or stubs, the debugging problem is aggrevated by the existence of an element which is usually foreign to a software debugger, namely the hardware itself. The testing methodology can anticipate the possible problem areas with "smoke tests" which demonstrate that an understanding of the interfaces exists.

3.1.8 Testing Tools and Techniques

The testing tools and techniques used during the integration phase of the development cycle have an important effect on defining a debugging methodology. They affect the problems uncovered, the symptoms revealed, the validity of the testing itself, and in some cases point out precisely where in a program the problem exists. Integration phase testing tools and techniques fall into three main classes:

- Static code analysis tools and techniques
- Dynamic monitoring tools and techniques
- Code execution testing tools and techniques

Each of these classes of tools and techniques will be employed using the computer for which the software to be tested is actually built, a computer which emulates this computer, or a computer which simulates this computer. The techniques of computer emulation and computer simulation, while they apply to all three classes, are described in Paragraph 3.1.8.3.

3.1.8.1 Static Code Analysis Tools and Techniques

Static code analysis tools examine the source code of program modules to determine the program's logic flow, data usage, etc. The analysis performed is static, i.e., the program under analysis is not executed. The outputs of these tools are used in V&V by a technical analyst to verify the compliance of coded programs to design. All of these tools analyze programs coded in a specific programming language. Manual analysis of code has been supplemented by automated analysis for the following reasons:

- It is one of the more tedious tasks that can be placed on programmers.
- Human analysts make errors when doing tedious things.
- Computer time is much cheaper than analyst time.
- Tedious processes can be easily and cheaply repeated to check the effects of minor code changes.
- Code too complex to be retained and understood by the human mind can be automatically decomposed to its simplest representation to facilitate analysis [47].

The following discussion describes the static code analysis tools that can be employed during the program integration phase.

Automated Flow Charts

This program analyzes the syntax of a program in a specific programming language to graphically represent the control flow of the object program's internal logic. Some automatic flow charters incorporate programmer comments in the graphic display outputs. Most automated flow charters do not analyze usage of data variables, although they often reflect where data are set and used. Flow charts may be used for depicting system, subsystem, or program-level design. This type of tool is constrained by a specific programming language, making it inapplicable for multi-language development projects [47,13 & 34].

Code Auditor

This program analyzes the syntax of a CPC to examine each statement for specific coding conventions established by the contractor. This type of tool is constrained by a specific programming language and coding conventions, making it inapplicable for multi-development projects. However, it has been found to be an effective mechanism for enforcing programming standards and improving both verification and maintenance activities [13 & 34].

Set-Use Matrix/Cross-Reference Analysis

This tool is a program usually associated with compilers and provides information on the usage of programs, labels, tags, data variables, constants, subroutines, macros, or other program elements. The information usually includes the name, a set/use indicator, and the location(s) in the program where the identified item is set and used. The set/use matrix provides a static trace of data flow. A set/use matrix can be obtained for a CPCI by utilizing a sophisticated system monitor that uses the compiler-generated output for each program module or CPC as input. This type of set/use matrix, also referred to as a cross-reference analysis, is then generated for all data variables used and set by each of the CPCs in the CPCI. It is also possible to obtain cross-reference information on other system components, such as CPCs, file names, and macro names. This type of tool is constrained by a specific programming language, making it inapplicable for multi-development projects [13 & 34].

Editor

This tool is used to analyze source code for coding errors and to extract information that can be used to check relationships between sections of code. Among other functions, the error detection capability determines whether the code:

- Sets and clears flags properly
- Uses error-prone instruction sequences
- Sets up calling sequences properly
- Modifies instructions
- Attempts to reference or modify restricted data
- Uses restricted instructions
- Contains inaccessible instructions.

The second capability of an editor is similiar to the set/use matrix tool and provides a comprehensive cross-reference listing giving information pertaining to references to program data and subroutine calling structures. Editors work very well in finding mechanical violations of programming standards. They can also be used to flag coding techniques determined to be risky for the application [47,34 & 60].

Unit Consistency Analyzer

This tool analyzes the syntax of program modules written in a specific programming language to verify the consistent unit usage of globally defined data elements by modules. The purpose of the tool is to ensure that the use of unit definitions by each CPC is consistent with the system parameter definition. This tool requires that the programming language used to code the CPC must support data unit definition [13,34,60 & 69].

Interrupt Analyzer

This tool examines source code and determines potential conflicts in data usage and storage due to interrupts. Its use is a good example of using automated rather than manual methods to scan a large number of statements to find subtle combinations of code that could cause problems if undetected in the program's execution. The design of this tool is highly dependent upon the computer and programming language chosen [60,69 & 113].

Equation Generator

This tool reads assembly language code and translates portions representing mathematical equations to a real or pseudo HOL. The HOL version is output and then can be manually compared with the original equations to determine if the assembly language program accurately represents the equations. This tool is a good example of independent verification accomplished by separating the program verification functions from the program writing function.

This tool is required only for programs written in assembly language and must be designed for a specific language. Unless the language contains data-description features, the design of the equation generator can be a difficult task [47,60 & 113].

Comparator

This tool is used to compare two versions of the same program or two sets of data values to find areas of difference. This tool limits the scope of reverification that has to be performed on programs that have been modified [47.60 & 10].

Program Structure Analyzer

This tool is used to analyze the program paths under input control. They allow different types of sequences of code to be specified (e.g., all subroutine calls, all interrupt-related instructions, all extension register operations), and obtain information such as estimates on timing, paths followed, and entry conditions for specific paths [47,13 & 69].

Correctness Proofs

This technique establishes, in a mathematical fashion, that a given program performs a desired function. The proof technique mathematically determines the correspondence between a function in its coded form and the same function presented in the pertinent specification in mathematical and English language descriptions. Operational use of this technique has been limited mostly to manual proofs, although verifiers are being researched on a limited scale [47,60 & 69].

Symbolic Program Executor

This tool decomposes source code by logically executing it. It provides a capability to express paths in terms of both all necessary conditions to be satisfied in selecting the path and the result of transversing the path. Some symbolic program executors can also be used to produce a structured representation of an unstructured program [47,13,34 & 69].

3.1.8.2 Dynamic Monitoring Tools and Techniques

Dynamic monitoring is a technique for collecting data on the performance of an existing system for the purpose of evaluating or improving performance or reconfiguring the system. The process includes both the collection and the analysis of performance data and can be accomplished by hardware, software, or a combination of both. A hardware monitor is a unit attached directly to a computer's circuitry to obtain and record instruction execution, data transfer, and control information. Hardware monitoring techniques do not perturb the process under evaluation. They can also obtain occurrence and duration data of simultaneous events. A software monitor is a computer program that collects performance data on system operation. It usually perturbs the process under evaluation. Software monitoring techniques instrument a source program to obtain required information at strategic points in operation. Performance measuring techniques use a combination of hardware and software monitors. The following paragraphs describe the dynamic monitoring tools and techniques that can be employed during the program integration phase.

Path Flow Analysis

This program analyzes the syntax of a CPC to instrument the source code. Instrumentation is the process of generating and inserting instructions at strategic program locations. The modified program is then compiled and linked with recording routines.

The instrumentation is transparent to the programmer. The CPC is executed with user-supplied test case data and the execution of the CPC is dynamically recorded via the instrumentation. The output data from this type of tool describes the execution of each statement and sometimes includes information concerning input data processing. The output data are used to generate a more exhaustive set of test cases or to identify code that is inefficient or superfluous.

The path flow analyzer does not prove the program correct in any way, but it does provide an indication of the amount of testing applied to the CPC. The output generated by automatic execution analysis tools for a moderately sized program with a minimum set of test cases takes time to obtain, analyze and understand. An instrumented program may take as much as 75 to 85 per cent longer to operate on a single test case than the noninstrumented version of the program on the same test case. However, path flow analysis provides a quantitative measure of the percentage of a computer program that was tested [47,13,34 & 69].

Timing Analyzer

This is a computer program that monitors the execution time of all program elements. It often is a specialized capability of a program that does path flow analysis. It is used to verify CPC timing requirements. It adds a timing overhead and generally perturbs the process under evaluation [47,13,60 & 10].

Operating and Performance Measurement Techniques

These techniques require that the parameters impacting individual CPCI performance, as well as the interaction and dependence of those parameters upon each other, be identified so that specific performance characteristics (e.g., operating time, core/peripheral storage transfer requirements) can be measured. Some performance measurement techniques also use algorithms for processing and analysis of the data. These techniques are used to verify program performance in respect to the measured characteristics. They are designed in accordance with the specific requirements and charactertics of the program whose performance is to be measured. In addition, the characteristics of the computer must be considered in their design. These techniques combine special-purpose software programs and hardware monitors [13].

Hardware Monitor

This is a hardware tool that obtains signals from probes attached directly to a host computer's circuitry. The signals are recorded during the execution of a test case. The data are reduced after the test to obtain information on CPU utilization, channel activity, etc. The information can be used to verify and improve system and individual program performance [60,85 & 71].

3.1.8.3 Code Testing Tools and Techniques

Code execution testing refers to the traditional mode of software testing where the program is tested by executing the object instructions generally for the computer it is built, or on emulators or simulators. The following paragragphs describe the code execution testing tools and techniques that can be employed during the program integration phase.

Drivers

Drivers are normally needed in bottom-up development. They are computer programs that operate together with the software being tested and control and monitor the execution of each program entity (sub-routine, functional segment, program and CPCI). They are designed to demonstrate any or all of the following:

- All instructions have been executed at least once.
- All error conditions have been processed.
- All parameters are dimensionally correct.
- Special input values, such as discontinuity points are processed correctly.
- Subroutine calls are formatted correctly.
- All logic branches are correct.
- Arithmetic results are correct for nominal conditions.
- Minimum and maximum values are processed correctly.
- Extraneous conditional operations are processed correctly.

After subroutine testing, drivers are used to test the assemblage of subroutines to validate that the CPCI can perform functions without interface difficulties [10].

Stubs

Stubs are computer programs that simulate a software module which is not yet developed and will eventually interface with the software presently developed and being tested. They are normally needed in top-down development. Stubs are executed with the software under test and are passed control (and perhaps data) by that software and, in turn, return control (and perhaps their own data) to that software. Stubs can be coded to be as simple or as complex as necessary. They might simply print a message indicating they receive control and then return. They might pass canned or typical values of the simulated routine and then return. They can be coded to be the same size as the routine they are replacing, and operate approximately for the same length of time. They can also contain algorithms to perform some of the processing of the routines they are simulating [13 & 34].

Test Data Generators

These tools produce test data to exercise the target program. It creates test data using statistical algorithms, random-number generators, etc. The test data is generally output in a format directly accessible by the program undergoing the test [13 & 69].

Computer Emulator

An emulator is a computer whose hardware can be tailored to make it operate exactly like another computer. It is used in testing and development when the target computer itself is being developed. The emulation computer achieves the characteristics of the target computer through micro-programming the computer's control unit. The use of a microprogrammed computer trades off some of the hardwire control unit's excess speed for flexibility in the definition of its instruction repertoire. This tradeoff is achieved by making the control unit itself programmable. Its operation is then controlled by sequences of microinstructions. These are called microprograms, which are stored in a small, fast semiconductor memory within the control unit, called the control store, and executed by a lower-level control unit. The preparation of these microprograms which define the computer's instruction repertoire is termed microprogramming. Thus, a microprogrammed computer is simply a computer within a computer. The computer's overall speed is not reduced because the control store is much faster than the main memory, fast enough for the control unit to execute the several microinstructions required to perform a single machine instruction within a single cycle of the main memory.

Although microprogramming was orignally conceived as a mechanism for facilitating minor adjustments to a computer's instruction repertoire, the instruction set can be altered radically by replacing the full complement of microprograms. In this way, the machine language of a different computer can be substituted for the original.

Functionally, an emulator can be used on a stand-alone basis to execute programs written for another computer. To simulate actual operating conditions, it can be coupled to an environment simulator to furnish it the required inputs. It usually has a hardware interface between itself and a second computer which is employed to operate the environment simulator. The static code analysis and dynamic monitoring classes of tools can also be run on an emulator [27].

Computer Simulator

A computer simulator is a program which simulates the execution characteristics of a target computer using a sequence of instructions of a host computer. This simulating software is usually called an Interpretive Computer Simulator (ICS). In simulating the target computer, the ICS will produce bit for bit fidelity, with the results that would be produced by the target computer following the same operation and initial conditions. ICSs are very often used in a closed-loop situation with an environmental simulator to test

software systems when the actual hardware is unavailable. The static code analysis and dynamic monitoring classes of tools can also be run on an ICS [60,10 & 71].

Terminal Simulator

A terminal simulator presents input messages to a control program as if they had been input from an actual terminal device [60].

Peripheral Simulators

Peripheral simulators range from those that provide a functional simulation of a peripheral device responding to a computer program on the assumption that all interface constraints are satisfied to those that provide responses modeled to a detailed level of timing and message formatting [60, 10, & 71].

Test Beds

A test bed is a computer program, and often a connected driving computer system, that simulates actual hardware and interfaces. This approach is generally an expensive and time-consuming method of conducting lengthy tests but it permits full control of the test environment and allows test repeatability and diagnostics [60,113, 10, & 71].

Algorithm Simulators

Algorithm simulators are computer programs used to test highly interactive logical and mathematical components such as those found in satellite onboard software. The simulator program generally tests hardware performance, convergences, failure modes, and accuracy. It is both a model of the external environment and a function-by-function simulation of the logic and mathematics of the flight software. These models are programmed on a general-purpose computer in such a way that their computations are equivalent to the computations on the target computer. The output allows verification that the algorithm satisfies the system software requirements and provides acceptance criteria for tests run on the target computer [113].

Scoring Program

This program performs the same calculations as a target system on a host machine and then automatically compares its results with the actual results computed by the target system. It notes discrepancies between the two computer results that exceed a given or computed acceptable margin (or tolerance). The scoring program combines the functions of an algorithm simulator and a comparator [60].

Data Reduction Programs

These programs translate machine output into a format easily read for project personnel. In some case, these programs subject machine output to statistical or analytical analyses before outputing the listing [13].

