THE BELL SYSTEM Technical Journal

SPECIAL SUPPLEMENT

SAFEGUARD DATA-PROCESSING SYSTEM

THE BELL SYSTEM TECHNICAL JOURNAL

THE BELL SYSTEM TECHNICAL JOURNAL is published ten times a year by the American Telephone and Telegraph Company, J. D. deButts, Chairman and Chief Executive Officer, R. D. Lilley, President, C. L. Brown, Executive Vice President and Chief Financial Officer, F. A. Hutson, Jr., Secretary. Checks for subscriptions should be made payable to American Telephone and Telegraph Company and should be addressed to the Treasury Department, Room 1038, 195 Broadway, New York, N. Y. 10007. Subscriptions $15.00 per year; single copies $1.75 each. Foreign postage $1.00 per year; 15 cents per copy. Printed in U.S.A.

THE BELL SYSTEM TECHNICAL JOURNAL

DEVOTED TO THE SCIENTIFIC AND ENGINEERING
ASPECTS OF ELECTRICAL COMMUNICATION

SPECIAL SUPPLEMENT

SAFEGUARD Data-Processing System

Preface

The papers in this special supplement to The Bell System Technical Journal differ markedly from the more quantitative typical B.S.T.J. papers, which are characterized by their analytical and experimental approach, usually with a definitive telecommunications tie-in. The reason for this contrast is that these papers, taken together, are intended to serve a quite special purpose.

In its defense work for the U.S. government for the past several years, the team of Bell Laboratories and Western Electric, with close support from many contracting firms, has carried out the development of what is believed to be the most complex real-time software/hardware system ever successfully undertaken. These papers constitute an integrated story of the scope of the *software* task, the way it was organized and managed, and the principal lessons learned (problems encountered as well as successes achieved).

We are making this story available in the hope that the computer/data-processing community and others might profit from those developmental and administrative approaches that proved to be particularly effective and avoid those avenues that were found to contain pitfalls.

THE B.S.T.J. EDITORIAL COMMITTEE

SAFEGUARD Data-Processing System:

Foreword

The U.S. government needs and obtains a wide range of services from the nation's businesses. From the Bell System, these services range from the large amounts of ordinary telephone service required to carry on its day-to-day operations to the development of complex systems designed to ensure the nation's defense. With respect to the latter, Bell System policy is summarized in a remark by H. I. Romnes at a stockholders' meeting on April 15, 1970:

> "The Bell System engages in military work as a responsibility we owe our country. We make available some of the communications expertise of the Bell Telephone Laboratories and the Western Electric Company to carry out programs for which responsible agencies of the government have a defined need. We did not seek out military work nor do we seek to expand the amount we have."

The largest system development ever carried out for the Department of Defense by the Bell System started with some exploratory research and development work in 1957 and culminated with the completion of installation and testing of the SAFEGUARD Ballistic Missile Defense System in early 1975. Western Electric was the prime contractor for the SAFEGUARD system and Bell Laboratories was responsible for the design. Major subcontractors were Raytheon and General Electric for the radars, Martin Marietta and McDonnell Douglas for the missiles, and Univac and IBM for the data-processing system.

SAFEGUARD may be the most complex system ever produced by a single, integrated, research and development project and the system would take many volumes to describe. The overall design required the solution of many complex technical problems, and the major subsystems—the two radars, the two missiles, data processing, command and control, and communications—are lengthy stories in themselves. However, the data-processing subsystem development probably has the greatest relevance to the Bell System. This is so because more and more systems are organized around a stored-program, general-purpose computer, controlling system operation on a real-time basis. SAFEGUARD is an extraordinarily large system of this type. It provides a sort of upper bound for the other developments in many ways. For that reason, this supplement to The Bell System Technical Journal consists of papers that describe the major issues arising in the development of the data-processing subsystem, with emphasis on the software. The material included is limited to that which is felt to be useful to the general computing community, *and is an attempt to describe the lessons learned rather than just the successes.* As a result, other system developers may be helped in identifying some management techniques and technical approaches to avoid as well as those that might be useful to them.

To restrict this supplement to a manageable size, the level of detail had to be restricted. The papers are highly interdependent and are intended to be read as a group. Although many details of the design and development are not treated here, the volume as a whole provides a comprehensive summary of the pragmatic approach required for a highly schedule-sensitive project.

The volume begins with an introduction and overview paper. This paper provides important background material for all the other papers, including not only the general organization of the data-processing system but also the role of the data processor in the overall system and a brief history of the ABM system.

The remaining papers are organized into six sections, each covering a major facet of the effort. The Systems Engineering section consists of one paper that discusses the generation and control of requirements. Fundamental control of the entire software development was achieved through the Data Processing System Performance Requirements discussed in this paper.

The Hardware section contains papers describing the data-processing system architecture, emphasizing the modular nature of the system and the maintenance and diagnostic techniques that were important parts of the strategy for obtaining high availability.

The Real-Time Software Development section contains the description of those aspects of the design that depend most critically on the real-time nature of the application and the multiprocessor computer. The successful use of a pool of identical processors to provide the total required processing capacity was one of the major features of the project, and these papers summarize the impact of this system characteristic on the design of the operating system and the overall structure of the software. In particular, the techniques used to structure the software to make the most efficient use of all processors are described in the paper entitled "Process Design: The Structure of Real-Time

Software Systems." Other papers in this section describe the facilities and techniques used to test and debug the system.

The Support Systems section discusses those facilities that were of major importance in supporting the development of the real-time software. The overview paper which introduces this section provides a critical examination of some key decisions in establishing the support environment, which is necessary to every software development. As a result, this paper, and the other papers in this section, should be particularly relevant to other such efforts.

The Development Tools and Techniques section contains two papers that describe special techniques that were used to improve programming efficiency. Although it was not possible to gather enough data to establish unequivocal efficiency improvements, the results are interesting enough to warrant consideration on other projects.

The final section, Project Control, describes some of the more important techniques used throughout the project to monitor progress and maintain control. Although no panaceas were found for any of the well-known problems of controlling software developments, the successful completion of the project demonstrates that adequate techniques are available. Since industry-wide experience indicates that many large software developments in the past have had as much trouble with general project control as with the technical aspects of design, the discussion of the variety of project control techniques used and their effectiveness is believed to be important.

It is impossible in a brief description of a large system development to find any adequate way to acknowledge the contributions of everyone involved. In addition to the major subcontractors listed earlier, important contributions were made by a large number of other organizations. Although all the authors were major participants in the activities which they have documented, many other individuals made contributions equally important. Each of the over two thousand people involved during the course of the project made real contributions toward its success, and it is not possible to acknowledge individually here the very large number of these who provided the key technical and managerial innovations that were vital to that success.

THOMAS H. CROWLEY
Executive Director
Safeguard Design Division

N. H. Brown,
M. P. Fabisch, and
C. J. Rifenberg

Introduction and Overview

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System: Introduction and Overview

By N. H. BROWN, M. P. FABISCH, and C. J. RIFENBERG

(Manuscript received January 3, 1975)

This paper provides the background information necessary for understanding the other papers in this volume, and serves as an introduction to them. It provides a brief history of SAFEGUARD, discusses the hardware and the software involved, and then focuses on the technical and managerial approaches to producing the software.

I. INTRODUCTION

SAFEGUARD is an antiballistic missile (ABM) system primarily designed to respond to attacks by intercontinental ballistic missiles. It is composed of three major subsystems: missiles, radars, and data processing and control. Incoming missiles, after being detected and tracked by the radars, are intercepted and destroyed by defensive missiles. The radars and defensive missiles are controlled by the data-processing system.

Development of the large, real-time data-processing system for the SAFEGUARD Ballistic Missile Defense System was a significant undertaking from any point of view. Developing a system with unique processing and availability requirements led to the involvement of thousands of people and a very substantial commitment of resources. The resulting multiprocessor data-processing system entailed the development of new and sophisticated algorithms, the design of unique testing programs, and the extensive employment of simulations.

These SAFEGUARD papers primarily emphasize the techniques and methods of a software development effort that produced millions of lines of code. Although the classified nature of the project precludes description of a few of the innovations in both software and hardware, most of the important problems encountered involved no security questions and the objective of these papers is to serve the data-processing community by imparting some of the lessons that were learned.

II. OVERVIEW

2.1 Historical context

At Bell Laboratories, research and development on the first anti-ballistic missile (ABM) system, the NIKE-ZEUS, began in 1957. The data-processing hardware requirements for NIKE-ZEUS were met by the development of special-purpose digital computers, an outgrowth of the use of analog computers in previous air defense systems. NIKE-ZEUS field test sites were established in New Mexico, California, and the Pacific. Applications programs and techniques were developed for using digital computers as controllers for tracking and missile guidance, for trajectory estimation and discrimination, and as planning and resource allocators in battle management. These application programs were installed and tested at the field sites during the late 1950s and early 1960s. In 1962, an historic intercept was achieved when a NIKE-ZEUS missile launched from Kwajalein Atoll in the Pacific successfully intercepted a TITAN ICBM launched from Vandenberg Air Force Base.

With the termination of the NIKE-ZEUS project in 1963, NIKE-X system development began. This system required a highly reliable data-processing system (DPS) that could support a peak throughput of about 10 million instructions per second and a peak I/O transmission for radar control of about 70 thousand 64-bit words per second. To achieve these requirements, a special-purpose digital computer was designed using integrated circuits and core storage techniques. A field test site for the NIKE-X development was established at Meck Island, part of the Kwajalein Atoll. Testing at this site has had significant impact on the development program.

In 1967, the basic design of the NIKE-X machine was incorporated into the SENTINEL ABM system. Throughput requirements were met by a multiprocessor capable of using as many as ten processors.

Originally, the goal of SENTINEL was the protection of cities from a ballistic missile attack. In 1969, new objectives, including the protection of U. S. MINUTEMAN ICBM bases rather than cities, were announced. This redirection was indicated by a new system name, SAFEGUARD. SENTINEL equipment remained unchanged. The field test site for NIKE-X now became the Meck Prototype System. Its objectives were redefined from those of an R&D program to those of supporting SAFEGUARD design. A detailed test program was established for the Meck system, providing indispensable support for SAFEGUARD in hardware, software, and algorithm development, as well as multiprocessor operation and reentry environment characterization.

The entire software development of SAFEGUARD has been directed at the specific needs of a real-time, high-throughput, very reliable computing system. The applications programs, operating system, support software, and data-reduction facilities were all designed to meet these objectives.

2.2 System description

There are three types of sites in the SAFEGUARD system: Perimeter Acquisition Radar (PAR), Missile Direction Center (MDC), and one Ballistic Missile Defense Center (BMDC). Figure 1 provides a functional overview of these sites. Although several PAR and MDC sites were planned, only one of each is being deployed. The PAR site utilizes a single-face, phased-array radar to provide early detection and trajectory data on threatening ICBMs. Functions of this site include long-range surveillance, detection, and target selection of threatening objects, and ICBM-threat tracking for SPARTAN intercept. This last capability significantly increases the long-range SPARTAN field of fire. The PAR site does not perform missile guidance. The MDC complex uses the target trajectory and classification data from the PAR along

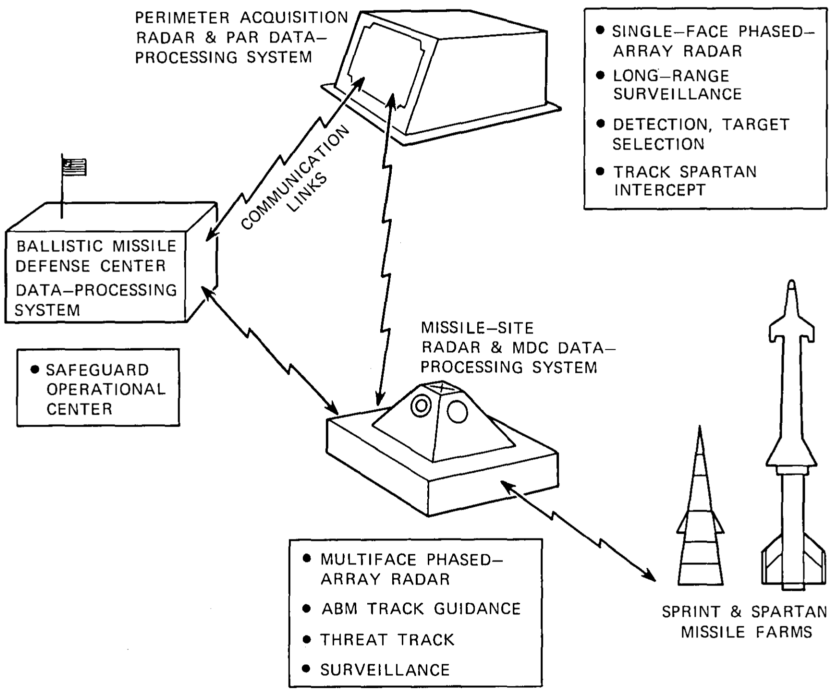

PERIMETER ACQUISITION RADAR & PAR DATA-PROCESSING SYSTEM
• SINGLE-FACE PHASED-ARRAY RADAR
• LONG-RANGE SURVEILLANCE
• DETECTION, TARGET SELECTION
• TRACK SPARTAN INTERCEPT
COMMUNICATION LINKS
BALLISTIC MISSILE DEFENSE CENTER DATA-PROCESSING SYSTEM
• SAFEGUARD OPERATIONAL CENTER
MISSILE-SITE RADAR & MDC DATA-PROCESSING SYSTEM
• MULTIFACE PHASED-ARRAY RADAR
• ABM TRACK GUIDANCE
• THREAT TRACK
• SURVEILLANCE
SPRINT & SPARTAN MISSILE FARMS

Fig. 1—SAFEGUARD system.

with additional data supplied by its multiface phased-array radar. This site provides additional surveillance and target tracking and also performs the functions of track and guidance for the SPRINT and SPARTAN missiles. Both PAR and MDC sites report to the BMDC, a central command center. The BMDC provides a command interface with other military systems and a means of disseminating command directives and controls.

The PAR and MDC radars are controlled by the data-processing systems, collocated with the radars. At the PAR and MDC sites, application programs perform surveillance, tracking, target classification, radar management and testing, intersite communication, and display functions. Additional application programs at the MDC support the battle management and missile guidance functions. The BMDC data-processing system primarily contains display and command/control programs.

Both PAR and MDC radars are controlled by the DPS through the use of digital commands. These commands are used to control beam pointing, frequency selection, receiver gating, thresholding, etc. The SAFEGUARD system design makes use of some constraints on the combinations of radar operations that can be performed and, therefore, on the sequences of pulse transmissions. Appropriate radar commands must be generated by the application programs and sent to the radar at least every few milliseconds. The radar pulse patterns used in SAFEGUARD provide a framework for the time design of the real-time application programs.

2.3 DPS requirements

The data-processing system design was dominated by requirements for high throughput and stringent availability/reliability constraints; i.e., requirements supporting a high probability that the system would be available when required for a mission and highly reliable during the mission.

The fact that the radar is controlled by the DPS contributed significantly to both input/output (I/O) and processing requirements. Appropriate radar commands must be generated by the application programs and output to the radar at least every few milliseconds, yet the DPS must be able to complete processing between two radar events. This contributes to estimates of a peak-load throughput of 10 million instructions per second.

Input/output requirements were further increased by a variety of special-purpose peripherals such as missile controllers and data-transmission controllers for intersite data transmission. The DPS was also required to communicate simultaneously with computing peripherals, especially disc and tape, as well as to provide status information to, and receive commands from, system-control personnel.