3.1.8.4 Relation of Testing Tools and Techniques to Other Debugging Considerations

The effect of other debugging considerations on testing tools and techniques pertinent to the debugging methodology development can be described as follows:

- Software Management Methodology. The internal or external management system regulations and standards may advocate the use of testing tools and influence their employment. For example, AFR 800-14 recognizes comparators, editors, flowcharters, logic/equation generators, pathfinders, and ICSs. The internal and external configuration control procedures used for application software development can also be used for tool acquisition or development. Likewise, a test tool evaluation policy is often established to aid in the selection and evaluation of new tools.

- Software Tolerance Requirements. The software requirements considerations presented in Paragraph 3.1.2 greatly influence the types of testing tools used during the program integration phase. Some of the effects of tolerance requirements may include:

  - Sizing constraints can greatly effect the applicability of such tools as the path flow analyzer. This tool requires two to three times more core due to the insertion of instrumentation code in the testing software [13].

  - Tools, such as the path flow analyzer, computer simulators, and test beds, have a definite degrading effect on program and interface timing [13].

  - Tools, such as computer, peripheral, and algorithm simulators, must be more reliable than the reliability specified by the tolerance requirements [13].

- Language Considerations. The characteristics of the programming language, the associated compiler/assembler, and the computer characteristics all may impact the tool selection process. Some programming languages permit or require declaration information which increases the information output and, in turn, aids in the analysis of the software under test [75 & 59]. The tools affected by the programming languages are those that analyze the syntax of source language programs. Many HOLs are test oriented, and for that reason should be used to optimize the debugging process. Compilers/assemblers often include set/use matrices and storage utilization maps. Recent trends in compiler/assembler development incorporate more testing/debugging options [50] This has a positive effect on debugging because problems may be uncovered at the time the code is being compiled. Systems also presently exist which incorporate these tools within the design

development stage, working on design language data instead of program code. This approach enables the detection of problems detected by such tools as the path flow analyzer, code auditor, flowcharter, and editor during the program design stages [69] Compiler/assembler-dependent tools may need the capability to be compiled at more than one computing facility. Computer-dependent tools using compiler/assembler output may need to run on more than one computer. Both of these possibilities may require the replication of the test tools for the specific environment. In environments which have sizing or timing limitations, the operational computer may have to be emulated to use certain tools.

- Interactive vs Batch Systems. The literature survey did not reveal any of the above test tools which also include an interactive capability (i.e., when a problem is flagged, the tool/system allows correction and then reprocesses the corrected code). Test tools may be available on both batch and interactive systems, but the tools run to completion before any changes can be made to the software.

- System Hardware/Software Architecture. The static code analysis class of tools are available principally for large scale systems and for avionics computer systems [69]. With the exception of hardware monitors, the dynamic performance monitoring class of tools is employed mainly by large-scale systems [69]. Hardware monitors are used to some degree by all computer systems [60 ,106 & 25]. As for the code execution testing class of tools, the large scale systems, minicomputer systems, distributed systems, and real-time systems, employ all of the tools of this class, where appropriate. Microcomputer systems employ computer emulators and computer simulators as their principal test tools [27, 88 & 24].

- Integration Concepts. The static code analysis tools, the dynamic monitoring tools (except hardware monitors), and the code execution testing tools (except test beds and emulators) are limited during integration to testing a set of CPC increments which can fit with the tool in core. Except for this limitation, they may be used with any development methodology.

- Testing Methodology. The testing tools and techniques described above are applicable to different stages of the testing methodology. Module and CPC-level testing aids are designed to help the programmer uncover errors in program code that cause abnormal behavior or termination to occur with a given set of inputs. The selection and use of test tools at this stage depends on whether they provide sufficient information to demonstrate the following:

  - The CPC's internal logical construction.

- The CPC's input test case data, including nominal, default, null, critical, maximum, and minimal data values.

- Integrity of the CPC's output data.

- Data base integrity, before and after CPC execution.

- The CPC's instruction-execution frequency and related timing information.

The following testing aids are frequently used for module and CPC-level testing [57].

- Drivers

- Stubs

- Test data generators

- Data reduction programs

In addition, the following test tools may be used but generally provide information applicable to CPC-level testing.

- Path flow analyzer (for CPC instruction-execution frequency)

- Timing analyzer (for CPC timing information)

- Unit Consistency Analyzer (for data base integrity)

- Computer emulator (if target computer is not available)

Many of the tools discussed for module and CPC-level testing are also used for CPC incremental-integration testing, such as drivers, stubs, data reduction programs, and test-case generators. Additional tools used for CPC integration testing concentrate on verifying that the basic algorithms correctly operate together.

Selection and use of test tools at this stage should consider how the tools provide information on the following:

- CPC interface integrity.

- Input data that is representative of the actual or live data.

- Instruction execution frequency and timing data.

- Core allocation data.

- Data base integrity, before, during, and after CPC operation.

- Output data integrity, such as message and display formats.

The following tools are used for CPC integration testing [47, 13 & 10].

- Path flow analyzer
- Operating and performance measurement techniques
- Set use matrix/Cross-reference analysis
- Unit consistency analyzer
- Hardware Monitor
- Automated flow charts
- Code auditor
- Editor
- Interrupt analyzer
- Equation generator
- Program structure analyzer
- Symbolic program executor
- Computer simulator

For CPCI, PQT, and FQT testing, selection and use of test tools should consider how the tool provides information on the following:

- Satisfaction of requirements of Development (Part I) Specification.
- All components of CPCI interface together.
- Limits of tolerance and qualification criteria.

In addition to the tools used in CPC incremental-integration testing, the following tools can be used to assist in CPCI, PQT, and FQT testing [60, 10, & 71]:

- Test beds
- Terminal simulator
- Peripheral simulator
- Algorithm simulator
- Scoring program
- Comparator

3.2 SOFTWARE ERRORS

The subject of software errors has been quite thoroughly researched by recent studies with the intent of collecting, categorizing, and evaluating errors for the purpose of developing software error prediction models, and of supporting relevant test tool development and reliable software model development. In addition, it is a prime consideration in defining a debugging methodology. This discussion presents a summary of relevant studies on software errors that pertain to the debugging study, in terms of:

- Types of errors
- Error symptoms
- Error messages
- Relation of software errors to other debugging considerations

3.2.1 Types of Errors

A consolidated list of software errors is presented in Figure 19. The basis for this table is Boeing's RADC error analysis study [38]. Findings from other studies were merged where appropriate [117,82,29 , & 47]. The list is important because types-of-errors are being correlated with testing tools and techniques. Figure 20 represents a translation of recent tool evaluation and error analysis studies into a matrix equating testing tools and techniques with the types-of-errors they uncover [117,47]. The matrix should be a significant input to the debugging methodology definition task.

3.2.2 Error Symptoms

Figure 19 includes both error causes and error symptoms. Understanding the causes of software errors leads to an understanding of the symptoms of errors. It is the symptoms of errors with which the debugging process basically deals [77]. The symptoms of a problem give hints to the cause of the problem, and it is on these hints that the debugging tools and techniques are employed.

There appear to be two classes of symptoms, primary and secondary. Primary symptoms are symptoms apparent at the time a problem is encountered. These include the failure features noted in the problem/discrepancy report. The following symptoms belong to the primary class:

- Error messages
- Incorrect output/results
- Abnormal initiation
- Abnormal termination
- Inputs not accepted
- Incorrect output format/sequence
- Missing output
- Excessive output
- Excessive run time
- Infinite loop

CATEGORIES/TYPES OF ERROR

COMPUTATIONAL ERRORS
- TOTAL NUMBER OF ENTRIES COMPUTED INCORRECTLY
- PHYSICAL OR LOGICAL ENTRY NUMBER COMPUTED INCORRECTLY
- WRONG EQUATION OR CONVENTION USED
- MATHEMATICAL MODLING PROBLEM
- RESULTS OF ARITHMETIC CALCULATION INACCURATE/NOT AS EXPECTED
- MIXED MODE ARITHMETIC ERROR
- TIME CALCULATION ERROR
- TIME CONVERSION ERROR
- TIME TRUNCATION/ROUNDING ERROR
- SIGN CONVENTION ERROR
- UNITS CONVERSION ERROR/CONFLICT ERROR
- VECTOR CALCULATION ERROR
- CALCULATION FAILS TO CONVERGE
- QUANTIZATION/TRUNCATION ERROR

LOGIC ERRORS
- LIMIT DETERMINATION ERROR
- WRONG LOGIC BRANCH TAKEN
- LOOP EXITED ON WRONG CYCLE
- INCOMPLETE PROCESSING
- ENDLESS LOOP DURING ROUTINE OPERATION
- MISSING LOGIC OR CONDITION TEST
- INDEX NOT CHECKED
- FLAG OR SPECIFIC DATA VALUE NOT TESTED
- INCORRECT LOGIC
- SEQUENCE OF ACTIVITIES WRONG
- FILTERING ERROR
- STATUS CHECK/PROPAGATION ERROR
- ITERATION STEP SIZE INCORRECTLY DETERMINED
- LOGICAL CODE PRODUCED WRONG RESULTS
- LOGIC ON WRONG ROUTINE
- PHYSICAL CHARACTERISTICS OF PROBLEM TO BE SOLVED, OVERLOOKED, OR MISUNDERSTOOD
- LOGIC NEEDLESSLY COMPLEX
- INEFFICIENT LOGIC
- EXCESSIVE LOGIC
- STORAGE REFERENCE ERROR (SOFTWARE PROBLEM)
- DATA OPERATED AS INSTRUCTION OR VICE VERSA

I/O ERRORS
- MISSING OUTPUT
- OUTPUT MISSING DATA ENTRIES
- ERROR MESSAGE NOT OUTPUT
- ERROR MESSAGE GARBLED
- OUTPUT OR ERROR MESSAGE NOT COMPATIBLE WITH DESIGN DOCUMENTATION (INCLUDING GARBLED OUTPUT)
- MISLEADING OR INACCURATE ERROR MESSAGE TEXT
- OUTPUT FORMAT ERROR (INCLUDING WRONG LOCATION)
- DUPLICATE OR EXCESSIVE OUTPUT
- OUTPUT FIELD SIZE INADEQUATE
- DEBUG OUTPUT PROBLEM (RELATIVE TO DESIGN DOCUMENTATION)
- LACK OF DEBUG OUTPUT
- TOO MUCH DEBUG
- HEADER OUTPUT PROBLEM
- OUTPUT TAPE FORMAT ERROR
- OUTPUT CARD FORMAT ERROR
- ERROR IN PRINTER CONTROL
- LINE COUNT/PAGE EJECT ERROR
- NEEDED OUTPUT NOT PROVIDED IN DESIGN
- INSUFFICIENT OUTPUT OPTIONS
- I/O RATES IMPROPERLY MODELED
- INTERRUPT PROCESSING ERROR
- REGISTERS NOT SAVED

DATA HANDLING ERRORS
- VALID INPUT DATA IMPROPERLY SET/USED
- DATA WRITTEN IN OR READ FROM WRONG DISK LOCATION
- DATA LOST/NOT STORED
- DATA, INDEX OR FLAG NOT SET OR SET/INITIALIZED INCORRECTLY
- NUMBER OF ENTRIES SET INCORRECTLY
- DATA, INDEX OR FLAG MODIFIED OR UPDATED INCORRECTLY
- NUMBER OF ENTRIES UPDATED INCORRECTLY
- EXTRANEOUS ENTRIES GENERATED (TABLE, ARRAY, ETC.)
- BIT MANIPULATION ERROR
- FLOATING POINT/INTEGER CONVERSION ERROR
- INTERNAL VARIABLE ERROR (DEFINITION OR SET/USE)
- DATA PACKING/UNPACKING ERROR
- ROUTINE LOOKING FOR DATA IN NON EXISTENT RECORD
- BOUNDS VIOLATION

Figure 19. Software Errors

CATEGORIES/TYPES OF ERROR	
DATA HANDLING ERRORS (cont'd) DATA CHAINING ERROR DATA OVERLOW OR OVERFLOW PROCESSING ERROR READ ERROR ALL AVAILABLE DATA NOT READ LONG LITERAL PROCESSING ERROR SORT ERROR OVERLAY ERROR SUBSCRIPTING CONVENTION ERROR DOUBLE BUFFERING ERROR DATA STORED IN WRONG ORDER LOCAL VARIABLE USED IN PLACE OF GLOBAL	TAPE PROCESSING INTERFACE ERROR TAPE UNIT EQUIPMENT CHECK NOT MADE ROUTINE FAILS TO READ CONTINUATION TAPE ROUTINE FAILS TO UNLOAD TAPE AFTER COMPLETION ERRONEOUS INPUT TAPE FORMAT
OPERATING SYSTEM/SYSTEM SUPPORT SOFTWARE ERRORS COMPILER PRODUCES ERRONEOUS MACHINE CODE OS MISSING NEEDED CAPABILITY	USER INTERFACE ERRORS OPERATIONS REQUEST OR DATA CARD/ROUTINE INCOMPATABILITY MULTIPLE PHYSICAL CARD/LOGICAL CARD PROCESSING ERROR INPUT DATA INTERPRETED INCORRECTLY BY ROUTING VALID INPUT DATA REJECTED OR NOT USED BY ROUTINE INPUT DATA REJECTED BUT USED INPUT DATA READ BUT NOT USED ILLEGAL INPUT DATA ACCEPTED AND PROCESSED LEGAL INPUT DATA PROCESSED INCORRECTLY POOR DESIGN IN OPERATOR INTERFACE INADEQUATE INTERRUPT AND START CAPABILITY CODE CHANGE/CORRECTION INPUT INCORRECTLY
CONFIGURATION ERRORS COMPILATION ERROR SEGMENTATION PROBLEM ILLEGAL INSTRUCTION UNEXPLAINABLE PROGRAM HALT	DATA BASE INTERFACE ERRORS ROUTINE/DATA BASE INCOMPATIBILITY UNCOORDINATED USE OF DATA ELEMENTS BY MORE THAN ONE USER
ROUTINE/ROUTINE INTERFACE ERRORS ROUTINE PASSING INCORRECT AMOUNT OF DATA (INSUFFICIENT OR TOO MUCH) ROUTINE PASSING WRONG PARAMETERS OR UNITS ROUTINE EXPECTING WRONG PARAMETERS ROUTINE FAILS TO USE AVAILABLE DATA ROUTINE SENSITIVE TO INPUT DATA ORDER CALLING SEQUENCE OR ROUTINE/ROUTINE INITIALIZATION ERROR ROUTINE USED OUTSIDE DESIGN LIMITATION ROUTINE WON'T LOAD (ROUTINE INCOMPATIBILITY) ROUTINE OVERFLOWS CORE WHEN LOADED REAL-TIME ROUTINE NOT REENTRANT ROUTINE/SYSTEM SOFTWARE INTERFACE ERRORS OS INTERFACE ERROR (CALLING SEQUENCE OR INIALIZATION) ROUTINE USES EXISTING SYSTEM SUPPORT SOFTWARE INCORRECTLY ROUTINE USES SENSE/JUMP SWITCH IMPROPERLY	USER REQUESTED CHANGES SIMPLIFIED INTERFACE AND/OR CONVENIENCE NEW AND/OR ENHANCED FUNCTIONS CPU DISK TAPE I/O CORE SECURITY NEW HARDWARE/OS CAPABILITY INSTRUMENTATION CAPACITY DATA BASE MANAGEMENT AND INTEGRITY EXTERNAL PROGRAM INTERFACE

Figure 19. Software Errors (cont'd)

CATEGORIES/TYPES OF ERROR	
PRESET DATA BASE ERRORS DATA OR OPERATIONS REQUEST CARD DESCRIPTIONS ERROR MESSAGE TEXT NOMINAL, DEFAULT, LEGAL, MAX/MIN VALUES PHYSICAL CONSTANTS AND MODELING PARAMETERS EPHEMERIS PARAMETERS DICTIONARY (BIT STRING) PARAMETERS MISSING DATA BASE SETTINGS GLOBAL VARIABLE/COMPOOL DEFINITION ERRORS ITEMS IN WRONG LOCATION (WRONG DATA BLOCK) DEFINITION SEQUENCE ERROR DATA DEFINITION ERROR TABLE DEFINITION INCORRECT LENGTH OF DEFINITION INCORRECT COMMENTS ERROR DELETE UNNEEDED DEFINITIONS	DOCUMENTATION ERRORS ROUTINE LIMITATION OPERATING PROCEDURES DIFFERENCE BETWEEN FLOW CHART AND CODE TAPE FORMAT DATA CARD/OPERATION REQUEST CARD FORMAT ERROR MESSAGE ROUTINE'S FUNCTIONAL DESCRIPTION OUTPUT FORMAT DOCUMENTATION NOT CLEAR/NOT COMPLETE TEST CASE DOCUMENTATION OPERATING SYSTEM DOCUMENTATION TYPO/EDITORIAL ERROR/COSMETIC CHANGE INSUFFICIENT PRECISION HARDWARE MODELING INCORRECT MISSING SYMBOLS OR LABELS DIMENSION OR UNITS ERROR
RECURRENT ERRORS PROBLEM REPORT REOPENED PROBLEM REPORT A DUPLICATE OF PREVIOUS REPORT	REQUIREMENTS COMPLIANCE ERRORS EXCESSIVE RUN TIME REQUIRED CAPABILITY OVERLOOKED OR NOT DELIVERED AT TIME OF REPORT REQUIRED TIMING NOT FOLLOWED

Figure 19. Software Errors (cont'd)

ERROR CATEGORY	TESTING TOOLS AND TE																
	AUTOMATED FLOW CHARTS	CODE AUDITOR	SET-USE MATRIX/ CROSS-REFERENCE ANALYSIS	EDITOR	UNIT CONSISTENCY ANALYZER	INTERRUPT ANALYZER	EQUATION GENERATOR	COMPARATOR	PROGRAM STRUCTURE ANALYZER	CORRECTNESS PROOFS	SYMBOLIC PROGRAM EXECUTOR	PATH FLOW ANALYZER	TIMING ANALYZER	OPERATING & PERFORMANCE MEASUREMENT TECHNIQUES	HARDWARE MONITOR	DRIVER	STUB
COMPUTATION ERRORS	X			X		X	X	X		X	X		X	X	X	X	X
LOGIC ERRORS	X		X	X	X	X		X	X		X	X				X	X
I/O ERRORS			X		X	X		X					X	X	X	X	X
DATA HANDLING ERRORS	X	X	X	X				X								X	X
OPERATING SYSTEM/SYSTEM SUPPORT SOFTWARE ERRORS						X		X	X							X	X
CONFIGURATION ERRORS		X						X	X								
ROUTINE/ROUTINE INTERFACE ERRORS				X	X			X	X							X	X
ROUTINE/SYSTEM SOFTWARE INTERFACE ERRORS				X	X			X								X	X
TAPE PROCESSING INTERFACE ERRORS				X				X								X	X
USER INTERFACE ERRORS		X															
DATA BASE INTERFACE ERRORS		X	X					X								X	X
USER REQUESTED CHANGES								X									
PROSET DATA BASE ERRORS				X				X								X	X
GLOBAL VARIABLE/COMPOOL DEFINITION ERRORS		X	X		X			X								X	X
RECURRENT ERRORS						X											
DOCUMENTATION ERRORS	X		X														
REQUIREMENTS COMPLIANCE ERRORS		X	X				X						X	X	X	X	X

Figure 20. Testing Tools/Techniques Applicable to Error Categories

TESTING TOOLS AND TECHNIQUES

PROGRAM STRUCTURE ANALYZER	CORRECTNESS PROOFS	SYMBOLIC PROGRAM EXECUTOR	PATH FLOW ANALYZER	TIMING ANALYZER	OPERATING & PERFORMANCE MEASUREMENT TECHNIQUES	HARDWARE MONITOR	DRIVER	STUB	TEST DATA GENERATOR	COMPUTER EMULATOR	COMPUTER SIMULATOR	TERMINAL SIMULATOR	PERIPHERAL SIMULATOR	TEST BED	ALGORITHM SIMULATOR	SCORING PROGRAM	DATA REDUCTION ROUTINE
	X	X		X	X	X	X	X	X	X	X			X	X	X	X
X		X	X				X	X	X	X	X			X			
				X	X	X	X	X	X	X	X	X	X	X			
							X	X	X	X	X			X			
X							X	X	X	X	X			X			
X																	
X							X	X	X	X	X			X			
							X	X	X	X	X			X			
							X	X	X					X			
							X	X	X	X	X			X			
							X	X	X	X	X			X			
							X	X	X	X	X			X			
				X	X	X	X	X		X	X			X	X	X	

2

Secondary symptoms are less apparent. They may be partially apparent, together with the primary symptoms, or may be surfaced by employing various debugging tools and techniques. The following symptoms belong to the secondary class:

- Occasions of occurrence
- Location of occurrence
- Inputs involved
- Outputs involved
- Expected results
- Hardware configuration
- Software configuration
- Status of problem (new vs old)

There is not a one-to-one correspondence between error symptoms and error causes. For example, if the cause of an output error message is a data value out-of-range due to a mathematical formula coding error, the symptom leading to this discovery is the error message. It is the error message then which leads the debugger to analyze the source code and discover that the equation fails for certain input values. On the other hand, the equation might be complex enough so that the debugger will have to try various ranges of input values to isolate the area of the problem. This might have to be followed by determining what really should happen in these cases, and finally determining where the equation is faulty. Thus, the cause of one error might be found through the uncovering of different amounts and types of symptoms. This affects the development of debugging methodology by indicating that there is no direct relationship between debugging tools/techniques and error causes. The direct relationship is between the tools/techniques and the symptoms of the errors.

3.2.3 Error Messages

Error messages are a primary error symptom that provide the most useful primary debugging information to the debugger. They generally describe where the problem is, what is wrong, and, if properly implemented, give information as to where the software error exists. This last feature simplifies the debugging process to the extent that initial anomalies lead rapidly to detectable problems. Programs coded in a highly structured language, augmented with a complete and informative set of error messages, will be considerably easier to debug than programs written at the machine level which output bit patterns and debugging information [77].

A common practice of implementing error messages is to develop a general-purpose error message routine which is called when an error condition is detected. The calling routine passes an error message number, plus any dynamic characters to be encoded in the message to the error message routine. This routine then retrieves the appropriate message from the data base, outputs the message, and returns to the calling program, depending on what response, if any, was made to the message. Error messages which are centrally defined, are significant. They allow for more control of the wording of the message, thus providing clarity.

All error messages produced by the general-purpose routine should appear in their entirety on the primary output device, (e.g., the line printer). Shortened versions of the message might appear on slower output devices (e.g., console typewriter).

It is desirable that the rules for the design of error checking logic be defined in a guideline document, such as a standards and conventions document. Guidelines, for example, could include requirements to perform the following types of error checks:

- All parameters on operation requests and data cards.
- Reasonableness of value. (If it is possible to define a range for a value, then checks should be performed to insure that the value input is within the range.)
- Proper format (e.g., a hollerith value should not be accepted for parameters defined as integer.)
- Completeness of input.
- Conflicting requests.
- Illegal combination of parameters [112].

It should be noted that a sophisticated form of error notification implementation is not always possible. Error message logic requires additional memory storage and also can degrade a program's timing. For some systems, such as a small real-time microcomputer subsystem, error messages might not be practical. Where timing and storage is critical, short messages (even simply numbers) might be output and the system's documentation can provide the necessary expansion of information.

3.2.4 Relation of Software Errors to Other Debugging Considerations

The relationships of software errors to debugging methodology are as follows:

- Software Management Methodology. The software management methodology should contain mechanisms for error and debugging data collection which can be used for subsequent debugging tool and methods analyses. The problem processing system should employ discrepancy report forms and resolution report forms for reporting problems and their resolution. These forms can also be used to describe the problem, what testing uncovered it, and how it was debugged. Thus, they provide a suitable vehicle for the debugging evaluation process.
- Software Tolerance Requirements. The secondary error symptoms of occasion and location of occurrence are very difficult to uncover for tolerance requirements errors. Tolerance requirements problems manifest themselves generally only at the CPC level, and debugging the problems at a lower level is not well developed.

- Language Considerations. A powerful compiler can catch many software errors at compile time. The literature survey revealed no development efforts dealing with error correction at the compiler level. It seems that with the error data now becoming available (e.g., the data presented in Figure 19), future language/compiler development efforts will attempt to eliminate the concrete, real-world errors represented by this data. The literature survey concerning error data collection and analysis indicated differences between errors reported for higher level language software and errors reported for assembly language software. The differences in errors reflect the difference in levels of abstraction required by the programmer in stating the problem/solution. Subtle types of errors produced by new, unestablished compilers/assemblers were not addressed by recent literature. The symptoms of these problems would be especially useful in debugging tool selection/evaluation.

- Interactive Versus Batch Systems. Since interactive systems gain control of operation on abnormal termination of a program, they should present error symptoms similar to the batch mode. In addition, interactive systems should provide the debugger with the software configuration status, so that problems of error reproduction are not encountered. In the batch system, the debugger often has more control over the elements in the environment during debugging runs.

- System Hardware/Software Architecture. The previous discussion of errors has been based on large-scale systems and minicomputer systems. Real-time, microcomputer, and distributed systems present still other problems.

  Real-time systems are designed to insure that the software does not terminate abnormally when an error occurs. This has the effect of allowing the error to corrupt other parts of the software which makas debugging very difficult because the symptoms are misleading and it is difficult to backtrace operational execution [93].

  Many microcomputer instruction sets are based on an eight-bit word, although some new models are sixteen-bit words. Representing an address in an eight-bit instruction is impractical for a reasonable sized memory. Consequently, an eight-bit microcomputer must provide multi-byte instructions. Variable-length instructions are new to many assembly language programmers, especially those who have programmed only large-scale computers, which generally use fixed word lengths. Variable-length instructions make debugging difficult because the address part of the instruction may be inadvertently fetched and executed as in instruction. This kind of error is particularly hard to find because nonsensical instructions, when executed, often corrupt other parts of the program or data base.

Many microcomputer assemblers and compilers are designed to fit small memories, and as a consequence, provide only the most primitive kinds of diagnostics. The error messages may be misleading or errors may not be diagnosed, leaving the programmer to determine what went wrong during program execution [106].

The unique errors associated with distributed systems involve computer-to-computer communication. Failure-detection recovery logic is usually present in the fully developed software, but for module integration testing between the interfacing software of two separate computers, special tools and techniques are required for debugging [32].

- Testing Methodology. The literature survey found no information regarding errors discovered by different testing approaches.
- Testing Tools and Techniques. The symptoms produced by the static code analysis and dynamic monitoring types of testing tools generally disclose secondary symptoms presented in Paragraph 3.2.2.2. The code execution testing tools produce both the primary and secondary classes of symptoms.

3.3 DEBUGGING AIDS

This discussion addresses the aids currently being employed for the debugging process. General debugging systems are first discussed. Then, the various classes of debugging tools are presented, including the relation of each tool to the previously discussed relevant debugging considerations.

3.3.1 Debugging Systems

Most debugging tools belong to a debugging system supporting the development process. A debugging system, for example, might possess some or all of the standard debugging tools (e.g., dumps or traces) whose operation may be requested, or controlled, by a debugging software monitor. Interactive debugging systems are representative of a debugging system in that they possess a set of commands which allow the debugger to activate the various debugging tools of the system. Figure 21 is a generalized list of the commands [87, 44, 57, 11, 6, 3, 54, & 8].

COMMAND	ACTION
DUMP	Display all or portions of a program's data or code.
TRACE	Output information on sequence of program's execution of code.
RETRACE	Trace backwards from current statement being executed.
GO	Operate the program.
STOP	Cease operation of the program.
SLOW	Slow down execution speed of the program.
FAST	Speed up execution speed of the program.
SET	Set or modify program's data or code.
IF	Perform debugging action if certain conditions are met.
AT	Perform debugging action at certain statement or location.
REMOVE	Discontinue current debugging action.