The nature of the application imposed requirements for high availability; therefore, a maintenance system was required for fast recovery and quick fault isolation and repair in the event of a hardware malfunction.

Size and complexity increased the problem of verifying the system. This imposed a requirement for a system exerciser that could be used to verify as much of the system as practical.

2.4 Tactical site configuration

This section describes in detail four aspects of a site DPS configuration: hardware, software structure, maintenance and diagnostic subsystem, and exercise subsystem. Except for the absence of an exercise subsystem at BMDC, the DPS structure is similar for MDC, PAR, and BMDC.

2.4.1 DPS hardware

Figure 2 shows the equipment at the MDC site consisting of a central computer and associated peripherals. The central logic and control (CLC) is the multiprocessor computer used to drive each DPS. Under software control, the CLC can be configured into two separate partitions of arbitrary size, each capable of operating as an independent com-

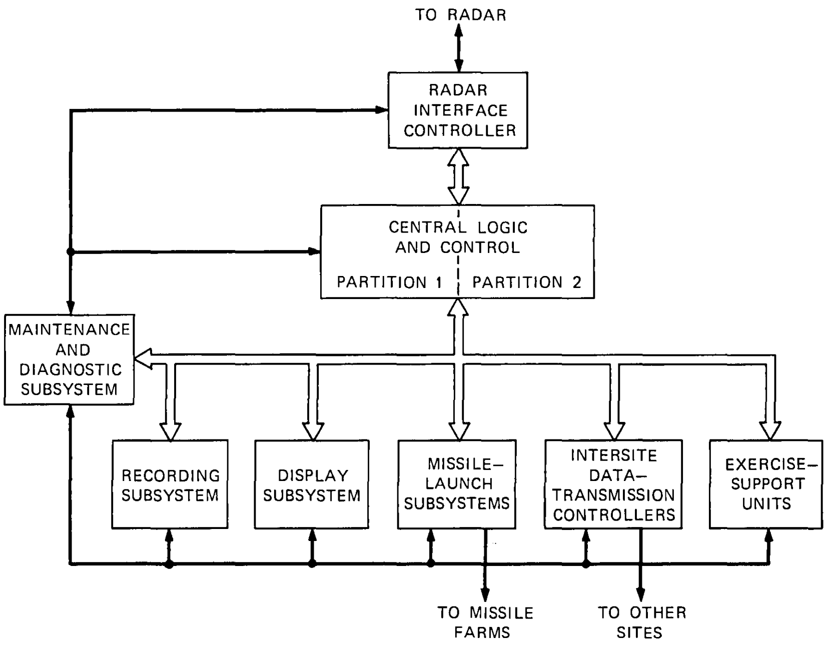

TO RADAR
RADAR INTERFACE CONTROLLER
CENTRAL LOGIC AND CONTROL
PARTITION 1
PARTITION 2
MAINTENANCE AND DIAGNOSTIC SUBSYSTEM
RECORDING SUBSYSTEM
DISPLAY SUBSYSTEM
MISSILE-LAUNCH SUBSYSTEMS
INTERSITE DATA-TRANSMISSION CONTROLLERS
EXERCISE-SUPPORT UNITS
TO MISSILE FARMS
TO OTHER SITES

Fig. 2—SAFEGUARD data-processing system equipment.

puting system. Application software executes on the larger partition. Exercise drivers (described below) for the application software and support activity execute on the smaller partition, which also provides a pool of spare equipment.

The CLC can be configured with up to ten processors. Single-processor throughput of about 1.5 million instructions per second is achieved by a combination of design techniques including instruction execution overlap and use of high-speed arithmetic algorithms. Instruction overlap is achieved by utilization of three asynchronous control units for instruction fetch, operand fetch, and arithmetic execution. Every processor has access to each of several read-only instruction memories called program stores, and several read/write memories called variable stores. These stores have a memory cycle time of 500 ns and a double word size of 64 bits to provide a memory bandwidth in excess of that required for maximum performance of a single processor.

The input/output controller (IOC) controls the transfer of data between the CLC and its peripherals. Since processors do not communicate directly with peripherals, processing and I/O can occur simultaneously. The IOC provides full-duplex operation on 16 channels. Priority circuitry within the controller allows time-multiplexed operation of the channels. The IOC executes commands from IOC programs resident in variable store. Both processors and peripheral devices can initiate IOC program execution.

A timing generator provides a real-time clock and a programmable mechanism for initiating activities at specified times. It can cause the initiation of an IOC program when a specified time of day has been reached. A status unit provides a means of monitoring, in real time, the status of any DPS unit. It also serves as a central point for the distribution of control over the DPS.

CLC peripherals are divided into several subsystems. The Maintenance and Diagnostic Subsystem and the Exercise Subsystem will be described later.

The radar interface controller is the primary interface between the radar and the I/O controller of the CLC. Control and data words are exchanged between these two units. The radar control computer accepts formatted binary words from the CLC and distributes data to the appropriate radar subsystem where a digital-to-analog conversion takes place.

The recording subsystem contains the standard computer peripheral devices: magnetic tape transports, disc memory units, line printers, and card reader.

A man-machine interface is provided through the display subsystem which includes cathode-ray-tube displays with light pens, wall displays, and teletypewriters.

Digital data are transferred between sites by means of the intersite data transmission controllers.

The missile launch subsystems convert CLC commands into control signals for the collocated and remote missile farms and receive missile status conditions, encode them, and send them to the CLC.

2.4.2 DPS software structure

The collection of application software used to drive the DPS is called the application process. The application process is built from basic computing units called tasks, which are single routines with or without subroutines. The operating system, considered to be part of the process, schedules tasks from a predetermined, priority-ordered task list for execution on the next available processor. Once in execution, a task is not interrupted before completion except for error conditions.

A bit string associated with each task on the priority-ordered task list indicates completion of predecessor condition(s) prior to task execution. The operating system enables execution of the highest-priority task with all predecessor condition bits set. Thus, an important part of process design is development of the priority-ordered task list and the predecessor conditions for each task. The predecessor conditions fall into three main types:

(*i*) Time—Functionally, the programmable feature of the timing generator is utilized in setting predecessor condition bits.

(*ii*) I/O completion–Input/output may be initiated by a processor or by a peripheral device. In either case, a task does not "hold" a processor while waiting for I/O completion. Instead, upon I/O completion, a predecessor condition bit is set for a designated task.

(*iii*) Other task completion—Long-running computations are often subdivided into several shorter ones. Appropriate sequential computational requirements are preserved by designating other task predecessor conditions.

Where possible, the application process is asynchronous, i.e., tasks are only enabled when data are available to be processed.

2.4.3 Maintenance and diagnostic subsystem (M&DSS)

The M&DSS is composed of test equipment and software that supports digital equipment maintenance. The M&DSS verifies the availability and readiness of DPS hardware by conducting nonreal-time, programmed, diagnostic tests on equipment through an independent data bus connected to each digital unit. These special M&D data paths are also used to support other objectives of the M&DSS which include initializing DPS hardware and, in the event of a malfunction, automatically supporting DPS recovery operations. The M&DSS also provides a centralized control point for status monitoring, equipment allocation, and manual interface with DPS software.

The M&DSS has two distinct facilities for running diagnostics. The primary one involves the M&D processor group, which uses a modified CDC Model 1700 computer system to provide fully automatic, high-speed execution of test programs with automatic interpretation of results through use of fault-location dictionaries. The other facility involves the M&D console group, which uses a cathode-ray-tube display console for manual execution of diagnostics and interpretation of results. Each facility is linked to all the digital racks in the DPS and to certain digital racks in the radar areas. These data paths provide the means by which M&DSS software can access each unit as required for DPS initialization, recovery, and diagnostic operations.

2.4.4 System exerciser

A system exerciser was designed for PAR and MDC sites. It provides support for development and integration of the applications processes, evaluation studies that include fidelity validation of various simulators, and site readiness verification of both local and multisite system configurations.

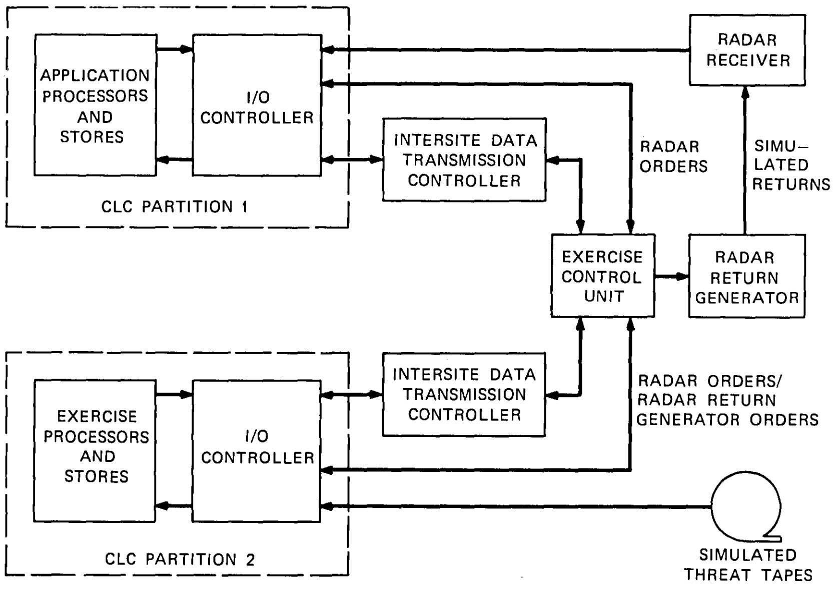

APPLICATION PROCESSORS AND STORES
I/O CONTROLLER
CLC PARTITION 1
RADAR RECEIVER
INTERSITE DATA TRANSMISSION CONTROLLER
RADAR ORDERS
SIMU-LATED RETURNS
EXERCISE CONTROL UNIT
RADAR RETURN GENERATOR
EXERCISE PROCESSORS AND STORES
I/O CONTROLLER
INTERSITE DATA TRANSMISSION CONTROLLER
RADAR ORDERS/ RADAR RETURN GENERATOR ORDERS
CLC PARTITION 2
SIMULATED THREAT TAPES

Fig. 3—Functional representation of the hardware configuration for the PAR system exerciser.

Software was developed to run on the exercise partition of the CLC to generate simulated radar returns and simulated intersite communication. Special hardware was developed to inject the simulated threat data at the receiver of the radar. This allows testing a significant portion of the radar and drives the data processor with realistic data at its actual interface with the radar. Figure 3 provides a functional representation of a PAR exercise configuration.

The principal communication between the two partitions is through the exercise control unit (ECU). The ECU intercepts application program orders to the radar, and intersite messages, and directs them to the exercise partition. The ECU routes simulated radar returns generated by exercise software to the radar-return generator for conversion to analog waveforms and injection into the receiver of the radar.

The exercise software is a real-time process similar in construction to the application process. An off-line facility is used to simulate a threat and generate tapes with a time sequence of the manner in which the threat appears in the radar viewing volume. These tapes are used by the exercise process in generating replies to application-process radar transmissions.

2.5 Software development

2.5.1 Tactical Software Control Site

To develop the large number of programs required for the deployed system and its support, a Tactical Software Control Site (TSCS) was established at Madison, New Jersey. The software development organization, consisting of designers, programmers, test teams, and many others, was located at a few distinct facilities in northern New Jersey, all within a few miles of each other, and a single North Carolina location.

A test bed was required to reproduce accurately the software environment existing at site such that performance of software in its operational environment could be verified; software testing could be accomplished in close proximity to the design organization; and testing could precede site availability to reduce development time. To reproduce the site software environment, the test bed was required to have a representative complement of computing hardware for the PAR and MDC; replicate the interfaces between the computer and peripherals; replicate the peripheral devices to the extent that device performance and characteristics were not completely isolated from the computer; and provide the capability for actually netting the PAR and MDC processes for purpose of system testing. Thus, a test bed was established at TSCS and contained separate PAR and MDC configurations corresponding to the PAR and MDC sites. The configurations provided peripheral hardware needed by software, but did not include all of the analog portions of the radar or missile interfaces. Communication paths between PAR and MDC test-bed configurations were included via the data-transmission controllers. This permitted TSCS netted testing in advance of system testing at the sites.

Experience from previous development projects indicated that all available test-bed time would be required for system testing, operating-system development, and hardware installation and maintenance. Support functions (e.g., software preparation and analysis) were therefore designed for operation on general-purpose computers such as the IBM System/370 and HIS 635. These machines were then also required at TSCS.

2.5.2 Software development cycle

The software development cycle for SAFEGUARD was not substantially different from that of other large systems. In practice, individual phases of the development cycle overlapped since the general approach followed was integration of a basic working system with increasingly more complex capabilities. The separate phases of the development cycle consisted first of the *requirements-generation* phase, in which system requirements were determined, established, negotiated, documented, and rigorously controlled. The *design* phase consisted of process design and program design. In process design, the system requirements were translated into a software architecture which defined global data structures, tasks, task priorities, and task-timing requirements for the data-processing environment. In program design, the local data base, algorithms, and control structure for the individual tasks were determined. In the *coding and unit-testing* phases, code was written, compiled, and checked at the unit or task level, using a simulator, drivers, and standard debugging techniques. Next, at the test bed, separate *process-integration* teams combined blocks of new, debugged unit code into processes for increasing functional capabilities. When the tactical software achieved a predefined level of capability, it was sent to site for *site integration.*

Activities at site were similar to those at the TSCS. However, at site the entire complement of peripheral hardware was available for integration with the system. Moreover, it was at site that formal acceptance tests were run. The final phase of system development was *system integration,* in which the PAR, MDC, and BMDC sites were "netted" and the coordinated operation of the entire system was achieved. During all phases of system development, *evaluation* played a strong role. A separate organization was responsible for evaluating system requirements, implementation algorithms, and system-test results. Feedback resulted in frequent changes and refinements in many areas.

Following is an expanded overview of some important features of the SAFEGUARD software-development cycle.

2.5.3 Requirements

The Data Processing System Performance Requirements (DPSPRs) are a set of documents that define the requirements of SAFEGUARD tactical programs for the PAR, MDC, BMDC, and system exerciser processes. Requirements were generated by the system engineering organization in accordance with overall system objectives, which were defined by the Department of Defense. Changes to the requirements were made as a result of detailed software design by the development organization, Meck prototype system-test-program data, system-evaluation efforts, and detailed review with the U.S. Army SAFEGUARD System Command (USASAFSCOM).

The DPSPRs met their original objectives of providing a clear definition of the computing requirements. They have continued to be the up-to-date system definition of SAFEGUARD performance, and have been used to specify all system-testing and acceptance requirements.

2.5.4 Design

Process design was the definition of overall software structure including task assignment and global-data-base design. The objective of process design is to meet system requirements with the minimum-cost DPS configuration. This activity was complemented by program design which involved developing the algorithms, internal data base, and control structure necessary to implement the function defined for a task. This activity led to a detailed software specification, including specific mathematical equations or decision tables.

Decisions were made in both process and program design to support early development of a system to which greater capability would be gradually added. Emphasis was placed on modularity in design to ease system growth.

It was found to be essential to initiate the design of the data recording and reduction system early in the development cycle. An attempt was made to define data to be recorded for each computing function, and to design the data base to include consideration of recording and the subsequent analysis to be carried out upon the recorded data.

In many areas simulations were used to validate the design. In some cases, a few selected equations were implemented on a time-sharing system for a quick exploration of correctness and adequacy. In others, a subset of the real-time computer program, complete with its interface structures, was simulated.