Figure 21. Typical Activation Commands for Debugging Tools

The debugging tools of batch-mode debugging systems are activated by requests similar to these commands. (SLOW and FAST actions are not applicable).

There are basically two types of batch or interactive debugging systems: (1) execution-oriented systems; (2) record/replay systems. Execution-oriented systems produce output as the program is executing while record/replay systems record data as a program executes and output it after the program has finished executing. Examples of execution-oriented systems are New York University's AIDS system [44], the Institute of Defense Analysis' HELPER system [57], and the Purdue University's PEBUG system [11]. RAND's EXDAMS system [6] is an example of a record/replay system, while the University of Oregon's MANTIS system [3] is a combination of both types.

3.3.2 Debugging Tools

The following discussion describes the various classes of debugging tools currently in use. Some of the debugging considerations previously discussed will be related to each tool described.

3.3.2.1 Dumps/Displays

Dumps are displays of the computer and peripheral register/storage for purposes of showing the status of processing at a particular time. This is perhaps the most commonly used debugging tool and it exists in many forms, as follows:

- Post-Mortem Dumps. A display of the contents of main memory, provided in various formats, output automatically or upon request, and usually when a program has terminated abnormally. The same form of dump is sometimes available before program operation, or subsequent to normal operation. These dumps can either display all of main memory or selective portions of it [67, 116 & 61].

- Snapshots. Output for this dump occurs as soon as the request is made and execution continues upon its completion. The user specifies one or more snapshot points and the information desired. The instruction at each such point is preserved and replaced by a transfer to the snapshot routine. When the program is executed and control reaches a transfer point, a jump to the snapshot routine occurs and the required information is output. The original instruction is then obeyed and execution of the program is resumed. The advantage of the snapshot technique is that it obviates the need to include explicit print statements in the program; the disadvantage is that it is usually a facility provided at a low level and the snapshot points may have to be specified in terms of actual machine addresses [67].

- Trap/Breakpoint Dumps. These capabilities cause program execution to halt when certain conditions are met (e.g., a certain location is executed). When a breakpoint is reached, a dump or display of storage may be taken. In some cases, this is a static display and the program continues; in others, it is a dynamic display updated as the program continues. These dumps can either display all of main or auxiliary storage or selective portions of them, and can be provided in various formats [79].

- Programmed-In Dumps. These are programmer coded print statements for the purpose of printing the content of registers, data items, tables, etc. These displays are sometimes referred to as diagnostics. They may be conditional displays which are output only when specifically requested, when sense switches are set, or when some internal condition is met. They display exactly what the programmer thinks will be needed in an understandable format; however, they perturb the software execution, and require extra storage and execution time of the program [67 & 18].

- Monitor Dumps. Monitor dumps are usually provided by hardware debugging tools. They provide a display of selective portions of main memory and do not perturb the program's execution. Depending on the hardware, they can provide output that is either easy or difficult to analyze. These displays are very useful in real-time systems where it is desired not to perturb a program's timing [23, 28 & 39].

- Auxiliary Storage/Utility Dumps. These usually include post-program execution displays of disc areas and magnetic tape outputs, which are also useful in debugging. Also included are: source code, corrector, and software configuration status listings [116].

Dumps are related to some of the other debugging considerations which have previously been brought out as follows:

- Software Tolerance Requirements. The types of dumps applicable to a system with stringent timing, storage, and throughput requirements should include monitor dumps [87, 39, 93].

- Language Considerations. There is wide-spread opinion that the behavior of programs should be explainable in terms of the language in which they are written. If this is true, it is especially applicable to dumps. In most cases the dumps provided by a system are at a lower level language than that of the programs. The following types of formats exist for dumps:

  - Machine Format. Machine format dumps reflect the condition of the software in main memory. The display is produced in terms of main memory locations. (In some cases the locations have symbolic labels associated with the program.) The contents of these locations are displayed in numerous selectable formats for some systems, e.g., octal, hexadecimal, floating point, decimal, character mode, or symbolic machine instructions [116,30].

  - Language-Oriented Format. Language-oriented format dumps reflect the language in which the software is coded. There are two basic types of language-oriented dumps: (1) compile-time inserted devices; and (2) run-time devices. Compile-time devices include the insertion of conditional code (similar to programmed-in dumps) which can be switched on or off for execution, or provide displays of data values or blocks of values in language-coded terms (e.g., Extended Fortran Debug Feature) [104]. Run-time devices range from correlation of higher level or assembly level language to machine locations, to the display of HOL statements, allowing modification, and redisplay of new statements. For example, the DDT system at the University of California at Berkeley has facilities for symbolic reference to assembly language programs [43], while the GRAPE system at Harvard University displays 40 FORTRAN statements on a refresh display at a time, and as each statement executes, that statement is briefly modified to show the results of the execution [42].

- Interpretive Format. Interpretive format refers to the display of a program's code and data in a form interpreting the language in which the program was coded. In other words, the display of code and data is interpreted, formatted, and displayed in a totally user-oriented form. For example, the UNRAVEL system at the University of Kent provides dumps that can include the following [19].

  1. The values of all variables in the spotlighted program with the name of the corresponding variable against each value.

  2. Information about the operating environment of the spotlighted program; e.g., status of I/O devices, contents of buffers, location of workspace areas, running time, or reason for failure.

  3. Workspace areas, printed in appropriate formats (e.g., dictionaries, stacks, lists).

  Also, RANDS EXDAMS System [6] displays values, node trees, and maps in a motion picture format (see Paragraph 3.3.2.4).

- Interactive vs Batch System. Both interactive and batch systems employ post-mortem, snapshot, trap/breakpoint, programmed-in, selective, and auxiliary storage dumps. Monitor dumps are usually only associated with interactive systems. Low level dumps (machine language) are usually suitable only to batch systems because of the analysis time requirements [67 & 116]. Dumps which are output on graphical debugging tools (see Pargraphs 3.3.2.4) are interactive in nature. Many items shown in a graphical dump are programmed so that they can be sensed by a light pen or similar device. The users execute a debugging process by pointing at an associated display item. The debugging program performs the requested task, updates the display, and waits for the next inquiry. The speed, selectivity, and flexibility provided by graphical dumps are a unique and important feature of interactive systems [95].

- System Hardware/Software Architecture. In most cases, all dumps available can be applied to large-scale, minicomputer, and distributive systems. Monitor dumps are especially suited for distributed and real-time systems, and microcomputer systems employ a unique hardware device called the Logic State Analyzer to provide monitor dumps (see Paragraph 3.3.2.4). Another monitor dump feature of microcomputer systems is the Interrupt Driven Mapping feature which uses a periodic interrupt source, such as a real time clock, to activate a program which collects data for a histogram. The histogram can then be compared to the system load map to determine which routines were most frequently active when interrupts occurred [36]. Computer evaluation kits for microcomputers provide a means for dumping memory contents on a printer via a read only memory (ROM)-based monitor [95].

3.3.2.2 Traces

Traces are displays based on the sequence of program operation. In some cases, they consist of a list of addresses of instructions executed without any transfer [116]. In other cases, a trace might consist of information on program and computer states being output for each statement or location executed [67].

In theory, the programmed-in dump relates to a trace by producing output whenever a specific statement is executed. Debug-oriented compilers also exist which insert print logic at branch-type statements (e.g., GOTO or IF) which result in the same type of output [33]. These trace-like dumps, however, do not fit the definition of a trace. A trace involves the monitoring of the execution sequence by a debugging tool as the program is being operated.

The problems which arise from using a trace are excessive output and greatly reduced execution speed. The former is controlled to some extent by a debugging capability which specifies an upper limit on the number of times any statement is be traced. The latter difficulty cannot be circumvented unless there is a capability for specifying specific areas of the program to be traced. Execution of the program in other regions proceeds at normal speed [67].

Traces can be categorized into three classes as follows:

- Dynamic Internal Traces. These are traces, performed by software within the same computer as the traced program, which are output as the trace progresses. The AIDS system, for example, prints out pairs of addresses, between which, instructions were executed without transfers [44]. The GRAPE system displays about 40 FORTRAN statements on a screen at one time, and as each statement executes, that statement is briefly modified to show the results of the execution [42].

- Recorded Traces. These traces collect data on the sequence of a program's execution and, upon completion, output the traced information. The EXDAMS system, after recording the instructions operated, allows the user to watch the program statements execute either forward or backwards, via visual display [6]. The University of Oregon's MANTIS system logs each exception to sequential execution into a trace table and later types out the trace of these exceptions [3].

- Monitor Trace. These traces are provided by a hardware device on a second computer monitoring the instruction execution of the primary computer. This is frequently done for tracing micro-computer execution and involves both the dynamic or recorded method. A system at Amherst, for example, includes a hardware device that triggers an interrupt after execution of each instruction in the computer. As with any interrupt, the computer saves the address of the next instruction and branches to an interrupt

service routine. The service routine, recovers this address. prints it out along with the contents of the machine registers, and then returns to program exectution [96]. A real-time trace method is also used where a microprogrammed computer halts the microcomputer under test, retrieves the program counter and instruction register, and saves them in a trace buffer in main memory each time the program counter is changed. Then, the computer under test is allowed to execute the instruction. At the end of a test run, the recorded data is processed by a program that generates execution time and frequency information [36].

Traces are related to some of the other debugging considerations which have been previously brought out as follows:

- Software Tolerance Requirements. Traces greatly increase program execution time and in most cases (except for some monitor traces) require memory space [67].

- Language Considerations. The information produced by a trace depends on the level at which the trace is being applied. At the machine code level, it might include program counter, instruction being obeyed, and contents of accumulator and index registers. At the assembly language level, output might be in terms of the literals used by the assembler. In an ALGOL/W debugging system, output consists of the source text line together with the values of all variable and function procedures required for expression evaluation. The display may also contain any newly assigned values, the outcome of conditional tests, procedure calls, and the correspondence between formal and actual parameters [67].

- Interactive vs Batch Systems. Traces are not normally employed as a part of interactive debugging. This is because other methods are easier to use (e.g., programmed-in dumps or breakpoints) [30]. In the batch mode, two methods of obtaining a trace are employed. The computer operator, when it appears obvious that a batch job is in a loop or experiencing other problems, can manually interrupt and request a trace, obtain a post-mortem dump, then terminate the job and return the trace output to the programmer [116]. Alternatively, the programmer can request a trace by specifying certain control parameters during a rerun of the job [61].

- System Hardware/Software Architecture. Because of timing and throughput degradation associated with traces, they really are not applicable to real-time systems [93, 37, & 32]. They are applicable to both large-scale and minicomputer systems, but are rarely available for minicomputers. For example, the ODT on-line debugger for the PDP-11 contains various debugging tools, but no trace capability [111]. For microcomputers, the monitor trace, which uses a secondary monitoring computer, is frequently employed [96].

- Software Errors. The primary error symptoms which might lead to the use of a trace are the following:
  - Error message
  - Inputs not accepted
  - Missing outputs
  - Excessive output
  - Excessive run time
  - Infinite loop

3.3.2.3 Set-Use Matrix/Cross-Reference Analysis Tool

This program is usually associated with compilers or assemblers and provides information on the usage of programs, labels, tags, data variables, constants, subroutines, macros, or other program elements. The information produced usually includes the name, a set-use indicator, and the location(s) in the program where the identified item is set and used. The set-use matrix provides a static description of data access. It can be obtained for a CPCI or a system of CPCIs by using a system monitor which examines the compiler-generated output for each program module or CPC. This type of set-use matrix, also referred to as a cross-reference analysis, is then generated for all data variables used and set by each of the CPCs in the CPCI. It is also possible to obtain cross-reference information on other system components, such as CPCs, file names, and macro names [13 , 34].

This system tool is very useful in debugging interface problems on large-scale systems. It enables candidate programs for data-setting problems to be determined, indicates routines that reference the error producing program, shows unused variables, defines file usage by system components, etc.

Cross referencing tools are related to other debugging considerations as follows:

- Software Tolerance Requirements. This tool appears to not affect the tolerance requirements of a system because it is used on a stand alone basis.
- Language Considerations. This type of tool is constrained by a specific programming language, making it inapplicable for multi-language development projects [13].
- Interactive vs Batch Systems. This tool is not interactive. It usually produces its output on a selective basis and no user decisions are involved.

- System Hardware/Software Architecture. The set-use matrix for an individual program is a feature of many compilers but cross-referencing between programs is more likely to be found in large-scale systems.

- Software Errors. This tool is an extremely valuable tool to be used by the programmer during module test/debug and software integration activities. It is especially important for the detection of interface errors between system components.

3.3.2.4 Hardware Debugging Tools

Hardware debugging tools are hardware devices or features, usually distinct from the operational hardware, which provide information on a program's operation. Hardware debugging tools can be categorized into six classes as follows:

- Hardware monitors. These are special hardware tools attached to the target computer for monitoring a program's execution. Capabilities provided by these devices include:

  - Synchronizing on a specific instruction and sequentially displaying subsequent instructions.
  - Triggering an interrupt after execution of each instruction in the user program. (Interrupt service software can then recover the address and print it with the associated machine's registers, before returning to program execution.)
  - Providing monitoring of specific locations or data words by storing bus information of each occurrence.
  - Storing a continuous history of information and halting on a specific condition.
  - Timing selected sequences of program code.
  - Event counting (e.g., accesses to an address or peripheral.)
  - Collecting sets of addresses or instructions which refer to a specific location.
  - Specifying a set of addresses which will trigger bus data storage.
  - Simulating non-existent hardware.
  - Replacing the system clock.
  - Employing microprogramming for program monitoring.
  - Slowing down and speeding up the execution rate of the computer [37, 39, 96, 73].

- Monitor computers. A monitor computer is one that communicates with the target computer, simulating the inputs and monitoring the outputs. Capabilities provided by the monitor computer include:

  - Simulating realtime events (e.g., peripheral transfers and interrupts).

  - Employing conditional halts to stop execution and return to the user whenever a specific set of conditions hold. (The advantage of this feature lies in the fact that the conditions are evaluated during each instruction cycle at run time, as opposed to the more traditional break-point insertion for real-time software which is effectively performed at load time.)

  - Halting the computer under test, retrieving the program counter and instruction register, saving these in a trace buffer each time the program counter is changed, and allowing the computer under test to execute the instruction. At the end of a test run, the buffer is processed by a program that generates execution time and frequency information.

  - Slowing down and speeding up the execution rate of the target computer [87, 36, 20 & 21].

- Computer Emulator. A computer emulator is microprogrammed to operate the instruction set of the target computer. The target computer software can then be operated in the emulator exactly as it operates in the target computer. (Paragraph 3.1.8.3 presents a more detailed description of emulation.) The principal debugging usefulness of an emulator results from the visibility obtained. This is especially useful for debugging avionics flight computer software, because these computers provide output only via the computer's input/output interface [27 , 74 , 2 , & 16].

- Computer Simulators. A computer simulator is programmed to operate the instruction set of the target computer by simulating that computer's instruction execution. It may also be programmed to simulate interfacing computers in a network system. The instruction simulator-type of computer possesses all of the debugging utility described for computer emulators. For example, in a simulation of the Brassboard Fault-Tolerant Spaceborne Computer using an Interpretive Computer Simulator (ICS) system, diagnostics provided in-depth analysis of the microprogram being validated. All of the flip-flops, register values, and significant simulation variables are printed making the execution of each micro-instruction completely visible to the analyst. The network simulator-type of computer appears to the target computer as a real communication network, complete with terminals and operators. This computer simulator generates all queries and responds to all answers by simulating the characteristics of the actual terminals or operators. Use of a computer permits reproducibility and

creation of system evaluation reports. By recording every message transmitted (control and data) and every message received, along with the system time, it becomes possible to play back an entire test scenario as a way of isolating errors [79, 88, 66, & 63].

- Hardware Breakpoints. Any standard computer can be equipped with a hardware breakpoint debugging aid by wiring a comparator module to a control switch panel. This hardware addition matches the contents of the memory address register with the contents of the control switches to halt the computer at the selected breakpoint address [1]. At the point where execution is halted, a trap/breakpoint dump (see Paragraph 3.3.2.1) or other debugging action may be requested.

- Graphics. Graphics devices are usually part of the operational hardware and do not fit the definition of a hardware debugging tool. These devices, however, can provide debugging information in such a unique manner that they are a special exception to the definition. When combined with sophisticated analysis software, they can provide a means of pictorially simplifying a program's logic. For example, RAND's EXDAMS system's output device is a cathode ray tube (CRT) display, and all debugging and monitoring aids utilize its two-dimensional and high-data rate capabilities. Some aids, in addition, use the CRT's true graphic (point and vector) and dynamic (time-variant) capabilities. The input devices consist of a keyboard and some type of graphical pointing device (e.g., a light-pen). Some of the EXDAMS displays include a tree-like flow back analysis display of the logic preceding a specific statement in a program and a motion-picture display of values, source code and execution in terms of nodes [90]. In IBM's GBUG system, the user executes a debugging step by pointing at an associated displayed item with a light-pen and closing a switch. The debugging program performs the requested task, updates the display, and waits for the next inquiry [95].

Hardware debugging tools are related to other debugging considerations as follows:

- Software Tolerance Requirements. Hardware monitors, monitor computers, breakpoints, and graphics generally do not degrade the timing, storage, or throughput requirements of software. The computer simulators and emulators degrade the timing of a program and I/O instructions may be especially troublesome.

- Language Considerations. Languages and compilers have no apparent relation to hardware debugging tools.

- Interactive vs Batch Systems. All of the types of hardware debugging tools may apply to interactive systems, with graphics requiring on-the-spot decisions and inputs. Computer emulators and simulators are the only forms of hardware tools applicable to the batch-mode of debugging.

- System Hardware/Software Architecture. All of the types of hardware tools apply to large scale, minicomputer, microcomputer, real-time, and distributed systems. Unique factors exist in these various systems requiring specific analysis for each configuration, as follows:

  - Real-Time Systems. A form of hardware monitor used in real-time systems is the memory bus monitor. For example, one such monitor exists which provides storage for 16 memory addresses, 16 data words, and 16 copies of two control words. A register set allows for selecting address limits of interest, and/or for selecting the addresses for an on/off address-dependent signal that can be used to condition other monitoring events. A base address register provides constant conversion for relocatable code, and a timer is available whose rate and duration are selectable. A display panel provides visual monitoring of selected data, and a set of manual controls. Most of the capabilities of hardware monitors previously described are available through such a device [39].

    Computer simulators have also been used for the development of real-time programs. A typical simulator performs simulated peripheral transfers through a user controlled device-counting scheme. This delays transfers until such time as the simulator has executed a number of instructions comparable to the number the actual machine would have executed during the course of a transfer. It is worth noting that the simulated periphals may be represented on the host machine by any convenient peripheral for analysis of data transfers.

    The correspondence between the simulated peripherals and the host's actual peripherals is made by the user. A typical system may also provide a user-controlled scheme for simulating peripheral interrrupts, and for halting the simulated execution of a program on any specified combination of possible occurences (e.g., the values of program registers, the number of instructions executed, relationships between the contents of memory locations). The conditions for halting may be changed at any time by the user issuing commands from a terminal, thus providing a powerful debugging aid [21].

  - Minicomputer Systems. Hardware monitors are also usefully employed for debugging minicomputer systems. For example, a stand-alone hardware device that can be connected to a PDP-11 running any software package was developed for the Energy Research and Development Administration. It has most of the features of the PDP-11's On-line Debugging Technique (ODT), plus some unique features. Since it operates in a real-time, on-line, dynamic mode, it does not affect any timing or add overhead to the system. Traps are not required to drive it, and it is continually available. The PDP-11 debugging tool can display information

modify memory, generate interrupts, simulate non-existing hardware, stack and display data or address values, count accesses to an address, halt when address accessed, and display the unibus signals--all in real time [83].

An example of computer simulation of minicomputers is Applied Data Research's MIMIC system. It runs interactively on the PDP-10 and handles the following computers: Digital Equipment Corporation PDP-11, PDP-8/E, PDP-7/9/15; Data General Corporation NOVA, SUPERNOVA, NOVA 800, NOVA 1200, and SUPERNOVA 5c; and Foxboro FOX-1. MIMIC has 19 debugging commands available to perform such functions as modifying data, search for data, dumping data, and breakpoints [79].

- Distributed Systems. The Honeywell Communications Environment Emulator (HCEE) is a program for the H1200 computer which simulates a user-defined communications network. The primary purpose of the program is to aid in the checkout of communications software residing in a target computer.

  The simulator resides in its own processor, with the system under test in a second processor. The two computers communicate via standard communications hardware. The target system thinks that it sees a real communication network, with terminals and operators. Actually, HCEE generates all queries and responds to all answers by simulating the characteristics of the actual terminals and their operators. Use of a computer permits reproducibility and creation of exhaustive system evaluating reports. By recording every message transmited (control and data) and every message received, along with the system time, it becomes possible to play back an entire sequence as a way of validating changes to the target system. These records become the basis for a complete series of reports describing the operation of the system in detail. The computer also permits automatic message generation and response analysis based on formats defined by the user [66].

- Microprocessor Systems. There are a number of unique hardware debugging tools for microprocessors. One kind of equipment that can often be justified is a portable front panel such as the Intel 820 or the Ucca International, Inc. Microcomputer Analyzer (YA-1). Although intended to check out hardware in the field, such basic test equipment provides enough displays and controls to make software checkout in a laboratory environment possible.

  The computer evaluation kit represents a more advanced development concept than the prototype. Such kits nearly always have a ROM-based monitor that gains control when power is applied. Some kits offer front-panel controls and displays; other depend on software to handle all operations. Many kits show the contents of the address and data buses, as well as some control signals, on LED or liquid crystal displays. These displays may be nearly worthless if the kit does not also provide some way to halt the CPU and execute single instructions [106].

The monitor is usually able to communicate with a standard terminal, operating at 10-to-30 characters-per-second. When power is applied, the monitor sends a prompting character, usually a question mark, to the terminal. The user then chooses from a set of commands, each represented by a single letter. Commands are often included for loading program contents into memory, dumping memory contents on the printer, initiating the execution of a program, and altering memory contents. The user's program must usually end with a transfer instruction back to the monitor program. In this simple protocol, all addresses and data values are entered as hexadecimal constants [106].

The monitor program often provides other commands (e.g., reading and writing a tape, copying contents of memory, and finding a particular bit pattern in memory). One command especially useful for debugging programs is the breakpoint command, which plants a call to the monitor at a user-specified location in the program being debugged. When that particular point in the code is reached, the monitor regains control and displays the CPU's register contents. Many monitors even allow the program to resume execution at the breakpoint location. This can be an invaluable aid to debugging a loop that fails only occasionally [106].

Even if other equipment is to be used in the final product, the evaluation kit can often be used as a test bed for checking out interface hardware and software before integrating them with the rest of the system. If the evaluation kit is to be used for some limited software development, its monitor can be plugged into the prototype to provide a common debugging tool. A new, Intel personal programming tool carries the evaluation kit concept one step further. It not only enables a user to enter programs manually into memory, and execute them, but also contains provisions for adding a programmer PROM and external interface circuitry [106].

Microcomputer development systems (MDSs) represent the most sophisticated capabilities of the microcomputer development environment spectrum. They are available from numerous sources ranging from the giant semi-conductor manufacturers to small systems support firms having only a single product line. A typical minimum MDS configuration might consist of a microcomputer, 16K bytes of memory, and a teletypewriter terminal. Software typically includes a text editor, assembler, and debugging monitor. Like minicomputers, MDSs are offered with a wide range of optional equipment, including more memory, a disk and associated disk operating system, and faster peripherals such as CRTs and line printers [33].

A novel kind of peripheral that can be attached to a fully equipped MDS is an in-circuit emulator. An emulator consists of a suitably programmed MDS that has an interface terminated in a 40-pin plug; the plug is inserted into the microprocessor socket of the system under test, in place of the microprocessor itself. Then, while the system is operating normally, the MDS collects data from it and tracks special conditions. The emulator can, for example, set up hardware conditions under which the system will halt and transfer control to the MDS operator. Then, with appropriate commands provided in the ICE software support package, the user can display and change register contents, issue special instructions, or change memory contents in the system under test. The in-circuit emulator thus represents an efficient way to overcome the inherent problem with micros: all the system registers are inside the CPU and can not be displayed in real-time without disturbing the hardware, software or timing of the system being tested [106].

The in-circuit emulator is only one example of a whole gamut of test instruments that have been newly developed for use with microcomputers. Indeed, many of these tools (e.g., logic analyzers) can be productively applied to the debugging of real-time systems based on large computers [106].

Logic analyzers are general purpose analysis and debugging tools used by microprocessor developers. The basic principle behind most logic analyzers is as follows. Probes bring in signals from the equipment under development. Each three-position-toggle-switch on the front panel of the analyzer can be set either to recognize a 1 or 0 signal on any given input line or to ignore that input. The analyzers memory is enabled, however, when the input conditions match the conditions specified in the switches. The inputs are collected in the analyzers memory, one for each clock pulse. The clock is synchronized to another clock located in the system under test. As a new word is written, the oldest word is purged from memory. A display formatter then acquires data from memory and produces digital displays, timing diagrams, or a map [65].

- Software Errors. The primary error symptoms which might lead to the use of hardware debugging tools are the following:
  - Incorrect results
  - Abonormal initiation
  - Abnormal termination
  - Inputs not accepted
  - Missing outputs
  - Excessive run time
  - Infinite loop

3.3.2.5 Breakpoint/Traps

Breakpoints and traps are debugging features associated with the interruption of a program's execution. Breakpoints are interruptions generated by hardware or software when locations or statements are executed or referenced. Traps are interruptions generated by software when specific conditions are met, regardless of where the condition is within a program [79].

There are various types of breakpoints/traps as follows:

- Unconditional. For breakpoints, all executions or references to a certain location or statement result in an interruption. For traps, all cases where a certain condition is met result in an interruption [3, 54].

- Conditional. For breakpoints, selective execution or references to a certain location or statement result in an interruption. For traps, selective cases, where a specific condition is met, result in an interruption [57, 11, 79].

- Frozen state. In this mode, breakpoints or traps result in the program execution being suspended in a status that exactly reflects the result of all machine instructions executed prior to encountering the breakpoint or trap. Other programs inside the computer still continue to run independently of the frozen program, the frozen state applies to a program, not to the computer itself. It can be controlled in several ways depending upon the computing environments, and whether the system is interactive or batch [40].

- Hardware (Breakpoint only). A hardware breakpoint is accomplished by wiring a comparator module to a control switch panel. This hardware addition matches the contents of the memory address register with the contents of control switches in order to halt the computer at the selected breakpoint address [1].

Single or multiple breakpoints and traps may be active at the same time. In addition, they are associated with some action to follow the interruption. These actions include [44, 57, 11, 6, 3, 54, 8, 79]:

- Displays. The address of the breakpoint or the instruction/ statement resulting in a trap may be printed.

- Changes. A value at a breakpointed location may be changed to another value.

- Branches. The execution sequence may be altered.

Breakpoints or traps are related to debugging considerations as follows:

- Software Tolerance Requirements. Breakpoints and traps generated by software degrade a program's execution time and require storage for the instructions [93]. Hardware breakpoints have no limitations in this regard [1].

- Language Considerations. Debugging systems exist which allow breakpoints and traps to be requested in machine-level language, high-level languages, or totally user-oriented terminology [44, 6, 30].].

- Interactive vs Batch Systems. Breakpoints and traps are used primarily on interactive systems because they lead to additional debugging actions [30].