The size of individual programs and the time required for their execution were two major parameters which were controlled. Initial sizing and timing estimates were made early in the development based on past experience with similar programs. Throughout the course of further development, sizing and timing estimates were tracked on a monthly basis.

Design reviews were held frequently and proved to be an effective means for communicating problems and solutions relating to planning or design issues to other members of the project. These were attended by a review board consisting of both designers and project managers.

2.5.5 Coding and unit testing

All of the software preparation and most unit testing was performed using commercial computers. This was primarily because test-bed time was too valuable to be consumed for compiling and unit testing.

Most SAFEGUARD software was written in CENTRAN, an extensible intermediate-level language resembling a subset of PL/1. CENTRAN generated efficient code. It provided many of the advantages of high-level languages, but could be interspersed with assembly language and system macros when necessary. It was adopted as the project standard.

To facilitate program preparation and unit testing, a linkage editor, a CLC simulator, and a disc library system were also developed for execution on the IBM System/370. The linkage editor bound units of CENTRAN object code for execution on the CLC or CLC simulator. The library system functioned as an editor and disc-file manager, which helped control CENTRAN source and object code. The linkage editor and simulator were developed on the SAFEGUARD project, while the library system was a SAFEGUARD modification of an existing IBM proprietary program.

2.5.6 Process integration

Following unit debugging, collections of units were tested for increasingly greater functional capabilities on the PAR or MDC test beds by independent integration teams. Frequently, large drivers were developed to assist in early functional testing. Subsequently, the system exerciser was used to stress and drive the application process to various conditions and loads.

Detailed analyses of integration tests were possible because the application and exercise processes contain real-time recording functions which were designed as an integral part of the software. Recorded data were reduced and analyzed primarily off line on the IBM System/370 using the SAFEGUARD Date Reduction System, although summary information was available on line.

A hardware/software CLC performance monitor was developed and installed at the TSCS. It was used primarily to validate that the process performance was consistent with its design. Troubles, such as heavily loaded time frames and long-running tasks, were analyzed. When possible, design changes were made to provide a more balanced system.

2.5.7 Site and system integration

When the application and exercise processes achieved predefined capabilities, they were sent to site for further integration. Capabilities already established at TSCS were reverified in the expanded hardware environment. Further testing concurrent with and complementary to test-bed integration was conducted, with primary emphasis on full process testing using the system exerciser. A comprehensive series of acceptance tests was run to demonstrate that system capability was consistent with requirements. Tests ranged from satellite tracking and identification to system exercises which drove the system to design traffic levels.

During system integration, which is the final level of product testing prior to delivery to the customer, it was not possible to exhaustively test all tactical threat environments. An "Endpoint Test" was defined at the design traffic level for each of the various system-operating modes. A series of tests was designed for each mode, at first simulating all communications with other sites, then netting pairs of sites, and finally netting the system.

The stress level was reduced in early testing by selecting subsets of the Endpoint Test environments and by running buildup tests at these lower stress levels before operating the netted system at design traffic levels. The use of a common environment for a number of tests, with traffic buildup by addition to this environment, and buildup of physically internetted sites in stages, led to the "test-chain" approach to testing. This approach, in which all tests in the chain support the Endpoint Test, greatly simplified the problems of integrating a dynamic system.

Commercial computers were installed at site during the site-and-system integration period for data-reduction support. This support was required on location to provide prompt analysis of data recorded during testing. Tight schedules and lack of available CLC time required that this facility be provided by a support computer.

2.5.8 Evaluation

System evaluation was primarily an analytical activity which, because of the complexity of the SAFEGUARD system, relied heavily on simulation. A SAFEGUARD system simulation was designed to provide insight into overall system operation with particular emphasis on battle-planning functions. Initially, the simulated system was made to operate in accordance with performance requirements. Since, quite properly, performance requirements often permit the designer considerable latitude, modeling of the system in this initial phase often entailed considerable invention. The goal was to ensure that objectives would be achieved if the system operated in accordance with performance requirements and that inadequacies in system design would be identified and corrected before resources were wasted attempting to implement a faulty design. Since there was a practical limit to the level of detail in which the various weapon system functions could be modeled, more detailed simulations of the particularly critical functions of surveillance, tracking, target selection, and guidance were added. By employing these simulations in concert, considerable insight was gained into detailed system operation.

As the design of the tactical hardware and software stabilized, these simulations were continually updated to provide a more accurate representation of tactical operation, and a continuous evaluation of the evolving system. Early development of detailed but evolving simulations permitted in-depth analysis of most critical areas of SAFEGUARD operation. A number of significant design modifications can be attributed directly to evaluation activity. A noteworthy example is the restructuring of both the PAR and MDC overload-response software to provide improved performance in a high-traffic environment.

Systematic and detailed analysis of the Meck prototype-system tests, which were designed to stress critical functional capability, provided confidence in the validity of analyses based on simulation. Finally, simulation, in addition to providing a tool for evaluation of overall system performance, permitted the definition of explicit thresholds for use in acceptance tests of the entire netted system.

2.6 Project organization and control

2.6.1 Organization

Organizations were established for each of the major software efforts, PAR, MDC, BMDC, and System Exerciser. A separate systems-engineering organization was responsible for requirements and evaluation. Support-software development organizations were also established for each major support activity such as DPS maintenance software, real-time support software, nonreal-time support software, and computer operations. Each major activity was directed by a project manager.

The software development organization consisted of engineers and programmers primarily from Bell Laboratories, IBM, and Western Electric. While project responsibility rested with Bell Laboratories, IBM was responsible for much of the software development. These development activities were directed by IBM managers who were in turn responsible to Bell Laboratories project managers for completion of the tasks. For the most part, Western Electric engineers and programmers were integrated directly into Bell Laboratories organizations, with the notable exception of test-bed-facilities management, which was turned over to Western Electric early in the development cycle.

2.6.2 Control

Overall scheduling for the project was the responsibility of the system-engineering organization. Project managers were held responsible for coordinating and setting schedules for software under their control, consistent with overall schedules.

Schedules were documented at several levels of detail in a management-information system. Visibility was provided by frequent design/schedule reviews, and by a Principal Event Report. The principal events were selected major milepost achievements in performance, and were scheduled within the total network of activities related to software and system development. A written report as to the performance achieved relative to the defined requirements for a principal event was required within 72 hours of the schedule date. All open items were reported with a schedule for their completion. Upon completion of an open event, written confirmation to management was required.

Further development control and discipline were achieved by the use of additional techniques. A Policies, Procedures, and Standards (PPS) Manual was established and maintained. The manual provided detailed policies and standards to ensure uniformity and control within the project. PPSs were written on change management, documentation, management reporting, programming standards, etc. Software change management standards were established early, and they were extended, modified, and adapted for use on each major activity. Typically, this included documenting troubles on standard Trouble Report forms and keeping track of them and their solutions in a Status Accounting System. Stable software was "frozen," stored, and officially released by a central organization.

Because of the difficulty of employing subcontractors on a large complex software development, very careful attention was given to defining interfaces and a detailed task description, monitoring, and evaluation system was devised. This system was fundamental to the success of the development effort.

Comprehensive documentation standards were also established early. Support software documentation emphasized requirements and user information; tactical software documentation emphasized requirements, design information, test plans, and well-commented listings. In general, documentation and software development were synchronized.

The emphasis on planning was fundamental to the overall management approach. Although no single planning format or technique was prescribed, each project manager was required to plan in detail for the complete design, implementation, and testing of his part of the system.

2.6.3 Resource requirements

Resource estimation and control were generally the responsibility of project managers. Normal budgetary procedures were used, requiring justification to and approval by upper management and the customer on a yearly basis. Manpower needs were estimated by project managers using experience and algorithms from other large projects together with a detailed plan of the work to be performed. Manpower restrictions were resolved by replanning and modifying schedules.

Support-computer needs were estimated by project managers and analyzed by the support-computer project manager, who coordinated the acquisition of support equipment. Application-computer require-

Table I — SAFEGUARD software development—quantities of instructions and statements

Real-Time Software Instructions	
CLC operating system	100,000
MDC applications	300,000
MDC exerciser	50,000
PAR applications	200,000
PAR exerciser	25,000
BMDC applications	60,000
Total	735,000
Support Software Source Statements	
CLC software preparation support	210,000
System simulation	50,000
Exercise support	30,000
Data reduction	150,000
Configuration management	70,000
Logic simulation	70,000
Total	580,000
Installation and Maintenance Software Instructions	
MDC radar installation	50,000
PAR radar installation	110,000
PAR radar test	60,000
Maintenance & diagnostic	300,000
Diagnostic operating facility	120,000
DPS installation & test	190,000
Total	830,000

ments were established and monitored through periodic sizing estimates by the PAR, MDC, and BMDC project managers.

The size and duration of the SAFEGUARD development effort was large indeed. Table I shows the size of the major components of software: real-time software, consisting of MDC and PAR applications and exercise programs, BMDC applications programs, and the CLC operating system, totalled 735,000 instructions; support software, such as compilers and simulators executed on commercial computers, totalled 580,000 statements, some assembly language, and some PL/1 and FORTRAN; installation and maintenance software for the data-processing system and the radars totalled 830,000 instructions. At least several hundred thousand additional instructions were developed for other purposes, such as test drivers and specialized simulations. The total development interval, starting with the generation of SENTINEL requirements and concluding with SAFEGUARD system integration, spanned 90 months.

III. CONCLUSION

Perhaps the most important lesson to be learned from SAFEGUARD is that a large, well-conceived development project, however ambitious, can be completed successfully. During the development, the number of sites was changed, drastically reducing the size of the deployment. This, coupled with test results, as well as changes in objectives, led to modifications in the overall system design. However, it can reasonably be said that the complete development, including the integration of the first installed sites, was performed on schedule and that the system met the prescribed performance specifications. Although cost performance is a little bit harder to define because of the effects of inflation over the period and because of changes in the deployment, it seems clear that costs were controlled reasonably.

To reiterate an observation made earlier, implementation of the SAFEGUARD data-processing system was a significant undertaking, one of the most complex ever attempted. Its production entailed the development of a highly reliable multiprocessor computer system, and the generation of millions of lines of code. The papers that follow describe some of the design of the system as well as the lessons that were learned and the techniques employed.

Section I
SYSTEMS ENGINEERING

Copyright © 1975 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

The Data-Processing System Performance Requirements in Retrospect

By D. W. MESEKE

(Manuscript received January 3, 1975)

The Data-Processing System Performance Requirements (DPSPRs) specify the required performance to be provided by the SAFEGUARD system software. They were developed primarily by one systems engineering department at Bell Laboratories. Their objective was to specify the required functional performance in sufficient detail to permit software development. The DPSPRs evolved from similar documentation that was developed for systems prior to SAFEGUARD. Their history, development, use, and document control system are described. Suggested improvements are also discussed.

I. INTRODUCTION

The Data-Processing System Performance Requirements (DPSPRs) are a set of documents that specify the required system performance to be provided by the tactical real-time software. A separate set of requirements exists for each site: one for the Missile Direction Center (MDC) site, one for the Perimeter Acquisition Radar (PAR) site, and one for the Ballistic Missile Defense Center (BMDC). The DPSPRs include requirements for such functions as site communications, displays and controls, radar control, interceptor response planning, and missile guidance. Since the SAFEGUARD system must operate continuously in real time with minimum down time, the DPSPRs also include requirements for exercise and fault detection to verify total system performance. The DPSPRs do not include requirements for installation and checkout software, software error control, or process initialization.

The primary objective of the DPSPRs is to specify the required functional performance in sufficient detail to permit the development of the software by the designers, yet not in such detail as to overly limit design freedom. A second objective is to state functionally how the system is to operate in its different defense modes. Thus, the DPSPRs formalize for the customer—the Army SAFEGUARD System Command (SAFSCOM)—the required system functions, their interactions, and the expected system performance.

The DPSPRs contain detailed requirements for each identified system function. They are not part of the high-level contractual documentation, and they do not contain the detail required to subcontract software development. They are written at an intermediate level along functional lines, but they do not dictate the organization of the software. For example, one section, Target Selection, provides requirements for calculating a set of parameters from quantities specified in another section, Track. When the software was designed, it was found more efficient to have the track software calculate the parameters and pass them to the target selection software. Because the DPSPRs did not specify software organization, it was possible to choose the more efficient software implementation.

This paper provides a retrospective view of the DPSPRs, identifying different aspects of their development that either worked well or should have been done differently. The history of the DPSPRs is given first, followed by a description of how they were developed. A short description of how they were used is given next, followed by a section on document control. The conclusion summarizes recommendations that may be useful for the generation of future data-processing performance requirements.

II. HISTORY

Prior to SAFEGUARD, considerable experience had been gained from the design of the NIKE-ZEUS, NIKE-X, and SENTINEL ABM systems. As part of NIKE-X research and development, a series of documents was developed to specify how various system functions would be performed. They described, for example, how the radars were to gather target trajectory data required to launch and guide a missile to intercept a target. These documents were the forerunner of the DPSPRs.

In January 1967, system studies were initiated to determine the feasibility of deploying a thin area-defense system, later defined as SENTINEL, using components (radars, data processors, missiles, etc.) developed for NIKE-X. The first Data Processing System Performance Requirements documents were written for the SENTINEL system.

In April 1969, Bell Laboratories was redirected by the Department of Defense to develop the SAFEGUARD ABM system. The initial issue of the DPSPRs for SAFEGUARD was completed in July 1969 in accordance with this redirection. The time interval was short because many of the DPSPR concepts and functions that had been developed for SENTINEL were applicable to SAFEGUARD and because this first issue contained mainly qualitative requirements; i.e., many parameter values were still to be determined. The purpose of this first issue was to disseminate as much information as soon as possible to the software designers, who had already organized to develop the SENTINEL system. This issue was succeeded by a second, more quantitative issue in May 1970, which was placed under internal document control.

On March 31, 1971, the DPSPRs were submitted to the customer for baselining. Baselining the DPSPRs consisted of a detailed document review and preparation of changes, after which both the customer and Bell Laboratories agreed to accept the documents. This process was completed on May 31, 1972. The baselined DPSPRs were then submitted for formal configuration control procedures under which all changes had to be (and must still be) approved by the SAFEGUARD Local Configuration Control Board.

III. HOW THE DPSPRs WERE DEVELOPED

Development of the DPSPRs was the function of the system design department, which initially consisted of about thirty engineers and programmers. Their first step was to write an "operational concept" paper for SAFEGUARD. The concept paper identified the defense objectives, the command and control configuration, and the general operation of the radars and missiles in their defense roles.

Based on the concept paper, the DPSPRs were organized according to the operational functions required at each site. The organization of a typical DPSPR is shown in Table I. The DPSPRs were arranged so that each section addresses a major system function. The ordering of the sections within a DPSPR was primarily based on the sequence in which the functions must be performed. Each section includes three main subsections: objective, operational description, and requirements.

Table I — MDC DPSPR organization

1.	General
2.	MDC Site Management
3.	Radar Management
4.	Surveillance
5.	Track
6.	Target Selection
7.	SPARTAN Interceptor Response
8.	SPRINT Interceptor Response
9.	SPARTAN Guidance
10.	SPRINT Guidance
11.	Equipment Tests
12.	Exercise Subsystem
13.	System Constraints
14.	Displays and Controls

The operational description includes, in most sections, a general description of how that function is to operate in different system-defense modes. It has been suggested that the DPSPRs should have had one section devoted to a complete operational description of the system rather than appearing throughout the documents. Since the level of detail varies from section to section, this suggested reorganization could probably have provided a more consistent description of the functional operation of the entire system. The concept paper did not provide the detailed descriptive information that was later felt to be lacking on the project.

Original plans called for each DPSPR section to have an inputs/outputs subsection that would define the interfaces among functions. These subsections were never included in the DPSPRs, primarily because there was insufficient time. Since the requirements for each function either specified or implied its inputs/outputs, it was felt that these subsections could be omitted. In retrospect, this probably was a mistake. For instance, an implied output of one function was missed by the software designers in a specific case in which one function was required to stop or inhibit an action previously started by another function. This mission output was not discovered until later during functional integration testing of the designed software. Then, many questions were raised:

(*i*) Is it really necessary to stop the action?
(*ii*) What happens if the action is not stopped?
(*iii*) Can the missing output be implemented without jeopardizing schedules?
(*iv*) How much retesting is required if a modification is made?

Clearly, a perturbation in the software development effort occurred that might have been avoided if the inputs and outputs had been explicitly stated.

The general policy for writing requirements for a function was to state the requirements without descriptions of how the function should be implemented. In many cases, this was difficult to do; it was often easier to say how a function should be done rather than to state a performance requirement for the function. This led to two problems. First, when a requirement specifies how something is to be done, the software designer feels constrained. He may know a better way to implement the requirement or he may want to try other ways. Second, if a performance requirement does not exist for a function and only the technique is specified, then the test designer must generate his own performance requirement, or his tests will check only to see that the proper technique has been implemented. For this reason, it has been suggested to both system designers and software designers that the DPSPRs should have included only system performance requirements with no mention of implementation or algorithms. This is a philosophically "pure" notion which might or might not work. The method of specifying requirements depends largely on the project organization and the talent of the people involved. For example, before writing a DPSPR function, the systems engineer discussed the particular function with his counterpart, the software designer. In some instances, primarily those in which there was a lack of specific experience, the designer requested that complete design details be supplied. In other cases, the designer wanted only an overall performance requirement because he felt he knew how to produce the design. Requirements were written both ways, but experience suggests that it is probably best to state the performance requirement and then provide a recommended technique to be used at the designer's option. In summary, the system designers tried to reason out the level of detail to be included and took the pragmatic approach of "getting the job done" and trying to satisfy both the software designer and the customer.

IV. HOW THE DPSPRs WERE USED

In software design, the DPSPRs were used in three phases: setting up the software structure, establishing the internal organization of each basic function, and functional testing.

In setting up the software structure, the routines and subroutines needed to perform the functions were based on the requirements in the DPSPRs. In software design, primary emphasis was placed on definition of the inputs required to perform the functions and the outputs required by other functions.

Next, the internal organization of the defined routines and subroutines was established. At this time, emphasis was placed on defining both the particular algorithms required within a function and the interfaces between routines.

As the design of the routines approached completion, the DPSPRs were continually consulted to determine if the designs met requirements. DPSPRs were then used to determine the functional testing required for the completed design.

In system evaluation, the DPSPRs were used primarily as a reference document. The first stage of system evaluation was to verify that the DPSPR specifications would meet system objectives. The evaluation program then determined if the implementation met the DPSPR requirements. The system evaluation effort led to development of new system functions, changes to existing ones to provide better performance, and sometimes modification of the requirements themselves.

The DPSPRs were used by the customer as the documents that specified performance of the system they were buying. The customer coordinated with the design engineers in the formulation of all pre-baseline versions of the DPSPRs. After baselining, the customer was deeply involved in the evaluation and discussion of each change proposed for the DPSPRs. In addition, the DPSPRs were used by the customer for his independent evaluation of the system design.

V. DOCUMENT CONTROL

After the first issue of the DPSPRs was published and distributed, an intensive review was held with software designers and system evaluators. This resulted in changes to add new requirements, to expand upon old ones, and to correct errors. No formal accounting of the agreed-upon changes was kept, and some systems and software designers were not made aware of these changes until they received their copies of the next issue. Clearly, there was a need for a better method of keeping track of problems and their solutions and a need for timely revisions.

To solve this, a document control system was established in which all DPSPR-related problems were identified by a Trouble Report (TR) and the solution to each problem was described by a Correction Report (CR). TRs could be written by anyone uncovering a problem, but had to be approved by the writer's immediate supervisor. Once approved, the TR was given a number, recorded in the log book, and sent to the supervisor responsible for the affected DPSPR. After his approval for action, the TRs were assigned to the persons responsible for the particular sections that were related to the problem. Each solution was described in a CR to be approved by the TR originator. So both the TR originator and CR originator had to agree upon the solution. When agreement was reached, the CR had to be approved by the supervisor responsible for the applicable DPSPR.

Since changes to the DPSPR generally implied corresponding changes in the software design, all CRs were reviewed and approved by all affected software design departments, with final approval delegated to higher levels of management as the software delivery date was approached. After final approval, the CR was sent to publications for preparation and distribution of the revision pages for the CR.

Three different methods of achieving this approval were tried before an adequate approval sequence was found. Figure 1 shows a flowchart of each of these methods. First (Method 1), after the DPSPR coordinator approved the CR, a copy was sent to each affected software design supervisor for an assessment of the software impact of the change in terms of cost and schedules. When all assessments were received by

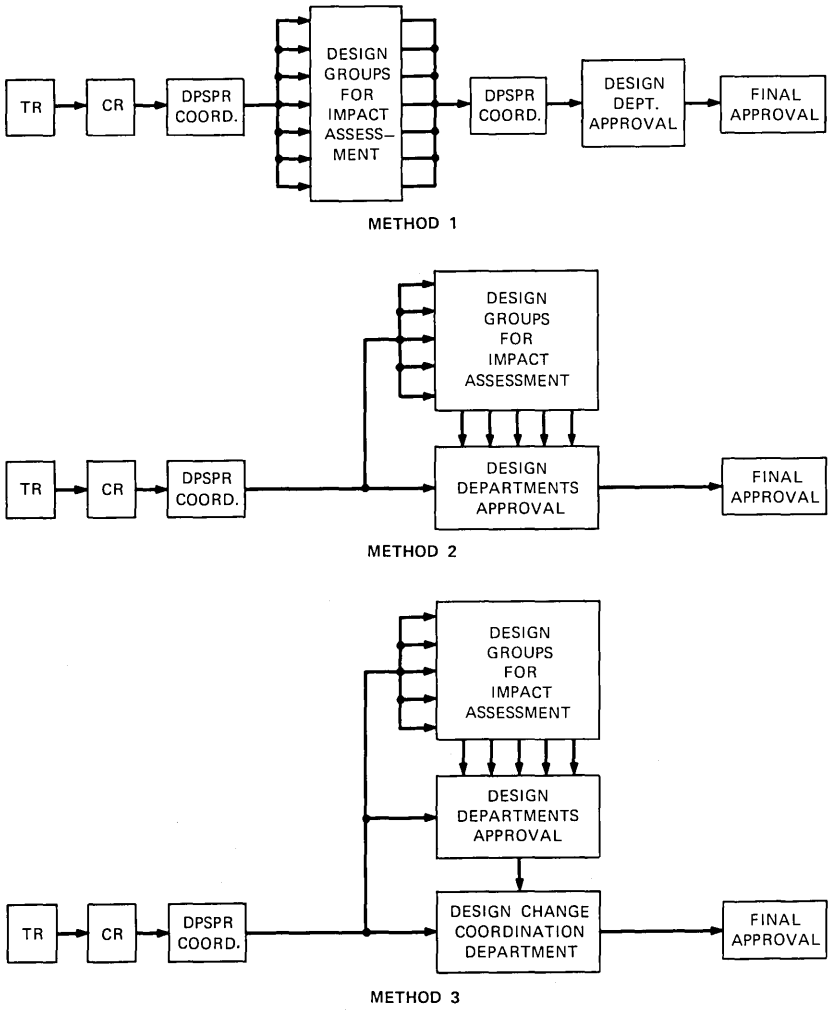

TR
CR
DPSPR COORD.
DESIGN GROUPS FOR IMPACT ASSESS-MENT
DPSPR COORD.
DESIGN DEPT. APPROVAL
FINAL APPROVAL
METHOD 1
TR
CR
DPSPR COORD.
DESIGN GROUPS FOR IMPACT ASSESSMENT
DESIGN DEPARTMENTS APPROVAL
FINAL APPROVAL
METHOD 2
TR
CR
DPSPR COORD.
DESIGN GROUPS FOR IMPACT ASSESSMENT
DESIGN DEPARTMENTS APPROVAL
DESIGN CHANGE COORDINATION DEPARTMENT
FINAL APPROVAL
METHOD 3

Fig. 1—TR/CR approval sequence.

the DPSPR coordinator, the assessments were attached to the CR and the CR was then routed in turn to the head of the system design department, to the head of the software design department affected by the change, to the heads of all other software design departments, and finally to the director of software design. This procedure resulted in significant delays in the return of impact assessments and in department-head routing. It only worked efficiently when the DPSPR coordinator hand-carried the CR through the approval sequence.

The procedure was then changed (Method 2) to one in which the CR was immediately routed to department heads and, at the same time, information copies were sent to all software design supervisors whose design would be affected by the change. When each department head received the CR for approval, he requested software impact estimates from his supervisors. This procedure was more effective than the previous one; however, CRs tended to become backlogged in the department-head routing process. This resulted primarily because no one representative of the design organization had the responsibility to ensure that each CR received appropriate and timely action.

The final procedure (Method 3) was quite similar to the previous one except that one department head was designated as the change coordinator with the responsibility of ensuring that each CR received the appropriate attention and that all software changes were properly coordinated.

The DPSPRs were submitted to the customer for baselining on March 31, 1971. From that time until the DPSPRs were finally baselined in May 1972, changes were allowed in the DPSPRs by means of the procedure described above. This allowed the DPSPRs to be reasonably current during this period; however, additional effort was required by the customer to review the TR/CR changes as well as the submitted DPSPRs. After baselining, the only change to the TR/CR procedures described above was that approved CRs were incorporated into an Engineering Change Proposal (ECP) which required customer approval before the CRs associated with the ECP were forwarded to publications for generation and distribution of revision pages. There were instances, of course, where software design changes had to be made to make the system work before customer approval could be obtained. The control procedures allowed for this as a "management risk."

The control procedure enabled the project to keep track of all problems and their solutions and to control the changes in system design. However, after the document control procedures had been prepared, a few suggestions were made that might have improved the process.

First, in addition to detailing the specific change to the DPSPR, the CR should have included the rationale and/or study that led to the change. In cases where significant changes were made, they were generally documented in a memorandum; however, little or no rationale accompanied many small changes. Including the rationale would probably have reduced duplication of studies that were conducted by the system designer and the software designer to evaluate changes.

Second, the software design organization always should have been a party to the initial approval of a correction report. This was done when the TR was originated by software design, but was not done when a TR was initiated by system design or by system evaluation. By coordinating all correction reports through the software designer, there probably would have been fewer unapproved CRs to rework. This would also have made the software designer aware earlier that a change in his design was being proposed.

Third, the TR/CR approval sequence and publication of the change should have been streamlined as much as possible. Even though the designers knew of the change, most other DPSPR users were not aware of it until the revision pages were issued. One change to the approval sequence that might have shortened the approval cycle time would have been to establish a formal calendar date for final review and approval at the highest necessary management level when the CR began its approval sequence. Each CR would be reviewed on that date and rescheduled if a final approval decision could not be reached. This approach would have forced timely attention to each CR in the approval cycle.

VI. CONCLUSIONS

One of the most fundamental needs in a software development project is a clear statement of requirements. The DPSPRs were designed to meet this need and were successful in doing so. They have also provided valuable insight into the design of testing and evaluation procedures. The most notable deficiency in the DPSPRs was a lack of explicit definition of interfaces among the various functions. More concentrated effort in specifying exact definitions of these interfaces would have greatly helped the software designers. The most important lesson learned in setting and maintaining requirements is that changes to the system design must be carefully controlled. It is essential that software designers be made fully aware of the content and implications of each system change.

Section II

HARDWARE

SAFEGUARD Data-Processing System:

Architecture of the Central Logic and Control

By J. W. OLSON

(Manuscript received January 3, 1975)

The Central Logic and Control (CLC) unit is the digital computer that controls SAFEGUARD. Its development represents the first reduction to practice of large-scale multiprocessing in a computer system. This paper describes the CLC and explains some of the decisions behind its design.

I. INTRODUCTION

The Central Logic and Control (CLC) represents the first practical application of the multiprocessing concept to a large-scale computing system. A modular design is employed in which as many as ten processors and two Input/Output Controllers (IOCs) share as many as 32 memory racks. The units are interconnected by a flexible switching network that allows the system to be partitioned into two independent computers. Partitioning can be controlled by software, and complete reconfiguration may be accomplished in less than one second.

This paper focuses on the architecture of the CLC, and on how system requirements influenced the decisions behind its design.

II. DESIGN PHILOSOPHY

2.1 System requirements

Availability, reliability, and performance requirements are placed on the CLC because of its importance to SAFEGUARD. The data-processing system is required to be fault-tolerant. This means that the system must be able to perform its workload in the presence of any single malfunction. In addition, the CLC is allowed only a limited amount of down time. High-reliability specifications are placed on each of the components from which the CLC is fabricated to increase the mean-time-to-failure. High CLC performance requirements are dictated by the nature of its primary job, controlling a radar tracking system in real time and the launch/guidance of missile interceptors. Sufficient

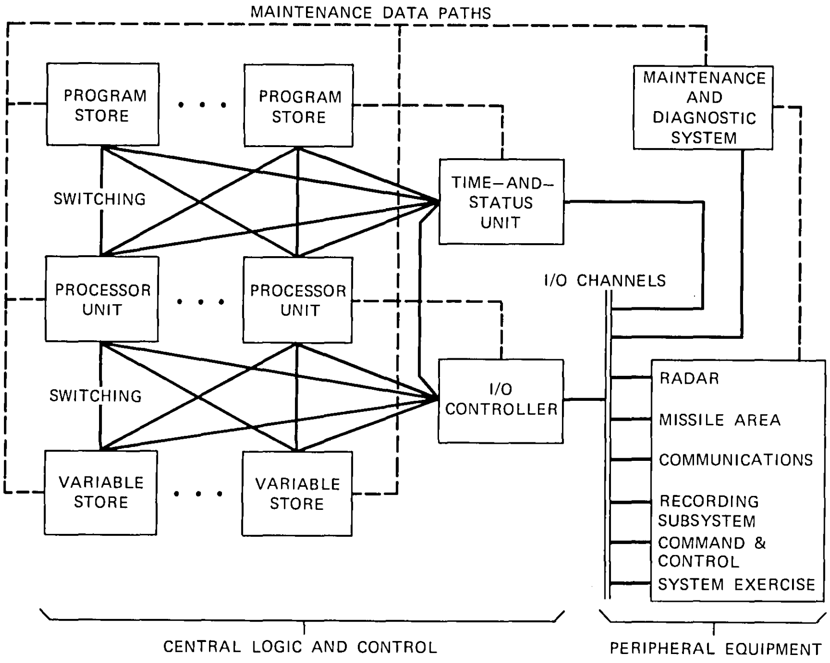


Fig. 1—Central Logic and Control unit.

reserve power must be available to handle peak loads. A block diagram of the CLC is shown in Fig. 1.

2.2 Resulting architecture decisions

2.2.1 Modularity

The CLC is composed of five types of elements: up to ten processors, sixteen racks of program store, sixteen racks of variable store, two IOCs, and two time-and-status units. This system is capable of operation with only one element of each type and may grow in a modular fashion. The IOC provides peripheral-world access to the computer while the time-and-status unit provides a number of special functions which include real-time clocking, monitoring system status, and controlling the configuration of the hardware resources in the system. The multiple elements communicate via well-defined interfaces and are interconnected by a flexible switching network.