- System Hardware/Software Architecture. All types of systems use breakpoints or traps in some form [57, 106, 93, 32, & 111]. For example, the PDP-11 vendor-supplied ODT system provides two breakpoint features: a search for specific bit patterns, and a search for words which reference a specific word [111]. Hardware breakpoints are frequently used in debugging real-time systems [118 distributed systems [37], minicomputer systems [1], and microcomputer systems [106]. Also, in a time-sharing system, terms such as storage take on different meanings according to the current state of the system. A particular segment of code may be in one or another area of main storage, or it may be on an external storage device, depending on the particular point in execution of the program. Because of this, the programmer must be allowed to qualify breakpoint or trap requests to specify the exact item or the exact location of the item of interest. This is usually provided by a related request which allows specification of real or virtual storage, tasks identification, external devices, program modules, etc. [8].

- Software Errors. The primary error symptoms which most commonly lead to the use of breakpoints or traps are the following:
  - Error messages
  - Incorrect results
  - Incorrect output format/sequence
  - Excessive output

3.3.2.6 Reversible Execution/Backtracking

This debugging aid, also referred as flowback analyses [6] and traceback [67] allows the debugger to determine how control reached a point in the program where an error occurred. Terminating program execution at the location where an error message is output, allows the logic to be retraced. Retracing program execution may be specified by number of execution steps, by labels, or by values of Boolean expressions.

Backtracking tools are related to other debugging considerations as follows:

- Software Tolerance Requirements Considerations. This technique is not applicable to time critical operations (i.e., data items referred to in retraced instructions are set on a time or interrupt basis). Also, this capability generally requires a large amount of core storage and is not applicable to core-bound systems [94].

- Language Considerations. This capability has been used in systems which are essentially language independent in usage (e.g., HELPER [57], AIDS [44], and EXDAMS [6]).

- Interactive vs Batch Systems. The capability is not available in existing batch systems.

- System Hardware/Software Architecture. The backtracking tool appears to have been employed only on large scale systems. The AIDS system was developed for the CDC 6600. The EXDAMS and HELPER systems are essentially adaptable to more than one machine, but are such a large systems that they too are applicable only to large scale machines.

- Error Symptoms. The backtracking tool is actually applicable to most primary error symptoms but would usually be applied for the following:
  - Error messages
  - Incorrect results
  - Abnormal terminations

3.3.2.7 Program/Data Modifiers

The Program/Data Modifier tool is used to set or modify the contents of specific memory locations, the values of program variables, or program statements.

The following forms of this tool exist:

- Interactive Modifications. Most interactive debugging systems possess the capability to set or modify a program or its data via a user oriented set of commands. For example, an IBM system [8] allows the debugger to DEFINE certain private symbols, manipulate them with the SET command, and make decisions based on other values. The MANTIS system [3] allows the debugger to change the values of variables and arguments by use of AT, ON CALL, and BEFORE RETURN FROM commands, in conjunction with FORTRAN-like specifications (e.g., J = 'HELLO', DOUBLE = 5765.7890D34, X(5) = 1, and &(4,5) = 5).

- Recompilation/Correction. In batch systems, programs may be recompiled or changed via symbolic or machine level correctors to provide a program variables or statement change capability. Frequently, this tool is used as a breakpointing capability by allowing instructions to be changed to cause the program to halt when certain instructions and/or conditions are reached. Some interactive systems possess a capability known as "partial recompilation" which permits source language modifications to be made while nearly eliminating the requirement to completely recompile an entire program [4].

- Hardware Modifications. The hardware monitor tools discussed previously frequently enable the contents of memory locations to be set or modified [39, 106, 66 & 83].

The Program/Data Modifier tool is related to some of the other debugging considerations which have previously been brought out as follows:

- Software Tolerance Requirements. This tool requires the program modified to be in a static state, and obviously affects program timing.

- Language Considerations. Debugging systems exist which allow program/data setting or modification to be performed in machine-level, higher level, or user-oriented terminology [8, 67 & 30].

- Interactive vs Batch Systems. This tool is available for both interactive and batch systems. It is much more widely used in interactive systems.

- System Hardware/Software Architecture. Large-scale, real time, minicomputer, and distributed systems employ both hardware and software versions of this tool. Microcomputer systems use only hardware program/data modification tools [106].

- Software Errors. This tool is actually applicable to most of the primary error symptoms but is most commonly applied for the following errors symptoms:

  - Incorrect results
  - Inputs not accepted
  - Incorrect output format/sequence
  - Missing output
  - Excessive output

3.3.2.8 Data Recording/Reduction

Data recording/reduction tools are used primarily to minimize the execution timing impact when large amounts of data need to be accumulated and output for debugging purposes. These tools accomplish this by tranferring large quantities of output data to an auxiliary mass storage device (typically magnetic tapes) in an unformatted form (thus reducing required processing time). The data is later processed, selectively or in total, into a user or debugger-oriented form. Data recording/reduction tools can be categorized into the following classes:

- Program Execution-Oriented Recording/Reduction. This class of tools records information on the processing steps taken during execution of a program. The recording is subsequently reduced and displayed to provide such debugging features as: source code tracing; flow-back analysis (backtracking from the point of an error); motion picture type displays of values, source code execution, and node trees. RAND's EXDAMS and Univac's TALK systems are examples of this tool [6 & 86].

- Input-output Recording/Reduction. This class of tools records information on program's inputs and outputs or the computer subsystem as a whole. The recording is subsequently reduced and displayed to provide information on key interfaces. This technique has been employed since early development of large-scale systems. The SAGE Air Defense System, for example, recorded radar inputs which were later reduced and evaluated by data reduction programs, and which could also be replayed to precisely duplicate the original processing [45].

Data recording/reduction tools are related to other debugging considerations as follows:

- Software Tolerance Requirements. While the intent of these tools is to reduce execution timing impact, the recording logic and auxiliary storage time contributes to degrading of program timing and storage [93].

- Language Considerations. The program execution-oriented recording/reduction tools are ideally suited for the provision of readable, understandable output. The translation of recorded data into the higher-level source language is shown by the EXDAMS and TALK systems to be an easily attainable capability. Formats more user-oriented than source language may be additionally supplied by these systems.

- Interactive vs Batch Systems. Both types of data recording/reduction capabilities are employed by batch and interactive systems.

- System Hardware/Software Architecture. The program execution-oriented class of this tool is applicable to large-scale, mini-computer, and distributed systems. The input/output class is applicable to all types of systems, with monitor computers usually performing the task on microcomputer systems [24]. The input/output class is probably the primary tool used in debugging real-time and distributed systems.

- Software Errors. Data recording/reduction tools are actually applicable to most of the primary error symptoms but is most comonly applied to the following:

  - Incorrect results
  - Inputs not accepted
  - Incorrect output format/sequence
  - Missing output
  - Excessive output

3.3.2.9 Comparators

Comparators are computer programs used to compare current versions of software and data with previous versions of the same software. The comparison is usually performed on external input sources, (e.g., between two system master tapes, data base tapes). Comparators are useful in disclosing errors existing only on one software configuration or version [116,47,60 & 85].

Comparators are related to other debugging considerations as follows:

- Software Tolerance Requirements. Comparators have no apparent relation to tolerance requirements.

- Language Considerations. Comparators usually compare data in the form in which it exists although some have a capability for minor formatting features.

- Interactive vs Batch Systems. Comparators are applicable to both interactive and batch systems.

- System Hardware/Software Architecture. Since comparators are basically stand-alone, off-line tools they are applicable to all types of systems.

- Software Errors. All of the primary error symptoms presented in Paragraph 3.2.2 might lead to the use of comparator tools.

3.3.2.10 Problem Status Reporters

These tools maintain a file of software problems found and resolved. The tool helps verify that problems found are eventually corrected, and helps maintain a history of the debugging process for analysis purposes (e.g., for tool evaluation) [13, 60 & 28].

This tool is a stand-alone program which can operate completely independent of the system on which debugging is performed. The other debugging considerations which have been previously discussed have no relation to this tool.

3.3.2.11 Test Deck List Tool

This tool is used to list the complete test deck employed when a problem was encountered. It is normally run before the test is operated, and is attached to the discrepancy report form associated with the problem if one is encountered during program execution. It aids the problem duplication process [116, 13].

This tool is a stand-alone program which is applicable mainly to batch systems. It is usually not applicable to real time, distributed, or microcomputer systems. The tool has no apparent relation to other debugging considerations previously discussed.

3.3.2.12 System Status Summary Tool

This tool produces a summary of the hardware and software configurations at the time a problem is encountered. It can contain such information as the computer system involved (in a multi-computer system), the magnetic tapes used, disc configuration, other equipment status, the version of the operating system used, the version of the corrector deck used, the version of any new programs used, the data base ID, and the compool ID. It is very useful in the problem duplication process [116, 13].

This tool is a stand-alone program which is applicable to both batch and interactive systems in some form. The other debugging considerations previously discussed have no relation to this tool.

3.3.2.13 Checkpoint Tools

These tools are features of the system support system which allows the capability to save an image of memory or disc at any point during the operation of single or multiple programs. The saved data may subsequently be reloaded and processing can be initiated from the checkpoint position. This tool is especially useful in the problem duplication process. There are two main forms of the tool, as follows:

- System Save/Restore. The contents of memory, general registers, program counters, disc and equipment status are saved on tape or removable disc packs. They may subsequently be restored and program execution continued [116, 69].

- Data Base Save/Restore. The updated data base is saved on tape and may be subsequently restored. The programs which operate on this data after the checkpoint can be re-executed from this point [116,38].

These tools have no apparent relation to the other debugging considerations discussed previously. For microcomputer systems, it is assumed an associated tool, such as a monitor computer, would be involved.

3.4 DEBUGGING METHODS

The literature survey presented very little information on the methods employed by debuggers to isolate and solve problems. The thought processes required of the programmer involved in debugging problems have been described by J. Schwartz [87]. Many sources discussed the need for problem duplication in the debugging process, and the need for predefined debugging steps [6, 3]. In addition, many sources detailed on-line debugging systems and how they were employed [6, 79]. This discussion consolidates the debugging methods discussed in the literature, and includes descriptions of the following processes:

- Preparation
- Understanding
- Duplication
- Elimination
- Symptom accumulation
- Debugging data collection

3.4.1 Preparation Process

Before debugging is initiated there are several activities necessary for the performance of debugging in a batch-mode system, including the establishment of predefined steps to be followed by computer operators when software problems are encountered during program execution. Such steps include:

- Training computer operators to activate the debugging trace features, if available, on occurence of a tight, infinite loop.
- Dumping general registers and a memory map when a program terminates abnormally.
- Recording the hardware and software configuration at the time of the problem if a system status summary tool exists in the support system. If not available, this data can be manually derived and reported.

- Listing the test deck by operating the test deck listing tool, if available.

- Listing correctors active at the time of the problem.

- Operating a system or data base save/restore tool.

- Attempting to rerun the task, using backup tapes, etc, if the error appears to be a hardware malfunction.

The exact predefined steps to be established are a function of the software characteristics and the debugging tools available.

In submitting computer runs in a batch system, the job request form can be used as a vehicle to indicate what should be done for the various types of problems that may be encountered. The same types of steps noted previously can be requested, and in addition, the following requests might be helpful:

- Requesting dumps on the basis of error message category (see Paragraph 3.2.3).

- Requesting rerun of a task with sense switches set so that programmed diagnostics can be taken.

Some operating systems contain debugging capabilities to automatically take predefined debugging actions based on types of software failures. These capabilities include:

- Specifying an automatic dump on abnormal program termination. In addition, specification of a certain category of error messages which should result in a dump.

- Data recording of selective variables, tables, etc., on abnormal termination. (The recording can later be reduced and displayed if needed.)

3.4.2 Understanding Process

During the program integration phase of development, the person doing the debugging may not have coded the programs being debugged. A learning process, therefore, might be required as the initial step when a problem is found and reported. Furthermore, problems reported by persons other than the debugger usually require a communications process to take place. Finally, given that the debugger has an understanding of the programs involved in a reported problem and that a problem clearly exists, there remains the task of determining what should occur, and what is, or is not, being done. This is required for locating and resolving the problem. These two efforts are often the most difficult part of the understanding process for they frequently involve trying to interpret the required operational capability and user needs. The understanding process of debugging is aided by the following:

- Documentation. The need for quality requirements, development, product, and interface control documents was outlined in the discussion of software management methodology in Pargagraph 3.1.1. Documentation is a key input to the understanding process. When good documentation exists, the other tools and techniques might not be necessary for the understanding process.

- Written Communication. For complex problems, a system should exist which allows the debugger to communicate the most current understanding of the problem. The Software Analysis Report (SAR) method described in Paragraph 3.1 is one approach for communicating problem status. It is especially important for complex mathematics or hardware modeling problems where design and program implementation need concurrence before a solution can be attempted.

- Verbal Communication. A system should exist which allows verbal discussion of problems between the debugger, the customer, and the user. In the case of large-scale systems, discussions with interfacing contractors may also be required. The Configuration Control Board (CCB) meeting approach described in Paragraph 3.1.1 is one approach to formal verbal communication on software problems.

- Source Code Analysis. Analyzing program source code during the integration phase is the most obvious step in the understanding process. Top-down design and structured programming practices greatly benefit the understanding process. Also, some software tools exist which aid the source code analysis task. For examole, language preprocessing, reformatting the source code into a more readable format, and obtaining automated flowcharts to equate the source code with the product specification aid source code understanding.

- Set-Use Matrix/Cross-Reference Analysis Tool. This tool (described in Paragraph 3.3.2.3) is important for understanding the interface relationships between programs/subsystems. It also provides a link between the source code and interface control documentation.

- Graphics. Graphical display tools (described in Paragraph 3.3.2.4) can present decomposed versions of software in terms of nodes, models, and graphs when augmented by sophisticated software. These displays can simplify the understanding of a program's logic. Graohical capabilities are very problem-specific in nature and it is difficult to generalize on their usefulness for systems in general.

3.4.3 Duplication Process

One basic debugging requirement is problem duplication or repeatability. It is required so that: (1) a problem can be shown to exist, and (2) debugging tools can be applied during reruns to obtain more symptoms. If a problem cannot be duplicated, the debugging process is significantly more difficult. The following methods exist for duplicating a problem.