The method of interconnecting elements within the multiunit computing system must permit ease of growth and be consistent with the availability and reliability requirements. The switching method chosen for the CLC is based upon a distributed implementation of the switching network such that a portion of the switch is included with each unique system element. Both economic and availability considerations favor a distributed switch in which each added processor and storage element comes with its own portion of the switching system to allow smooth system growth. System availability is enhanced because a failure of a portion of the distributed switching system affects only the unique element to which it is attached.

All communication among elements of these five types is handled asynchronously on a request-and-acknowledge signaling basis. The collection of processors is capable of asynchronously accessing any of the collection of memory elements. The switching network is such that if each processor makes an access to a different memory element, then all may receive service simultaneously. Priority circuits at each memory element resolve conflicting requests sequentially.

2.2.2 Multiple processors

Although it would have been possible to design a single processor system with sufficient performance, the CLC is a multiprocessor machine for three reasons. First, a single processor sufficiently powerful would have been a complex machine, difficult to design and difficult to get working. Second, a single-processor system would not have been expandable; if a more powerful machine were later found necessary and none were available, major software changes would have been required. Also, multiple processors satisfy a wide range of processing requirements including smaller applications. Finally, the multiprocessor design increases availability because processing can continue even if some processors have failed.

2.2.3 Two memory types

A multiprocessor design hinges around its storage design. A number of possible strategies are available to handle the necessary references of the multiple processors to main storage. The first strategy used in the design is the splitting of main storage into two independent portions called program (or instruction) store and variable (or operand) store. This organization doubles the data flow rate to each processor at the expense of independent instruction and operand fetch circuitry within each processor. One of the reasons for this architecture is to physically separate programs and data sets for reliability purposes. Thus, program store is a read-only memory, while variable store is a read-write memory which holds real-time I/O data and provides storage for the results of calculations.* To optimize memory utilization of the CLC

* Program store is read-only in the sense that processors have no instructions that write data into it. Software can alter program store via the store transfer unit which is described in a later section.

during the software development phase, additional switching paths are provided from variable store to each processor to allow instructions as well as operands to be stored in variable store.

2.2.4 "n + 1" redundancy

To achieve nearly continuous operation as economically as possible, the CLC employs $n + 1$ redundancy. Each of the five types of elements has at least a single replacement that is not required for running the application software and is therefore redundant. For example, if the application software requires 15 racks of program store for execution, then at least 16 are provided. The $n + 1$ element may be switched in to replace a failed element.

2.2.5 Partitioning

The CLC can be partitioned into two independently operating computers, each capable of executing its own job stream. By convention, these two partitions have been differentiated by the terms green and amber, with green usually the larger of the two fractions. However, since the computer is composed of a number of modular elements, the boundary defining which are green and which are amber is almost completely flexible, as illustrated in Fig. 2. In fact, all elements may be brought into the green partition to operate as a very large multiprocessor computer with as many as ten processors sharing the job load. As a further degree of flexibility, some elements, such as memory elements, may be placed into a shared green/amber state where they are available to both partitions simultaneously. Finally, an element may be defined

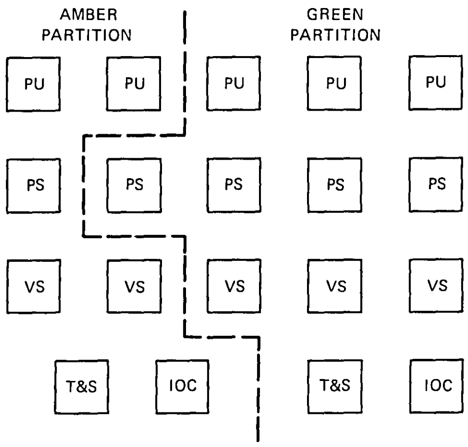


Fig. 2—Element partitioning within the CLC.

to be neither green nor amber and is said to be isolated. This state is necessary to remove malfunctioning elements without shutting down the entire system.

It is even more significant that partitioning is under program control. Further, the control logic for effecting partitioning is redundant. There is a fundamental asymmetry to the control of partitioning which allows the green partition to have priority over the amber partition. The partitioning logic may be placed into a state whereby a master/slave relationship exists between the green and amber partitions. Control software residing on the green partition may alter the partitioning of the system at any time. The amber or slave partition can in no way alter the partition boundaries. This will be described in more detail in Section 3.4.1.

III. DETAILED DESCRIPTION

3.1 The processors

The processor is the most important element in establishing the real-time computing capacity of the CLC, so the design of a high-speed processor has been a primary goal. Each processor contains three control units that operate asynchronously with respect to each other. Timing within each control unit is overlapped to some degree so that more than one instruction may be in execution. High-speed arithmetic algorithms and associated logical implementations have been exploited advantageously to increase the flow of operands through the arithmetic sections. The resulting processor design can execute successive fixed-point add operations on full-word 32-bit operands at an average rate of 4.15 million per second.

The processor organization, as shown in Fig. 3, is best explained by considering a typical arithmetic operation. Three functions must be performed: instruction fetch, operand fetch, and arithmetic execution. Three control units allow these functions to be overlapped, thus avoiding simple concatenation of the functions for successive instructions.

The Program Control Unit (PCU) prefetches instruction words from program store into an instruction word buffer. The PCU then extracts instructions from the buffer and determines which of the control units will participate in executing the instructions. For those instructions involving operand access, the operand control unit will address variable store to fetch or store all operands to be used internal to the processor. For those instructions involving arithmetic operations, the arithmetic control unit will perform all fixed-point and floating-point arithmetic.

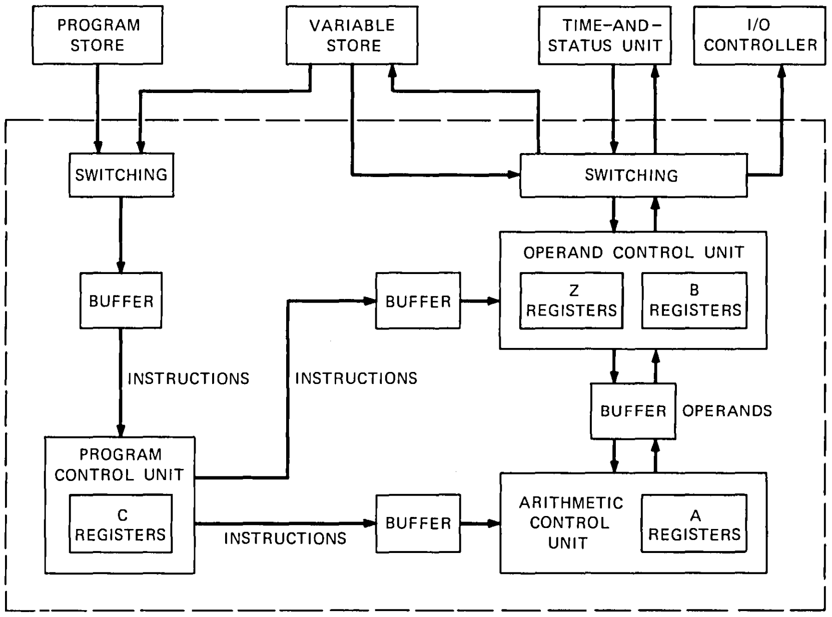


Fig. 3—Processor unit.

3.1.1 Program control unit

The PCU supplies instructions to the operand and arithmetic control units. Reference to program store is by absolute address from a location specified by a program address counter. A change from sequential operation can be effected either by interrupt or by executing a jump. Instruction sequencing is optimized by use of four double-word buffer registers that form an instruction stack. Whenever branches in the instruction sequence are encountered, alternate path fetching is employed to fetch both the normal path word and the jump path word. Both of these words are placed in the instruction stack to await a jump decision. Since many jumps are conditional to an arithmetic test within the processor, having both paths available will in general reduce the time needed to proceed regardless of which jump decision is made. In addition to the above optimizing, short instruction loops may be entirely contained within the instruction stack and executed without further access to program store. To smooth and optimize instruction flow to the other control units, instruction list buffers exist at the interface between the program control unit and the other control units.

3.1.2 Arithmetic control unit

The arithmetic control unit contains fifteen addressable A-registers for temporary storage of operands. All arithmetic operations are performed on operands from the A-registers with results returned to these registers. The registers make data currently in use immediately available to the processor. Within the arithmetic control unit, A_0 is defined to be a fixed accumulator for all arithmetic operations. The A_0-register functions alone as a single-length accumulator or in conjunction with an extension register to form a double-length accumulator. The double-length accumulator will handle the double-length results obtained for multiply operations and will hold the quotient and remainder for divide operations. The two-address arithmetic instructions will always place the result in A_0 and have the option to overwrite the second named register. This method allows some of the generality of a three-address format without the need for a third address.

3.1.3 Operand control unit

The operand control unit fetches operands from variable store; it performs any required operand fetching address arithmetic itself. Fifteen addressable B-registers provide temporary storage of addresses or index values. The operand control unit can perform shifts and edits on data contained in the B-registers. (Edits are instructions that access only a selected portion of a register.) Data can be exchanged between B-registers and A-registers.

The operand control unit provides a set of 15 addressable Z-registers which are used to control the operation of the entire processor. Interrupt jump and return addresses are found in the Z-registers. Memory protection is controlled by these registers; the appropriate bit in a Z-register must be set to allow the processor write access to a particular segment of variable store. One of the Z-registers is a delta clock which acts as an alarm clock. The delta clock will generate an interrupt if it is not reset before a selected primary countdown interval is exceeded.

3.2 The memories

To further increase the data-flow rate between processors and main storage, program and variable store are further subdivided into modular groupings, as shown in Fig. 4. Variable store is organized as 16 independent racks, with an independent data path from each rack to each of the processors. Since queuing is heavier at program store than at variable store, program store is organized as 32 independent modules with an independent data path from each module to each of

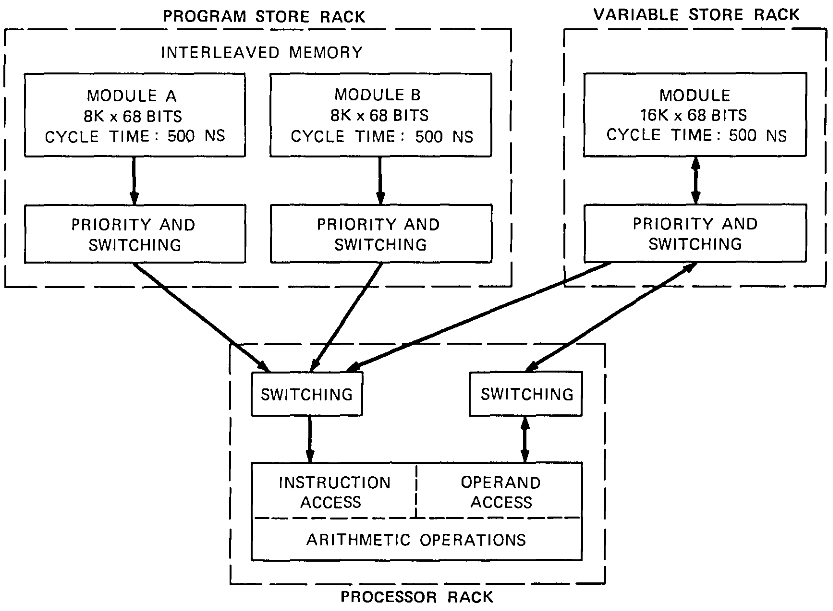


Fig. 4—Processor main-storage organization.

the processors. Processor addressing is interleaved between two modules; that is, the address structure is arranged so that adjacent program store words reside in two separate modules.

The memory module cycle time of 500 ns and the double-word size of 64 bits are selected to provide a memory bandwidth in excess of that required for maximum performance of a single processor. Each program-store and variable-store rack holds 16,384 64-bit words. There are four parity bits associated with each memory word.

In a multiprocessor system, the need frequently arises to prevent one processor from modifying data that another processor is accessing. A lock mechanism is also needed to avoid IOC and processor interference at variable store. To allow resolution of these problems, a special memory instruction called biased fetch is included. A biased fetch reads a word from variable store and, in one memory cycle, restores the word with the upper two bits set to binary ones. (Two bits are chosen because the parity of the memory word is not regenerated during the read/modify/write cycle.) The original word, *before* modification, is returned to the processor or IOC. The processor or IOC can test the upper two bits of this word to determine whether access to the data has been granted. If these bits are zeroes, the data are available; if they are ones, the data are not available.

3.3 The input/output controllers

In any computing system, input/output is of paramount importance and frequently determines throughput. The I/O Controller (IOC), which is shown in Fig. 5, directs the flow of information between variable store and the peripheral devices. Processors are thus relieved from communicating directly with the peripherals. Processors and IOCs can operate simultaneously. The I/O subsystem, which consists of the IOC and its associated peripherals, is duplicated to achieve system availability requirements.

A basic feature of the IOC is its ability to simultaneously and continuously service several peripheral devices. The fastest way to service any individual peripheral device is to transfer its entire block of data by preempting all of the transfer facilities. Since this violates the rule of simultaneous service to several peripheral devices, it is necessary to time-share the IOC facilities among all devices.

Each IOC contains 16 channels; each channel contains independent input and output cables, thereby allowing full-duplex operation. Priority circuits are utilized to allow time-multiplexed operation of the channels.

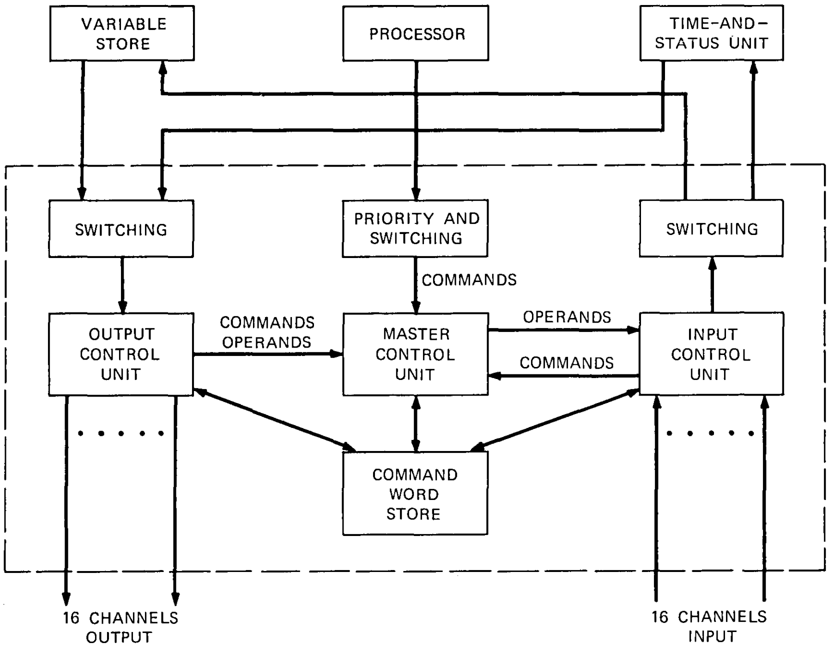


Fig. 5—Input/Output Controller.

Each peripheral is assigned a priority order which takes into consideration the allowable latency of a peripheral device requiring access to variable store. High-speed, synchronous devices usually are assigned higher priority channels than buffered, asynchronous devices.

The IOC is a programmable device. Its operations are controlled by commands it reads from variable store. The instruction repertoire includes jump commands and simple data operation commands. An IOC program can be initiated by a processor or by a peripheral device. The IOC accesses I/O programs by indirect addressing.

3.4 Associated equipment

Although the processors, the memories, and the IOCs are the principal components of the CLC, three other devices deserve mention: the status unit, the timing generator, and the store transfer unit. A block diagram of the time-and-status unit, which includes the above functions, is shown in Fig. 6.

3.4.1 Status unit

The status unit is essentially a register memory that may be read or written by all processors in a given partition. By reading from the status unit, processors obtain information about the condition of the data-processing system: parity errors, time-outs, power on-off, etc. By writing into the status unit, processors control the data-processing system.

Partitioning is controlled by signals from the status unit. Software can specify whether each component of the data-processing system is

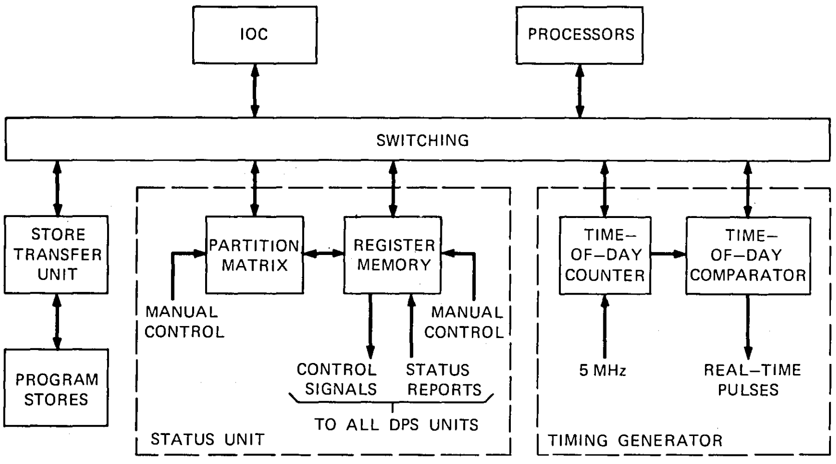


Fig. 6—Time-and-status unit.

to be partitioned green, partitioned amber, or is to be isolated. The status unit enables communication between elements in the same partition and disables communication between elements in different partitions or to elements which are isolated.

Since there are two status units, subtle logic-design problems exist. For example, status information from peripheral devices partitioned amber must affect only the amber status unit and not the green. One status unit must be designated the master and the other the slave in such a way that partitioning signals from the master status unit take precedence. Transients caused by powering up a status unit must not disturb this relationship.

The status unit also interacts with the IOC. If certain status unit bits change, the status unit presents a command request to the IOC. The IOC program thus initiated informs software of the event.

3.4.2 The timing generator

The timing generator performs two basic functions that are essential within a real-time system. The first is that of initiating activities at points in time that can be specified by program means. The second is that of providing an accurate time value which can be used in recording the time of occurrence of specific events during operation of the SAFEGUARD system. This is accomplished in the CLC by providing a time-of-day binary counter which is driven from a precise 5-MHz generator. As with other system components, for availability reasons the timing generator is duplicated. The timing generator is synchronized with a time-of-day standard. In addition, there is a procedure to synchronize the timing generator in the amber partition to the timing generator in the green partition. This is necessary whenever the amber timing generator is shut down for maintenance.

To fulfill the function of initiating activities at specified times, the timing generator performs time-notice comparisons of the time-of-day clock to a time-arranged list of orders stored within variable store. This activity is analogous to that of an alarm clock set to turn on various software processes. This function is handled via an I/O channel to relieve the processor from the housekeeping function of presenting time-notice orders to the clock. As long as the time-notice list has been prepared in advance, the IOC will methodically transfer a new order from the list maintained in variable store. In addition, the IOC will interact with the global data sets maintained in variable store to trigger various software events without necessarily providing a direct processor interrupt.

The second function of providing an accurate time value is accomplished by allowing all processors within the same partition to directly access the clock and fetch time of day as a binary word. Access to the timing generator is designed so that, regardless of the number of processors in queue, each processor may obtain time of day in less than a microsecond. The time-of-day value can be used to attach a time tag to various recorded events or to determine whether certain system deadlines have been missed.

The timing-generator and status units may be thought of as hybrid devices within the CLC from the viewpoint that they may be accessed directly by a processor using the internal switching network within the computer or they may be accessed as a peripheral device using an I/O channel. As these devices either provide control information or report status, they are not accessed frequently during normal operations and so they share the same switching port and I/O channel. To take advantage of the economy of sharing interfaces, they are grouped together in the same equipment rack which is designated the time-and-status unit.

For partitioning purposes, the time-and-status units are paired with the IOCs to which they are attached. Thus, time-and-status unit number one is always configured in the same partition as I/O controller number one. The same philosophy holds true for many of the peripheral devices connected to the IOC in the SAFEGUARD system.

3.4.3 Store transfer unit

The time-and-status unit also includes a third function called the Store Transfer Unit (STU). The STU is the only device that can write into the program store elements. For reasons of economy, it shares the same direct switching interfaces with the timing-generator and status units. New program segments flow from either tape or disc through the IOC to the STU and into the appropriate rack of program store via the internal switching network within the computing system. During recovery of the CLC, the STU associated with the IOC that is on-line at the time handles the reloading of the tactical software process into program store.

3.5 Instruction repertoire

The instruction repertoire for the CLC processor has been specified to accommodate the addressing structure of the computer. The processor can address program and variable store. A 20-bit internal address is used which, when mapped into actual memory addresses, allows addressing the maximum of 256 K double-words for both program and variable store. In addition, the processor contains internal register areas for temporary storage of operands. All arithmetic operations are performed on operands from the registers. The use of this type of memory hierarchy separates the two functions of operand fetch from main storage and arithmetic execution. The instruction repertoire takes this into account so that the access of operands from variable store is distinct from arithmetic operations.

The addressing structure of the CLC will accomodate dynamic relocation of data sets. This requires that the processor have the capability to store and modify addresses locally within its registers. A method of double indexing is employed, using the contents of as many as two B-registers and a 12-bit displacement value contained within the instruction itself, to form an address value.

There are two different instruction lengths, 16 bits or 32 bits. Most instructions work with operands contained in the fast internal registers. The method of addressing operands from these registers is characterized by the use of two-address instructions with register addresses in the range of 0 to 15. These instructions use the half-word (16-bit) length which contains an 8-bit operation code and two register addresses. Instructions which access variable store utilize the longer 32-bit instruction length. In addition to the operation code and address displacement value, the memory access instructions also specify two B-registers used in address generation and the source or destination register in either the A, B, or Z register areas. There is also an instruction which references variable-store operands in absolute fashion using a full 20-bit address field contained within the instruction. In addition, a subset of instructions, designated "true" instructions, permit constants to be stored within the instruction itself. These constants may be directly loaded into the internal registers of the processor.

The processor can handle both fixed-point and floating-point data represented in fractional two's complement notation. All arithmetic operations are normally performed on 32-bit operands for both fixed- and floating-point data. Exceptions include a half-multiply instruction, the ability to manipulate exponents, and the ability to perform address arithmetic on 20-bit values. Floating-point numbers are usually normalized. There is no hardware capability to perform double-precision arithmetic.

3.6 Hardware concept

The SAFEGUARD hardware concept permits fabrication of the data-processing system from a standard stock of racks, chassis, and integrated-circuit packages. The design is based upon integrated-circuit technology using a modified direct-coupled-transistor-logic circuit having circuit delays in the 5- to 6-ns range. The hardware provides a flexible system for interconnecting groups of integrated-circuit packages on chassis, and chassis into racks as shown in Fig. 7. To enhance

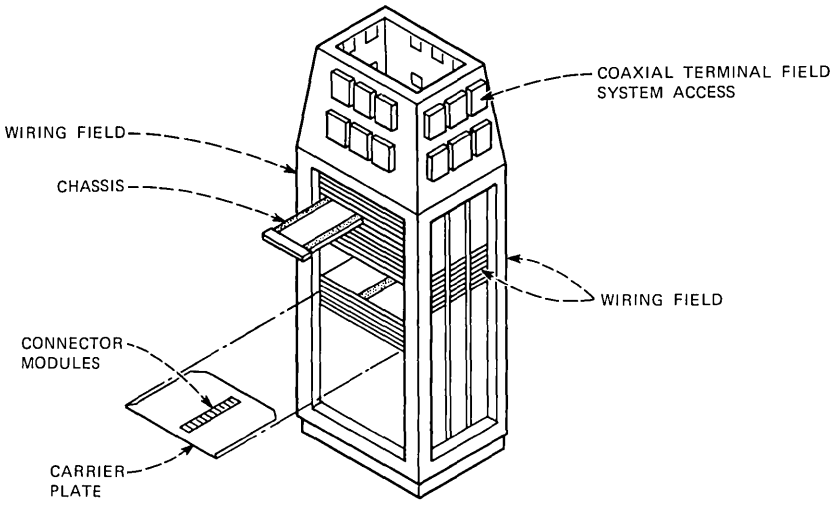


Fig. 7—SAFEGUARD rack.

reliability, the integrated-circuit packages are wire-wrapped to achieve connections on the chassis. Each chassis can accommodate 275 integrated-circuit packages and, therefore, more than 600 logic circuits. The chassis are housed in a water-cooled rack with two chassis mounted side by side on a chassis carrier plate which locates, supports, and cools the chassis. The chassis carrier plates are mounted on a 1-inch vertical pitch within the rack. There are a maximum of 59 levels in the rack housing 118 chassis.

It was recognized that a large multiprocessor would present a need for a large number of access connections. In fact, there is a need for more access connections to the chassis than could be provided with rear access only. Therefore, the chassis also uses both sides for additional access terminals. The rear contacts to the chassis are made in a conventional plug-in manner. The side contacts use a linear-actuated cam arrangement to engage the side contacts after the chassis has been situated properly in the rack. This arrangement results in wiring fields on three sides of the rack. In addition, internal connections are provided at the interface between the chassis, which are situated side by side on the carrier plate, to provide near-neighbor connections between groups of chassis. In total, the rack provides for more than 40,000 possible signal connections. It should be noted that having rack wiring on three sides has resulted in a diamond orientation of racks on a floor plan to allow physical access to all four sides of a rack. Rack-to-rack interconnections are provided by plug-in coaxial terminal fields at the top of the rack which allow as many as 11,520 connections in this area.

To preserve the integrity of the high-speed pulse transmission between the various units that make up the multiprocessor, a characteristic impedance of 100 ohms is maintained for the transmission of all signals. Coaxial cables are used for all connections between racks and for all rack wiring runs in excess of five feet. Twisted pair is predominantly used to wire the rack. The chassis connector maintains a fixed impedance across the connection by providing both a signal and a ground path using a highly reliable double-contact arrangement to gain entry to a chassis.

The memory racks include a 16-K by 68-bit-per-word core memory unit and the associated interface logic switching circuits which provide interconnection to the multiple units in the system. The core memory units are air-cooled and operate at a cycle time of 500 ns and have an access time of 300 ns.

3.7 CLC performance

One of the primary reasons for the development of a parallel and modular computing system for SAFEGUARD is the potential for high performance. In addition to the properties this architecture possesses for high availability, a multiprocessor organization possesses a great deal of reserve power which, when applied to a problem with the appropriate degree of parallelism, can yield high performance. This is the type of problem which is associated with a radar tracking system and which must be solved in real time.

In a multiprocessor system, the processors gain access to main storage according to a priority rule. The rate at which each processor executes instructions depends, therefore, on the severity of this queuing at main storage. Throughput will be defined as the number of instructions of a particular instruction mix executed per second by n processors.

Adequate performance, or throughput, of a parallel processing system depends upon a number of hardware factors, which include the speed of the processor, the speed of program store including its priority circuit, the total number of processors relative to the total number of independently addressable program stores, and the number of instructions executed per memory word fetched. From a software viewpoint, the distribution of programs and data sets within the modular memory and the instruction mix of the particular programs in execution are also important factors which directly affect throughput.

Since variable store queuing will, in general, be less than that at program store, its effect has been eliminated in the throughput data presented here. This has been done by dedicating a separate variable store rack to each processor for experimental studies.

Throughput data have been gathered using multiprocessor hardware with configurations containing as many as ten processors. Benchmark programs have been used which provide varying instruction mixes. Four instruction mixes were selected for testing. The NOP mix, consisting of no-operation instructions, defines an upper bound on throughput. The LOGICAL mix is a representative mix that is similar to CLC operating system code that might be executed during real-time operations. The MATH mix is also a representative mix, being a portion of the cosine subroutine from the CLC operating system. The JUMP mix consists exclusively of jumps and represents a kind of lower bound on throughput.

Figure 8 shows the effect of requiring all processors to execute out of one program store. The number of instructions executed per second increases with the number of processors until the program store is returning instructions as fast as it can. Throughput levels off when this point is reached, and a further increase in the number of processors does not increase throughput.

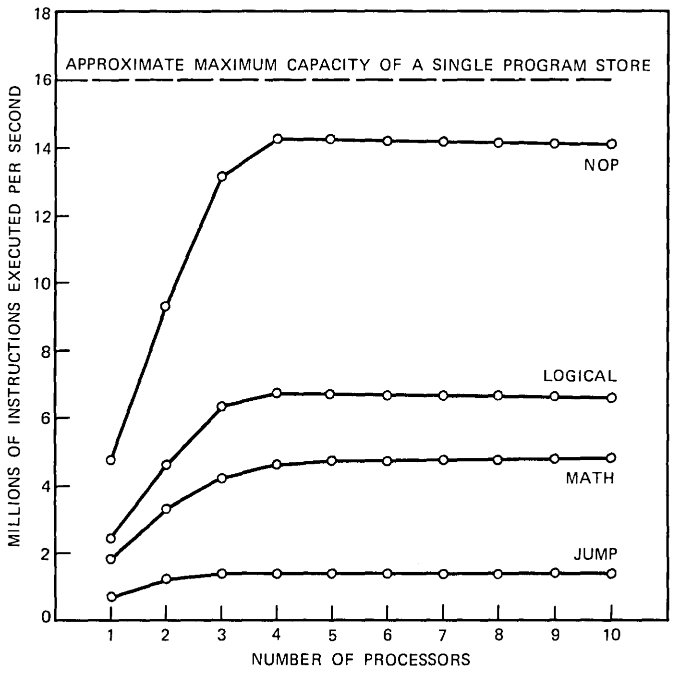


Fig. 8—N processors executing from one program store.

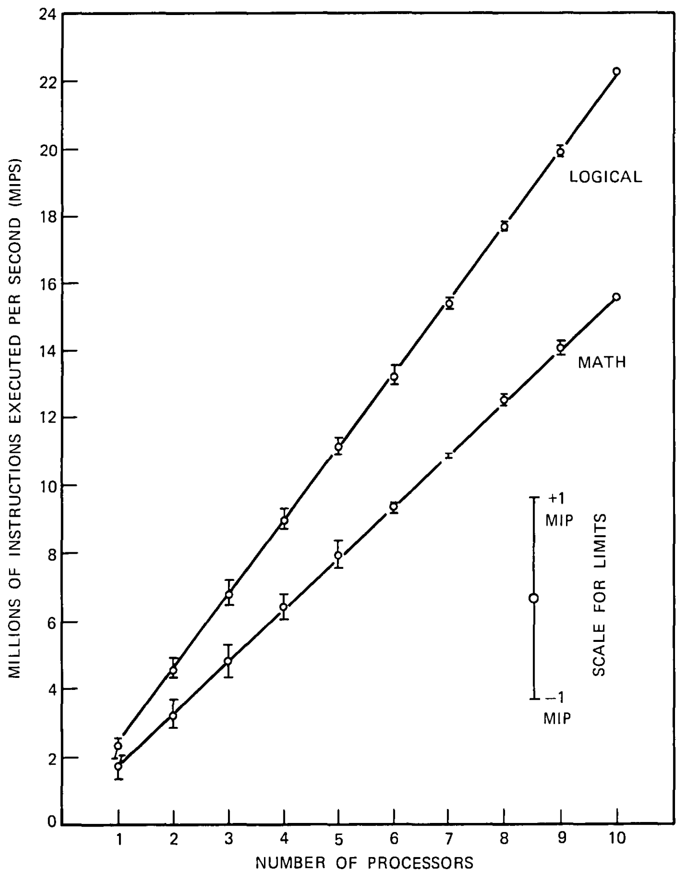


Fig. 9—N processors executing from N program stores.