- Test Material Reconstruction. If a test deck listing (see Paragraph 3.3.2.11) accompanies the software discrepancy report, it can be used to reconstruct the exact inputs which cause the problem.

- Hardware/Software Configuration Reconstruction. If a system status summary listing (see Paragraph 3.3.2.12) was requested at the time a problem occurred, it can be used to insure the proper hardware and software configuration.

- Scripted Inputs. If a problem was encountered during testing on a test bed or monitor computer (see Paragraph 3.1.8.3 and 3.3.2.4), the inputs are often made in the form of a scripted magnetic tape. This tape can be used to reestablish the exact inputs which caused the problem.

- Data Recording Replay. If a data recording/reduction tool (see Paragraph 3.3.2.8) was active when the problem occurred, the recording tape can sometimes be replayed to duplicate the inputs.

- Checkpoint Restart. If a checkpoint tool (see Paragraph 3.3.2.13) was employed to save the system or data base, these may be restored and processing continued from the checkpoint.

3.4.4 Elimination Process

A vital step in the debugging process is the orderly elimination of possible factors related to the problem. This process of elimination is especially important when subtle, obscure symptoms are presented. For example, if it can be shown that a problem only exists in the latest version of a program, further investigation via the accumulation of symptoms will often require that only the updated portions of the program be analyzed.

The following methods can be employed to eliminate the factors involved in a problem:

- Reruns. The test can be rerun on the same computer, another computer, or with backup tapes to determine a machine problem.

- Reconfiguration. The test can be run with a prior version of the program, with an earlier operating system, with correctors deleted, or with another data base to determine whether the problem has always existed or is due to a recent change.

- Comparison. In association with the reconfiguration runs, comparator tools (see Paragraph 3.3.2.9) can be used to compare versions of programs, data bases, or correctors to determine differences between versions.

- Work-arounds. Work-arounds of suspected areas of code can be made via selection of different program options, recompilation, partial recompilation, correctors, or program/data modifier tools (see Paragraph 3.3.2.7) to isolate the problem through elimination.

- Breakpoints. Breakpoints may be inserted before and after suspected areas of problem code for taking dumps. The area can then be divided into halves and breakpointed again.

3.4.5 Symptom Accumulation Process

Having obtained an understanding of the problem, duplicated it, and reduced its proportions to a manageable level, the intricacies of debugging begin. The problem symptoms presented to the debugger must be examined so that the problem cause can be isolated and a solution can be envisioned. The following methods can be employed to aid this process:

- Determining Program at Fault. For large-scale systems a tool such as the set-use matrix/cross-reference analyzer tool (see Paragraph 3.3.2.3) can be employed to isolate the primary program to be investigated.

- Assignment of Proper Personnel. After identification of problem type, the proper personnel can be assigned to aid in the debugging. For example, a problem with incorrect math results might be assigned to both a math analyst and a programmer.

- Obtaining Extra Output. Usually not all of a program's normal output options have been requested when a problem is encountered. In reruns of the test, additional output can be requested. This might reveal the full ramifications of the problem.

- Determination of What Needs to be Measured. Sometimes this is the most difficult task of debugging. Documentation and source code can be examined to provide likely starting points for the symptoms accumulation effort.

- Employing Debugging Tools. The debugging tools described in Paragraph 3.3 can be used to accumulate symptoms until a solution is envisioned.

3.4.6 Debugging Data Collection Process

This process is a post-debugging activity which may not aid the current debugging process. However, it does aid in the debugging of future problems of the same type. The data collection process involves the collection of information on how a problem was debugged. There are two methods available to aid in this process:

- Software Analysis Reports. SARs, or expanded discrepancy or resolution report forms (see Paragraph 3.1.1) can be used to describe how a problem was debugged.
- Problem Status Report Tool. A problem status report tool (see Paragraph 3.3.2.10) can be used to maintain a file on the discrepancy, resolution, and software analysis reports on every software problem encountered.

The results of this process establish the necessary inputs for debugging evaluation studies. It is an inherent attribute of software technology that it improves only when we learn from our mistakes. Debugging evaluation studies should be conducted after each development effort using the debugging data collection to improve the debugging methodology. The study snould include analysis of the effectiveness of both tools and methods, and recommend improvements for future development efforts.

APPENDIX A - SITE SURVEY BIBLIOGRAPHY

ADVANCED RESEARCH CENTER

"BMDATC Data Processing Standards - ARC Data Processing Users Guide - Volume III;" TM-HU-212/003/00B; System Development Corporation; Hunstville, Alabama, May 1977.

"Control Data CYBER 70 Computer Systems Models 72, 73, 74 Version 4, 6000 Computer Systems Version 4, "Intercom Reference Manual" Publication No. 60307100E; Control Data Corporation; Sunnyvale, Calif.; September 1974.

"Control Data CYBER 70/Model 76 Computer System 7600 Computer System, Scope 2.1 User's Guide". Publication No. 60372600C; Control Data Corporation; Aden Hills, Minn.; June 1974.

"PEPE Development Program"; System Development Corporation; Santa Monica, Calif.; 5 March 1976.

ROME AIR DEVELOPMENT CENTER

"Debug and Trace Routines"; DB20; Honeywell Information System; Series 600/6000 GCOS; September 1972.

"FORTRAN Code Auditor - Users Manual; RADC-TR-76-395, Rome Air Development Center; Griffiss Air Force Base, New York; December 1976, Vol I A035778, Vol II, A035914.

"Introduction to RADC R&D Computer Facility"; RADC-TR-77-61; Rome Air Development Center; Griffiss Air Force Base, New York; March 1977, A038657.

"Multics Pocket Guide Command and Active Functions Series 60 (level 68) Software"; AW17; Honeywell Information Systems; April 1976.

"National Software Works: Overview and Status"; Robinson, R. A., Digest of Papers-Compcon Fall 77, 15th IEEE Computer Society International Conference; September 1977; 270-273.

"Programmer's Manual Volume III - Commands and Active Functions, MULTICS Software"; AG92A; Honeywell Information Systems; 1973.

"QM-1 Application/Benefits" Nanodata Corp; Williamsville, New York; 1977.

"Third Semi-annual Technical Report for the National Software Works"; CADD-7702-2811; Massachusetts Computer Associates, Inc.; Wakefield, Mass.; February 1977.

"Structured Programming Translators"; RADC-TR-76-257, Rome Air Development Center; Griffiss Air Force Base, New York; August 1976, A031427.

TELEMETRY INTEGRATED PROCESSING SUBSYSTEM

"Category I Test Plan Text"; Roberts, J.; TDN-VN-24900/001/00; System Development Corporation.

"CDC FORTRAN Extended Version 4 Reference Manual"; Control Data Corporation; CDC No. 60497800.

"CDC NOS 1.0 Reference Manual"; Control Data Corporation; CDC No. 60435400.

"Computer Program Development Specification for the Telemetry Integrated Processing System, Distributed Operating System"; CG 26000; System Development Corporation; 9 May 1977.

"SEL Real-Time Monitor Reference Manual"; Systems Engineering Laboratories; SEL Pub. No. 323-321001-001/323-086001-007.

"Shopping List of Test Tools/Test Aids"; Patterson, W.; TDN-VN-24900/006/00; System Development Corporation.

"Software Development Standards and Conventions for the Telemetry Integrated Processing System (TIPS)"; TM-VN-(L)-0040/000/00; System Development Corporation; 28 June 1977.

"Terminal Support System (TSS) Training Manual for Training Purposes Only"; Systems Engineering Laboratories.

"TIPS Test Approach"; Roberts, J.; TDN-VN-24900/007/00; System Development Corporation.

AIR FORCE SATELLITE CONTROL FACILITY

"Category I Test Plan/Procedures for Model 15.3"; TM-(L)-5773/004/01; System Development Corporation; 21 November 1977.

"EBC Operators Guide and Users Manual for Model 14.2"; TM-(L)-5536/704/00C; System Development Corporation; 22 November 1977.

"Programming Standards and Conventions for Block 3 Software"; Aerospace Corporation; 1975.

"Real-Time System DT&E System Test Plan Model 14.2"; TM-(L)-5536/642/00C; System Development Corporation; 16 June 1977.

"STAGES Users Manual"; TM-(L)-3230/135/02H; System Development Corporation; 20 December 1976.

"System IIB Parameter Test Tutorial"; TM-(L)-5048/706/00; System Development Corporation, 3 March 1977.

"System IIB Users Manual Model 15.2"; TM-(L)-5048/714/00; System Development Corporation, 3 March 1977.

"1230 Computer Operators Guide Model 14.2"; TM-(L)-5536/702/008; System Development Corporation, 19 August 1977.

"1230 Users Manual Model 14.2"; TM-(L)-5536/700/008; System Development Corporation; 29 August 1977.

"3800 Central Computer DT&E System Test Plan and Procedures for Model 15.1B"; TM-(L)-5048/609/02; System Development Corporation; 20 October 1976.

"3800 Programming Standards and Conventions"; TM-(L)-5567/000/00; System Development Corporation; 16 July 1976.

APPENDIX B - REFERENCES

[1] Anthoni, F.; and Hemelaar, A.; "A Hardware Breakpoint Debugging Aid for the PDP-8/I"; Decuscope; Vol. 12 No. 3; January 1974; 27-29.

[2] Anzelmo, Ralph Harry; and Kaye, Theodore Lawrence; "Emulation of the AN/UYK-20 Tactical Data Computer on the Burroughs D-Machine"; Naval Postgraduate School; Monterey, California; March 1977.

[3] Asbhy, Gordon; Salmonson, Loren; and Heilman, Robert; "Design of an Interactive Debugger for FORTRAN: MANTIS; Software-Practice and Experience, Vol. 3; 1973; 65-74.

[4] Ayres, Ronald B.; and Derrenbacker, Richard L.; "Partial Recompilation" Proceeding AFIPS Spring Joint Computer Conference; 1971; 497-502.

[5] Baker, F. T.; "System Quality Through Structured Programming"; Fall Joint Computer Conference; December, 1972; 339-343.

[6] Balzer, R. M.; "EXDAMS-Extendable Debugging and Monitoring System"; ACM Spring Joint Computer Conference; 1969; 567-580.

[7] Bell, Thomas, E.; "Objectives and Problems in Simulation Computers" Proceedings, FJCC.

[8] Berstein, William A.; and Owens, James T.; "Debugging In A Time-Sharing Environment"; ACM Fall Joint Computer Conference; 1968; 7-13.

[9] Berri, R. E.; "Specifying Milestones for Software Acquisitions"; AIAA/NASA/IEEE, ACM Computers In Aerospace Conference; Los Angeles, California; October-November 1977.

[10] Biche, P. W.; Morgan, D. J.; and Wattenberg, R. E.; "Independent Test and Evaluation Guidelines"; Logicon Corporation; 24 July 1974.

[11] Blair, James; "Extendable Non-Interactive Debugging"; Debugging Techniques in Large Systems; Prentice-Hall, Inc.; Englewood Cliffs, New Jersey; 1971; 93-115.

[12] Boehm, R.; McElean, R.; Urfig, D.; Some Experience with Automated Aids to the Design of Large-Scale Reliable Software; Proceedings, International Conference Reliable Software; 1975.

[13] Bratman, Harvey; and Finfer, Marcia C.; "Software Acquisition Management Guidebook: Verification"; TM-5772/002/02, System Development Corporation; Santa Monica, California; November 1977.

[14] Bratman, H.; "Structured Programming: Techniques for Developing Reliable Software Systems"; SP-3693; System Development Corporation; Santa Monica, California; December 1972.

[15] Brinch, Hansen, P.; The Architecture of Concurrent Programs; Prentice-Hall; New Jersey, 1977.

[16] Britt, Benjamin; Cooperband, Alvin; Gallenson, Louis; and Goldberg, Joel; PRIM System: Overview; University of Southern California; Marine del Rey Information Sciences Institute; March 1977.

[17] Brooks, F.; The Mythical Man Month; Addison-Wesley; 1975.

[18] Brown, A. R.; and Sampson, W. A.; "Program Debugging"; Macdonand and American Elsevier; 1973.

[19] Brown, P. J.; "UNRAVEL - A Programming Language to Put Intelligence Into Dumps"; Computer Journal; February 1973; 10-12.

[20] Burger, P.; "System Integration and Testing with Microprocessors Software Aspects"; IEEE Transactions Independent Electronic and Control Instruments; Vol. IECI-22, No. 3; August 1975; 364-367.

[21] Burnett, P.; Kidd, P.A.; and Lister, A.M.; "Simulation of Real-Time Program Faults"; Computer Journal; Vol. 17, No. 1; February 1974; 25-27.

[22] Carrow, John C.; "Structured Programming: From Theory to Practice"; Second International Conference on Software Engineering; San Francisco, California; October 1976; 370-372.

[23] Chen, A. T.; and Siebert, R.S.; "A Data-Triggerable Program Debugging Aid"; Computer Description; Vol. 12, No. 9; September 1973; 90-93.

[24] Conrad, Marvin; King, David; and Price, Robert; "The Integration of Microcomputer Hardware and Software Development Tools and Techniques"; IEEE Computer Conference; Washington, D.C.; September 1977; 201-208.

[25] Creveuil, M. Y.; and Engel, G. M.; "A System/Software Development Process For Minicomputers"; AIAA/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-November 1977; 378-383.

[26] Cuff, R.; A Conversational Compiler for Full PL/I; Computer J. 15; 2 May 1972.

[27] Dyke, Frank P.; and Strum, Walter A.; "Emulation of Embedded Spacecraft Computers"; AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, Los Angeles, California; October-November 1977; 460-464.

[28] Enabit, Robert S.; "A New Approach to On-Line, Run-Time Program Logic and Error Debugging Using Hardware Implementation"; Behavior Research Methods and Instrumentation; 1970; 33-37.

[29] Endres, Albert; "An Analysis of Errors and Their Cause in System Programs"; IEEE Transactions on Software Engineering; Vol. SE-1, No. 2; June 1975.