Figure 9 shows the effect of providing an equal number of processors and program stores. For this case, the number of processors and program stores is incrementally increased from one to ten. The program stores are not dedicated to a processor on a one-for-one basis, but their access by the processors is randomized such that several processors may be attempting to read from the same program store at once. Hence, some reduction in throughput due to queuing is expected. The effect of queuing is small for one to ten processors. Figure 9 shows that throughput increases linearly with the number of processors. Data are shown for the LOGICAL and MATH mix only.

The data presented in Fig. 9 are for the case of an even distribution of memory access over all program stores. It is interesting to determine what happens to throughput for the case of an unequal work-load distribution. A series of runs were made for both the LOGICAL and MATH mixes where the number of processors was kept equal to the number of program stores with one important difference. One of the program stores was selected as a "favored" program store and its fraction of total instructions executed was varied from 0 to 100 percent while the remaining program stores shared the remaining work load equally. Figures 10 and 11 show the results for the six to ten processor cases. The curves represent throughput as a function of the "favored" program store. Zero percent means ten processors are executing out of nine program stores. Note that throughput is a maximum when the "favored" program store shares equally in the work load.

The curves of Figs. 10 and 11 are useful in that they show the sensitivity of throughput to an unequal distribution of the work load in memory. For instance, if one considers a 10-percent reduction in throughput to be serious, the above curves show for the seven-processor case that a single program store can have almost 40 percent of the work load without a serious reduction in throughput. For the ten-processor case, the corresponding number is approximately 25 percent.

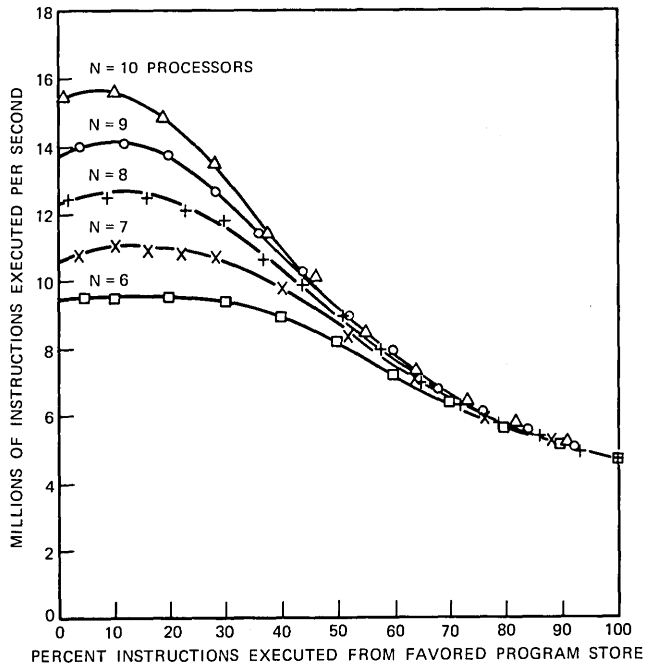


Fig. 10—Unequal program-store loading—MATH mix.

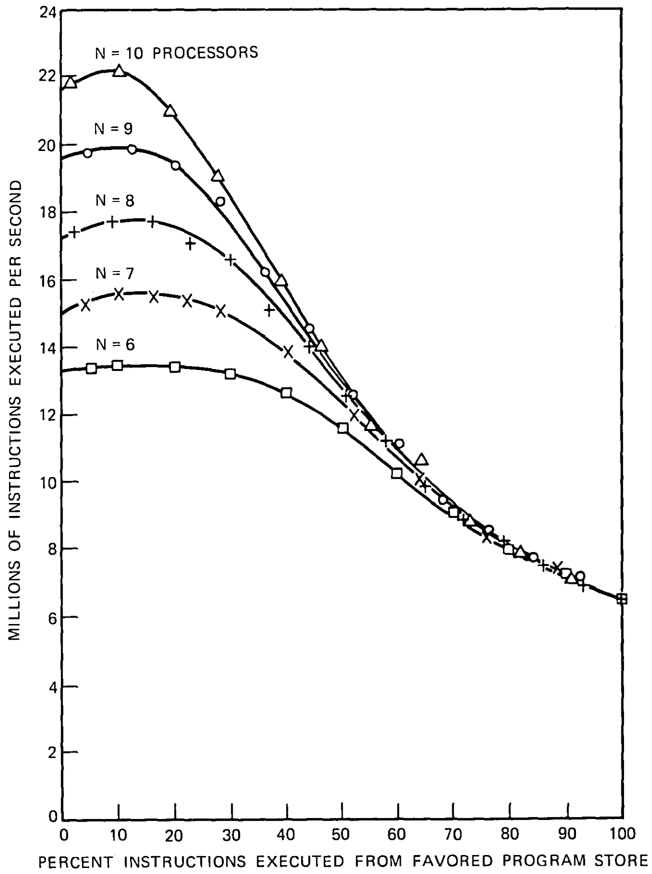


Fig. 11—Unequal program-store loading—LOGICAL mix.

Therefore, as long as the work load is not too unequally distributed, the dependence of throughput on work load distribution should not be critical. Throughput dependence on more than one program store having more than an equal share of the work load has not been investigated.

IV. CONCLUSIONS

4.1 Success of the modular design

The use of the well-defined interfaces and modular hardware building blocks capable of communication within the framework of a distributed switching system provides the basis for a dynamic computing complex—a structure that is capable of incorporating new functional units offering unique economic or performance advantages.* This structure has been very useful in satisfying the wide range of computing applications within the SAFEGUARD system. These range from a single processor, nonredundant installation to a ten-processor, maximum-sized system. Not only does this structure handle the wide variations in system sizing, but it can easily accommodate changes that may result from new or revised system requirements.

4.2 Reduced cost for "$n + 1$" philosophy

Historically, early fault-tolerant systems, such as ESS-1, employed 100-percent redundancy through use of a complete standby system.[1] That is, the system required to support the full work load is duplicated, with data processing proceeding in parallel on each system. This organization is conceptually simple and upon detection of a failure in either system, the other system can carry on the data-processing work load.

The multiunit system approach to gaining high performance can provide high system availability without the need for costly, complete duplication. The $n + 1$ redundancy approach has reduced the amount of equipment added for redundancy and for system exercise to a fraction of that required for a complete standby system.

4.3 Instruction repertoire

The CLC instruction repertoire was designed long before CLC software was written. As a result, programmers seldom use certain instructions and often wish for others. For example, character manipulation instructions are lacking, as is one instruction that will store all processor registers.

4.4 Status-unit performance

The status unit, as implemented in the CLC design, represents a comprehensive method of gathering system status and providing configuration control information to the various parts of the data processing system. The use of the status unit to control the configuration of a partitionable machine is unique and has been proven successful during the SAFEGUARD project.

* This structure, for example, will very easily accomodate the addition of an array processor, such as the Parallel Element Processor Ensemble (PEPE), or it will easily allow direct connection of a high data rate peripheral subsystem to the modular variable stores. Although not a part of the present SAFEGUARD system, extensions to the multiunit architecture, as described above, have been seriously considered and are entirely feasible.

4.5 CLC performance

The performance of a multiprocessor system depends upon a number of factors including the speed of the processor, the speed of the memory element and the speed of its priority circuit, the total number of processors relative to the total number of independently addressable memory elements, and the number of instructions executed per memory word fetched. The distribution of programs and data memory and the instruction mix of the particular program being executed are also important. CLC performance as a function of the number of processors and the number of independent program-store data paths has been measured by D. B. Knudsen, and the information presented in Section 3.7 is a result of that effort.

V. SUMMARY

The requirement that a computer function properly even though some of its components fail has been a primary goal in the development of the SAFEGUARD computer. The multiprocessor approach was chosen to achieve high performance and availability. The multiunit architecture has provided a system which satisfies a wide range of computing requirements on the project through the use of a single design.

REFERENCE

1. R. W. Downing et al., "No. 1 ESS Maintenance Plan," B.S.T.J., *43*, No. 5, Part 1 (September 1964), pp. 1961–2020.

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Maintenance and Diagnostic Subsystem

By J. R. HAHN, JR. and F. E. SLOJKOWSKI

(Manuscript received January 3, 1975)

The SAFEGUARD Maintenance and Diagnostic Subsystem (M&DSS) is a unique, independent, hardware group within the data-processing system through which the nonreal-time functions of fault detection and isolation are performed. In this paper, the M&DSS hardware and fault detection software are described and system performance is reviewed.

I. INTRODUCTION: AN OVERVIEW OF SAFEGUARD MAINTENANCE OPERATIONS

The specific tactical mission for which the SAFEGUARD system has been designed is of extremely short duration compared to the life of the system. Once such a mission has begun, fault isolation and repair are of no concern; at this point, mission success in the face of hardware failures is totally dependent on real-time fault detection and, when necessary, the automatic execution of system recovery. Thus, the fault detection and isolation features of the Maintenance and Diagnostic Subsystem (M&DSS) are oriented primarily toward the goal of maximizing system availability, the probability that, at any random point in time, a complete set of fault-free Data-Processing System (DPS) resources exists.

The M&DSS contributes to maximizing system availability in two ways. First, M&D tests are periodically run on critical DPS equipment to supplement real-time fault detection methods in minimizing the mean-time-to-awareness of hardware faults. These tests are automatically scheduled by real-time software in the green partition and the test requests are sent to the M&DSS over a special interface through the status unit. In this way, every processor in the DPS is switched into the amber partition and tested once every hour; the complete amber partition is tested once each hour; and the green I/O controller with its slaved peripheral controllers is switched amber and tested once every four hours. The M&DSS passes test results back to green system software again via the status unit interface.

Second, and more important, the M&DSS minimizes the mean time to repair of faulty racks by rapidly identifying a minimum set of replaceable or easily repairable modules in which the fault is located. These fault isolation functions may be initiated in response to fault symptoms detected either in real time or during the nonreal-time scheduled tests described above. In either case, fault isolation takes place with the failed rack isolated from the rest of the DPS.

The M&DSS accomplishes this goal through the unique integration of two significant maintenance concepts. First is the use of a special two-way maintenance data path into each DPS digital unit, which bypasses normal data paths. Second is the use of a small general-purpose computer dedicated to system testing, which applies tests over the maintenance paths and interprets test results.

The communication interface between the green partition status unit and the M&DSS provides a rapid and flexible means for bringing maintenance resources to bear on any DPS fault indication. Nonetheless, until a specific faulty rack has been identified, the particular response to be made to any given fault indication often involves judgments based on the total status of DPS resources. Thus, normal SAFEGUARD maintenance operations involve a significant degree of manual interaction. In general, two primary maintenance management functions are performed manually:

(*i*) Monitoring and response to overall system status as reported by green system real-time software and hardwired displays.
(*ii*) Direct control of maintenance testing: The M&DSS will not honor any scheduled test request unless manual "permission" is granted, any test in progress may be manually aborted, and alternate tests may be requested via green system software and the status unit interface.

II. THE SAFEGUARD MAINTENANCE TASK

In its largest configuration, the SAFEGUARD DPS consists of as many as 50 digital racks, each containing up to 100 logic chassis. Each chassis can have between 500 and 600 logic gates. A total installation can have over 2000 chassis with over 500 unique chassis designs. Approximately two million distinguishable faults can occur distributed over these 2000 logic chassis in the typical installation.

The primary goal of the SAFEGUARD M&DSS is to provide rapid fault isolation for the largest, most common class of faults likely to occur.

Other, more subtle faults will involve longer isolation times, but by optimizing isolation for the most common faults, the required overall mean time to repair will still be met. Several assumptions are made concerning this major class of faults which must be handled by the M&DSS:

(*i*) Only hardware faults are considered.
(*ii*) Only permanent faults are considered. Transient and intermittent faults, when they occur in the green partition, are handled by real-time error response mechanisms.
(*iii*) All faults have equal probability of occurring.
(*iv*) Only one fault will occur at a time: Measured device failure rates support this assumption.

These assumptions, along with further assumptions regarding real-time fault detection capabilities and the distribution of the various classes of faults expected, provided input to a series of parametric studies designed to arrive at specific M&DSS design objectives. The studies led ultimately to the goal of a four-hour mean time to repair for 90 percent of all DPS faults. The mean time to repair includes the time to:

(*i*) Isolate the fault to a reasonable number of suspect chassis.
(*ii*) Remove these chassis and test them on an automatic test set that identifies the specific faulty chassis and the failed circuit pack.
(*iii*) Repair the chassis.
(*iv*) Replace all chassis and verify the repair.

An analysis of the possible trade-offs of time between these activities led finally to the requirement that the M&DSS be capable of isolating 90 percent of the class of faults defined by the assumptions above, to three or less logic chassis within 15 minutes of their detection.

III. M&DSS HARDWARE

The conventional approach to digital fault diagnosis involves applying a set of input data to the particular circuit under test and, by comparing the output of the circuit to an expected value, deducing the location of the possible circuit faults that could have caused any observed differences. Obviously, the larger and more complex the circuit between input and output, the greater the number of circuit faults that could cause any specific output error, and the greater the ambiguity in the final fault resolution. The primary design feature of the M&DSS (Fig. 1) is aimed at overcoming this problem.

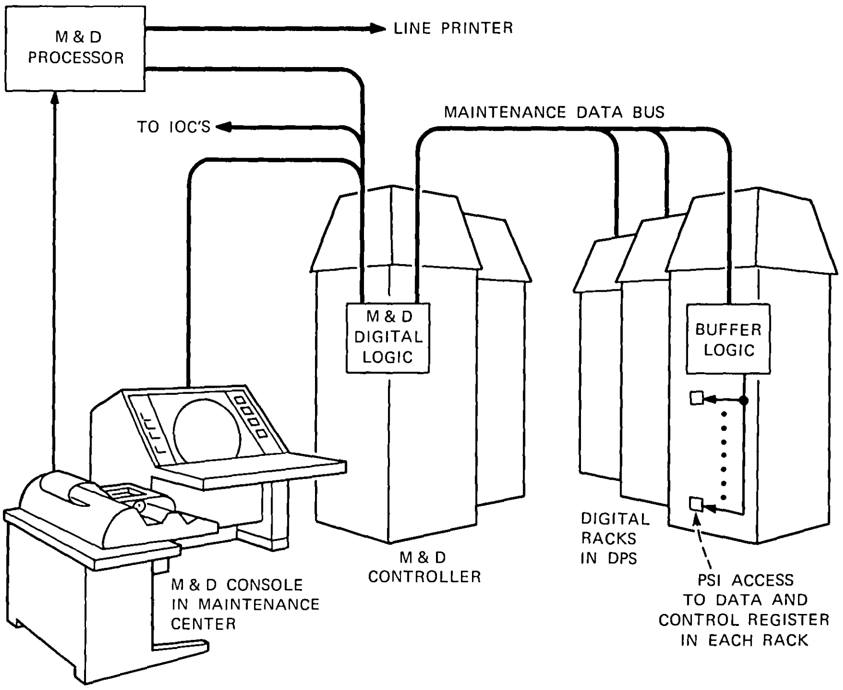


Fig. 1—Maintenance and Diagnostic Subsystem.