[30] Evans, Thomas G.; and Darley, D. Lucille; "On-Line Debugging Techniques: A Survey"; Proceedings-Fall Joint Computer Conference; 1966; 37-50.

[31] Falor, Kenneth; "Cross-Assemblers"; Mini-Micro Systems; September 1977; 72-74.

[32] Farber, David J.; "A Ring Network"; DATAMATION; February 1975; 44-46.

[33] Ferguson, H. Earl; and Berner, Elizabeth; "Debugging Systems At the Source Language Level"; Communications of the ACM; August 1963; 430-432.

[34] Fetty, James T.; and Roth, Martin S.; "Software Support Tools"; Intermetrics, Inc.; Cambridge, Mass.; 15 October 1976.

[35] Feylock, Stefan; and Donegan, Michael K.; "The Development of a Multi-Target Compiler-Writing System for Flight Software Development"; AIAA/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-November 1977; 447-452.

[36] Frakenberg, Robert; "Unraveling the Mystery in User Microprogramming"; Mini-Micro Systems; September 1973; 54-60.

[37] Fraser, A. G.; "A Virtual Channel Network"; DATAMATION; February 1975; 51-56.

[38] Fries, M. J.; "Software Error Data Acquisition"; Boeing Aerospace Company; Seattle, Washington; April 1977.

[39] Fryer, Richard E.; "The Memory Bus Monitor - A New Device for Developing Real-Time Systems"; Procedures AFIPS 1973; National Computer Conference; 75-79.

[40] Gagnoud, A.; and Ottavi, J.P.; "Frozen State for Computer Program"; IBM Technical Disclosure Bulletin; Vol. 17, No. 4; September 1974; 1066-1067.

[41] Gaines, R.; Compiler Construction for Debugging; Prentice-Hall; New Jersey; 1971.

[42] Green, Joseph H.; "Program Analysis - A Problem In Man-Computer Communication"; Harvard University; Cambridge, Massachusetts; 1969.

[43] Greenberg, M. L.; "DDT: Interactive Machine Language Debugging System Reference Manual"; AD-707406; University of California at Berkeley; 15 June 1969.

[44] Grishman, Ralph; "The Debugging System AIDS"; ACM Spring Joint Computer Conference; 1970; 59-64.

[45] Haentzchel, L. E.; "SAGE Recording - A Study of the SAGE Recording Capability - Past and Present"; TM-3254; System Development Corporation; Santa Monica, California; 30 December 1966.

[46] Hagen, S. R., et al; "An Air Force Guide for Monitoring and Reporting Software Development Status"; AD-A016-488; MITRE Corporation; September 1975.

[47] Hartwick, R. Dean; "Test Planning"; National Computer Conference; 1977.

[48] Hoare, C.A.R.; and Wirth, N.; "An Axiomatic Definition of the Programming Language Pascal"; Acta Information; Volume 2, 1973; pp. 325-355.

[49] Horning, J.; What the Compiler Should Tell the User; Compiler Construction An Advanced Course; Springer-Verlag; New York; 1976.

[50] Howley, Paul P.; and Scholten, Roger W.; "Test Tool Implementation"; AMA/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-November 1977; 372-377.

[51] Irvine, C.A.; and Brackett, John W.; "Automated Software Engineering Through Structured Data Management"; IEEE Transactions on Software Engineering; January 1977.

[52] Jackson, K.; and Prior, J.R.; "Debugging and Assessment of Control Programs for an Automatic Radar"; Computer Journal; Vol. 12, No. 4; November 1969; 303-306.

[53] Jensen, K.; and Wirth, N.; Pascal User Manual and Report; 4th Printing, Lecture Notes in Computer Science (18); New York; Springer-Verlag; 1974; pp. 88-103.

[54] Josephs, William; "An On-Line Machine Language Debugger for OS/360"; ACM Fall Joint Computer Conference; 1969; 179-186.

[55] Kimbleton, Stephen R.; "The Role of Computer System Models Performance Evaluation"; University of Michigan; Communications of the ACM; July 1972.

AD-A069 541 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2
SOFTWARE DEBUGGING METHODOLOGY. VOLUME III. LITERATURE AND SITE--ETC(U)
APR 79 M FINFER, J FELLOWS, D CASEY F30602-77-C-0165
UNCLASSIFIED RADC-TR-79-57-VOL-3 NL

3 OF 3
AD
A069541

END
DATE
FILMED
7-79
DDC

[56] Kirsch, B. M.; "An Improved Error Diagnostic System for IBM System/360-370 Assembler Program Dumps"; Report OSU-CISRC-TR-74-3; Ohio State University; Columbus, Ohio; June 1974.

[57] Kulsrud, H. E.; "Extending the Interactive Debugging System HELPER"; Debugging Techniques in Large Systems; Prentice-Hall, Inc.; Englewood Cliffs, New Jersey; 1971; 77-91.

[58] Lampson, B. W.; Horning, J. J.; London, R. L.; Mitchell, J.; and Popek, G.; "Report on the Programming Language Euclid"; SIGPLAN Notices; Volume 12, Number 2; February 1977.

[59] London, R. L.; "A View of Program Verification"; Proceedings International Conference on Reliable Software"; April 1975; 534-545.

[60] Miller, James, R.; "A Report on the State-of-the-Art of Verification, Validation, and Certification of Computer Software Development and a Guide to its Application"; Scientific Applications, Inc.; March 1977; 28-36.

[61] Mitsuhashi, J. M.; and Ngou, L.; "System IIB Parameter Test Tutorial"; TM-(L)-5048/706/00; 11 January 1974.

[62] Mullin, Frank J.; "Considerations for a Successful Software Test Program"; AIAA/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-November 1977; 68-73.

[63] Myers, Glenford J.; "Composite Design: The Design of Modular Programs"; International Business Machines Corporation; 29 January 1973.

[64] Ogdin, Carol A.; "Project Control"; Mini-Micro Systems; November-December 1977; 118-127.

[65] Ogdin, Carol A.; "The Logic Analyzer"; Mini-Micro Systems; September 1977; 62-70.

[66] Pearlman, Jack M.; Snyder, Richard; and Caplan, Richard; "A Communications Environment Emulator"; Proceedings AFIPS Spring Joint Computer Conference; 1969; 505-512.

[67] Poole, P. C.; "Debugging and Testing"; Advanced Course On Software Engineering; Springer-Verlag; 1973; 278-318.

[68] Pullen, E. W.; and Shutter, D. F.; "MUSE: A tool for Testing and Debugging a Multi-terminal Programming System"; Proceedings AFIPS Spring Joint Computer Conference; September 1968; 491-502.

[69] Ramamoorthy, C. V.; and Ho, S. F.; "Testing Large Software With Automated Software Evaluation Systems"; Proceedings International Conference On Reliable Software; Los Angeles, California; April 1975; 382-394.

[70] Reed, E. F.; "Avionics Software Development - Experiences From the B-1 Program"; AIAA/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-November 1977; 112-118.

[71] Reifer, D. J.; "Automated Aids For Reliable Software"; SAMSO Report TR-75-183; Aerospace Corporation; El Segundo, California; August 1975.

[72] Richards, Paul K.; "Developing Design Aids For An Integrated Software Development System"; AIAA/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-November 1977; 435-441.

[73] Saal, H. J.; and Shustek, L. J.; "On Measuring Computer Systems By Micro-programming"; Microprogramming and Systems Architecture: International Computer State-of-the-Art Report; Maidenhead; Berks, England; Infotech Information; 1975.

[74] Sager, G. R.; "Emulation for System Measurement/Debugging"; Proceedings of the IFIP Conference on Software for Minicomputers; Kerzthely, Hungary; September 1975; 107-123.

[75] Saib, Sabine H.; Benson, Jeoffrey P.; and Melton, Richard A.; "Software Quality Laboratory"; General Research Corporation; Santa Barbara, California; November 1977.

[76] Schiffert, C.; Snyder, A.; and Atkinson, R.; "The CLU Reference Manual"; Massachusetts Institute of Technology Project MAC; June 13, 1975; unpublished.

[77] Schwartz, Jacob T.; "An Overview of Bugs"; Debugging Techniques in Large Systems"; Prentice-Hall, Inc.; Englewood Cliffs, New Jersey; 1971; 1-16.

[78] Stillman, Rona B.; and Leong-Hong, Belkis; "Software Testing for Network Services"; COM-75-100774; July 1975.

[79] Supnick, Robert M.; "Debugging Under Simulation"; Debugging Techniques In Large Systems; Prentice-Hall, Inc.; Englewood Cliffs, New Jersey; 1971; 117-136.

[80] Sylvester, Richard J.; "Elements of the Computer Program Development Plan"; AIAA/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-November 1977; 18-22.

[81] Teichroew, D.; Hershey, Ernest; "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems"; IEEE Transactions on Software Engineering; January 1977.

[82] Thayer, T., et al.; "Software Reliability Study"; RADC-TR-76-236; Final Technical Report; August 1976. AD# A030798.

[83] Tolmie, D. E.; and Gallegos, M. E.; "PDP-11 Debugging Tool"; Los Alamos Scientific Laboratory; New Mexico Energy Research and Development Administration; August 1976.

[84] van Wijngaarden, A.; Mailloux, B. J.; Pack, J.E.L.; Koster, C.H.A.; Sintozoff, M.; Lindsey, C.H.; Meertens, L.G.L.T.; and Fisker, R.G.; "Revised Report on the Algorithmic Language Algol 68"; SIGPLAN Notices; Volume 12, Number 5; May 1977; pp. 1-70.

[85] Venese, Daniel M.; "Survey of Software Engineering Tools"; TM-WD-7852/000/00; System Development Corporation; Santa Monica, California; 25 October 1977.

[86] Ver Steeg, R. L.; "TALK - A High-Level Source Language Debugging Technique with Real-Time Data Extraction"; Communications of the ACM; Vol. 7, No. 7; July 1964; 418-419.

[87] Walker, Allan Warren; "An Interactive Graphical Debugging System"; Naval Postgraduate School; Monterey, California; June 1971.

[88] Wheatley, Richard; "A Simulation Technique In Microprogram Validation"; AIDD/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-March 1977; 125-129.

[89] White, William, J.; "Preparation of a Command, Control, and Communication Software Series"; AIAA/NASA/IEEE/ACM Computers In Aerospace Conference; Los Angeles, California; October-November; 1977; 189-196.

[90] Wolman, B.; Debugging PL/I Programs in the Multics Environment, Proceedings AFIPS FJCC; 1978.

[91] Wulf, W. A.; London, R. L.; and Shaw, M.; "Abstraction and Verification in Alphard"; Information Sciences Institute ISI/RR-76-46; June 14, 1976.

[92] Yourdon, Edward; "A Brief Look at Structured Programming and Top-Down Design"; Modern Data; June 1974; 30-35.

[93] Yourdon, Edward; "A Debugging Environment for Real-Time Systems"; Real-Time Systems Design; Paragon Press; Somerville, Massachusetts; 1967; 137-151.

[94] Zelkowitz, M.V.; "Reversible Execution"; Communications of the ACM; September 1973; 566.

[95] Zimmerman, Luther L.; "On-Line Program Debugging - A Graphical Approach"; Computers and Automation; November, 1967; 30-34.

[96] Zurkow, J. L.; "Hardware Helps in Tracing Microprocessor Program"; Electronics; Vol. 49, No. 13; 24 June 1976; 105-107.

[97] Air Force Regulation 800-14, Volume I: Management of Computer Resources in Systems; 12 September 1975; and Volume II: Acquisition and Support Procedures for Computer Resources in Systems; dated 26 September 1975.

[98] "BMDATC Software Development System - Program Overview" BMD Advanced Technology Center; July 1975.

[99] "Computer Program Development Library (CPDL) Catalog, Part 1 - CPDL Charter and User's Manual"; TM-(L)-5319/001/00; System Development Corporation; Santa Monica, California; 30 September 1974.

[100] DI-M-3410: Users Manual (Computer Programs) dated 1 November 1971.

[101] DI-M-3411: Programming Manual dated 1 November 1971.

[102] DI-T-3703: Category I Test Plan/Procedures (Computer Programs), dated 1 November 1971.

[103] DI-T-3717: Test Report (Computer Programs) dated 1 November 1971.

[104] "FORTRAN Extended Debug User's Guide"; Control Data Corporation; 1972.

[105] IBM, "OS PL/I Checkout Compiler: Programmer's Guide; IBM Program Product SC33-0007-3; October 1976.

[106] "Microcomputer Programming"; Mini-Micro Systems; November-December 1977; 79.

[107] MIL-STD-480: Military Standard; Configuration Control-Engineering Changes, Deviations and Waivers; dated 30 October 1968.

[108] MIL-STD-483: Military Standard, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs; dated 31 December 1970.

[109] MIL-STD-490: Military Standard, Specification Practices; dated 30 October 1968.

[110] MIL-STD-1521A: Military Standard, Technical Reviews and Audits for Systems, Equipment and Computer Programs.

[111] "PDP-11 Software Handbook"; Digital Equipment Corporation; 1975.

[112] "Programming Standards and Conventions for Block 3 Software"; Aerospace Corporation; 1975.

[113] SAMSOP 800-14 Attachment 2; "Computer Program Validation/Verification"; dated 15 February 1975.

[114] SECNAVINST 3650.1; Department of the Navy; Tactical Digital Systems Documentation Standards; 8 August 1974.

[115] "Software Development Standards and Conventions for the Telemetry Integrated Processing System"; TM-VN-(L)-0040/000/00; System Development Corporation; Santa Monica, California; 28 June 1977.

[116] "System IIB User's Manual Model 15.2"; TM-(L)-5048/714/00; 3 March 1977.

[117] "Verification and Validation for Terminal Defense Program Software"; Report No. HR-74012; Logicon, Incorporated; San Pedro, California; 31 May 1974.

[118] "1230 Computer Operators Guide - Model 14.2"; TM-(L)-5536/702/00B; System Development Corporation; Santa Monica, California; 29 August 1977.

[119] "3800 Programming Standards and Conventions"; TM-(L)-5567/000/00; System Development Corporation; Santa Monica, California; 16 July 1976.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.