Every digital rack within the SAFEGUARD DPS is equipped with a unique internal logic interface to the M&DSS. This interface consists of special programmable Pulsed-Set-and-Indicate circuits (PSIs) connected to most data and control registers within the rack. These circuits provide the means to read from or write into these registers independent of normal data paths. The PSIs are connected via an internal data bus to a maintenance buffer chassis within the rack through which the PSI'd registers may be selectively accessed.

The proper placement of PSIs was an integral part of the logic design process for each SAFEGUARD digital rack. Through PSI access, large blocks of sequential logic are effectively dissected into smaller combinational blocks, each having a number of inputs and outputs accessible via the M&DSS. This not only makes it quite simple to implement system recovery, as will be explained later in this paper, but also results in two important advantages related to fault isolation. First, it makes possible considerably greater fault resolution than can be had in standard logic design. Second, it makes practical the simulation approach to fault dictionary construction.[1]

Testing a digital rack, therefore, involves the repetitive execution of a simple four-step "program":

(*i*) "Set" data onto one PSI-accessible register.
(*ii*) "Set" bits in one or more control registers to enable circuit operation.
(*iii*) "Indicate" (read) the contents of another PSI-accessible register.
(*iv*) "Compare" the result to an expected value.

The execution of such programs is one of the primary functions of a digital rack called the M&D controller. The M&D controller receives maintenance programs from one of several program sources, translates and executes the program in a unit called the sequencer, and communicates with the rack being tested through fan-out logic called a data tree. The data tree is connected to the buffer chassis of each digital rack in the DPS through a separate maintenance channel.

Once the communication channel to a particular rack has been established, the sequencer uses this channel to set data into and read data from selected registers within the rack. Data returned through the "read" instructions can be compared within the sequencer to an expected value and the results of the comparison will be returned to the program source. Again, these three operations, write, read, and compare, are the essence of the sequencer function. The sequencer can also specify up to two additional channels to allow interface maintenance tests between racks.

DPS recovery is implemented through the M&DSS via sequencer "write" instructions stored in a protected core memory (part of the M&DSS itself) and designed to accomplish two functions:

(*i*) Set the appropriate partition bits in the status unit to configure a minimum DPS.
(*ii*) Initialize operational registers in selected DPS racks to boot-load a simple DPS control program and pass control to it; this program then completes the recovery operation.

When recovery is initiated, the M&D sequencer automatically switches to the recovery memory as its program source.

Since the M&DSS is used for both fault diagnosis and system recovery, it must be extremely reliable. The M&D controller, the heart of the M&DSS, can overcome most single faults within itself. It has built-in redundancy, built-in fault detection logic, and PSI access that permits the application of M&D tests to one of the redundant sequencers via another. The chassis involved in system recovery are duplicated, as are the stores containing the system recovery programs.

IV. NONREAL-TIME MAINTENANCE SOFTWARE

The M&D test program itself is the most basic unit of nonreal-time maintenance software. Conceptually, the design of an M&D test is quite straightforward, in keeping with the limited command repertoire of the M&D sequencer described above. Design begins at the level of "micro" tests, each oriented toward a single logic circuit path. Each consists of a number of set-up instructions that set a test vector into a register via PSI access, further instructions which toggle the necessary control bits to cause the test vector to propagate through the logic path to an "output" register, and finally an instruction to compare the output data to an expected value.

From 200 to 2000 such "micro" tests might be designed to cover all the circuits within a logic block. The size of a logic block depends on functional boundaries of logic within a rack. Five to ten such logic block tests typically make up the total test for a single SAFEGUARD digital rack; over 300 block tests are involved in the maintenance facility for the largest SAFEGUARD DPS configuration.

Three independent means exist for applying M&D tests to the digital equipment. The first and most direct means employs a mobile console that is used only during installation of a site. This console, containing a simplified version of the main M&D controller, has its own control panel and associated tape machine. The mobile console connects to the normal M&D buffer chassis in each rack to verify the operation of the rack before the installation of system cabling.

After system cabling is installed, the M&D controller has direct access to each rack, and the second means of applying tests is made available. This consists of the M&D console (shown in Fig. 1) through which tests are transferred to the M&D sequencer from magnetic tape, and test results are displayed on a cathode-ray tube (CRT).

Both the mobile console and the CRT console, however, are extremely slow, depending on magnetic tape as a test program source. Moreover, both return test results to the user in the form of an identification of the compare instructions that failed and the resulting error patterns. Fault isolation then requires a fairly knowledgeable maintenance man to interpret test results. Thus, while the CRT M&D console is a part of the tactical maintenance center, it exists primarily as an emergency backup to the third and most important test facility, the M&D Processor (MDP).

The MDP is a modified CDC Model 1700 general-purpose digital computer. It provides the means for fully automatic high-speed selection and transfer of tests to the M&D sequencer and the automatic interpretation of test results.

The total collection of M&D logic block tests is stored on MDP disc along with all MDP operating software, including a test control program that accepts commands ranging from a request to test a single logic block to a request for a test of an entire digital subsystem.

These test commands may be sent to the MDP automatically from green partition software or manually from its own TTY. In this latter mode, which is normally used for fault isolation, the test program saves the error symptoms (M&D noncompares) encountered and then requests that the fault dictionary tape for the logic block test which detected the fault be mounted on one of the MDP tape transports. Another MDP program then searches the dictionary to find fault lists for the noncompares detected. After the lists are processed, the result is printed out as a list of suspect chassis.

The MDP provides the additional bonus of extending the diagnostic capabilities of the M&DSS beyond PSI-accessible boundaries. The use of fault dictionaries is limited to SAFEGUARD digital logic, but faults in other equipment may be diagnosed by applying functional tests through PSI-accessible registers in a digital unit that interfaces with the unit being tested. An MDP program controlling the test analyzes test results as they occur and branches to other tests along a program path that terminates with the identification of one or more likely faulty circuit cards, or the output of an error code pointing to a written manual procedure to be followed for a final fault resolution. This approach has been successfully applied to the main SAFEGUARD memories and CRT consoles and their supporting equipment.

V. M&DSS APPLICATIONS AND PERFORMANCE

Any evaluation of overall SAFEGUARD M&DSS performance must, of necessity, consider the entire maintenance concept, not only the M&DSS itself, but also the role of the partitionable DPS, its status unit interface with the M&DSS, and the function of system recovery. All play a significant part in achieving the required system availability/reliability product.

At this time, however, the full-scale system tests that will eventually yield specific maintenance system performance data are just beginning. Nonetheless, data do exist in two categories. Extensive testing has been done on the detection and dictionary-isolation capabilities of the basic M&D tests.[1] The M&DSS has also been used extensively in the maintenance of the DPS equipment at the tactical sites during the installation and test period. Maintenance experience in this environment, while not directly translatable to the tactical situation, has produced considerable insight into M&DSS performance.

More than anything else, experience to date has demonstrated the fundamental power and flexibility inherent in the primary M&DSS feature, the extensive maintenance data interface with the entire DPS, in concert with the general-purpose computing capability of the maintenance data processor. Just as encouraging, however, has been the performance of a set of extended M&DSS capabilities developed during the early phases of installation and operation, before the widespread availability of M&D tests and dictionaries. A brief description of these capabilities is instructive as background for the quantitative performance data to be discussed later.

Central to all the extended capabilities of the M&DSS is a set of MDP programs known as Digital Unit Exercisers (DUX). One such program exists for each unique DPS rack type. Each DUX program provides the capability to control the functional operations of a rack on a macroscopic level and to "dump" the contents of individual registers or groups of related registers within the rack. DUX perform these functions by accepting commands in a functional language, translating these commands within the MDP into appropriate M&D sequencer "write" commands, and transferring these to the sequencer for execution. Subsequent "read" commands are used to dump the desired registers, and the results are output on MDP peripheral devices.

In actual hardware maintenance operations, DUX have been used primarily to provide manual interaction, via the M&DSS, with a set of real-time programs originally developed to verify the complete functional capabilities of the DPS.* Data currently being gathered at SAFEGUARD sites show that this mode of fault detection and isolation continues to play an important role.

Table I shows the results of data that have been gathered on the actual use of all MDP resources for a three-month period at the tactical sites. As mentioned earlier, the basic M&DSS and MDP software capabilities were designed to optimize fault detection and isolation on the most common class of faults anticipated, namely, single "hard" device failures. This class is shown in the table under the heading Hard Faults. The Other category includes timing and intermittent failures, design errors, and a variety of miscellaneous failures, largely mechanical in nature. It is important to note that these data were gathered midway during the site test and integration period, a time when design errors are indeed expected to be uncovered, and when frequent handling of the equipment, because of change activity, directly contributes to a greater number of intermittent and mechanical problems.

In view of these facts, the data shown in Table I are extremely encouraging. They show that, for the period covered, the M&DSS success-

* Though not the subject of this paper, it is worth noting that the various DUX capabilities also provide an extremely powerful means for system *software* debugging by allowing dumps and snaps of otherwise inaccessible DPS registers without perturbing the very condition being probed. This capability has found extensive use throughout SAFEGUARD software development.

Table I — MDP performance (July–September 1973)

Total Faults*		Fault Type		Grand Total (75)
		Hard Faults (51)	Other (24)	
M&D tests only	Detect.	96% (49)	83% (20)	92% (69)
	Isol.	92% (47)	54% (13)	80% (60)
DUX/ITPs required	Detect.	8% (2)	0% (0)	3% (2)
	Isol.	17% (4)	0% (0)	11% (4)
All MDP resources	Detect.	96% (49)	92% (22)	95% (71)
	Isol.	100% (51)	71% (17)	91% (68)

* In those cases where isolations exceed detections for a given capability, the fault was usually first detected by a user program. The CDC 1700 was then used to gather enough additional data to achieve isolation.

fully achieved its design goals with respect to the Hard Fault class. Moreover, through use of the MDP extended capabilities, the M&DSS achieved at least its detection goals with respect to all faults.* Finally, the M&D tests alone come very close to achieving design objectives for all faults. Experience, then, supported by the data shown above, leads to a number of specific conclusions regarding M&DSS performance.

Maintenance considerations must be an integral part of logic design. SAFEGUARD development schedules did not allow two or three iterations of the PSI placement-simulation-evaluation cycle. As a result, during test design, cases were discovered where additional PSIs, or a more efficient distribution of existing PSIs, would have produced significant improvements in fault detection, isolation, or both. In particular, more PSI access to control circuits and within logic feedback loops would have made it possible to define smaller and more independent logic blocks. In the most serious cases, hardware change orders were processed to add or rearrange PSIs. Nonetheless, nonoptimum PSI placement remains as the single most significant limitation on detection and isolation.

Increasing the speed of the entire M&DSS would significantly extend its fault-detection capabilities. In its present design, the M&DSS executes a complete read-write-compare cycle in approximately 35 μs, more than two orders of magnitude slower than many internal logic events in the DPS. In the design of the M&DSS, speed was sacrificed for reliability; for example, communication between the M&D controller and each DPS rack is in serial form to minimize the number of con-

* Isolation times using DUX are significantly longer than for M&D tests. Thus, we cannot conclusively say whether or not the goal of 15-minute isolation for 90 percent of all faults has yet been achieved.

nectors, relatively low-reliability components, in the entire path. As a consequence of this design decision, however, the M&DSS is limited in its ability to detect failures that only affect logic timing. A compare instruction can verify whether or not the expected value eventually appeared in a PSI's register, but not whether it arrived there on time. If, however, the M&DSS operated at system speed, it would be more effective in diagnosing this class of faults.

The extended capabilities of the M&DSS described earlier in this section are effective, however, in compensating for both the shortcomings owing to M&DSS speed and those owing to insufficient PSIs. By using M&D access to load and set into execution the more complex real-time functional test programs, the effects of timing faults and faults in complex control circuits can be detected. DUX capabilities can then be used to sample various PSI'd registers along the more elaborate functional path exercised by the test program, and the results can be interpreted to obtain fault isolation to a functional level. In fact, there are very few DPS fault conditions that cannot be handled by one or another of the maintenance tools available through the M&DSS. It is this aspect of experience that leads to a final conclusion on M&DSS performance.

The total M&DSS concept offers great power and versatility as a digital maintenance facility. "Total concept" means the integral combination of PSI access and general-purpose computational control of the PSIs. On-line dictionary search makes possible the rapid isolation of the largest class of common device failures, while the extended capabilities available through the MDP allow the remaining faults to be dealt with in such a manner that the only limitation is the ingenuity of the maintenance man.

In retrospect, the full range of M&DSS capabilities has yet to be fully explored. For example, again because of project schedule constraints, the logic block partitions originally defined have not been changed; but different partitions, chosen perhaps with timing faults specifically in mind, might allow timing faults to be handled via straight M&D test/dictionary methods. Conversely, the real-time DPS capability verification tests that have proven to be so useful in conjunction with the DUX might themselves be restructured with fault isolation more in mind (they were not originally designed for this purpose); it would then be possible to use the MDP to analyze the fault symptoms obtained through PSI access to yield on-line chassis level isolation information.

REFERENCE

1. C. J. Rifenberg, "SAFEGUARD Data-Processing System: The Dictionary Approach to Digital Maintenance," B.S.T.J., this issue, pp. S73–S85.

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

The Dictionary Approach to Digital Maintenance

By C. J. RIFENBERG

(Manuscript received January 3, 1975)

This paper provides an overview of one aspect of the SAFEGUARD approach to digital maintenance—the Maintenance and Diagnostic (M&D) program-fault dictionary. The M&D program detects the presence of faults. The associated fault dictionary provides fault lists for automatic fault isolation; it is generated by executing the maintenance program in an environment simulating the action of hardware in the presence of faults. The paper also provides some detailed discussion of simulator-performance improvements.

I. INTRODUCTION

A SAFEGUARD data-processing system consists of racks of equipment for three functional areas: a large real-time central computer facility, a large peripheral subsystem, and a Maintenance and Diagnostic Subsystem (M&DSS).[1,2] This paper describes an essential aspect of the SAFEGUARD maintenance plan, the Maintenance and Diagnostic program-fault dictionary.*

Fault-isolation dictionaries are available for most maintenance programs. Dictionaries provide a correspondence between fault-diagnostic-test failures and possible hardware faults (or faults of replaceable units) which could cause the failures. They have been used successfully in the No. 1 Electronic Switching System (ESS)[3,4]; however, ESS and SAFEGUARD dictionaries differ in their format, generation, and use. Both Armstrong[5] and Godoy[6] have described a method for efficiently simulating the action of hardware in the presence of faults. Their technique is used in the generation of SAFEGUARD fault-isolation dictionaries.

* Maintenance and Diagnostic programs are described by Hahn and Slojkowski.[1] In addition, supplemental maintenance programs are used to test hardware, which cannot be exercised by these programs, or to provide increased fault detection.

A test-control program accesses the dictionaries to isolate detected faults. After receiving results of test failure, the test-control program performs set union and intersection operations on the sets of fault lists in the dictionary entries associated with failed and passed tests to isolate them to an acceptable number of replaceable units (chassis).* If a maintenance program is completed without failure, the test-control program can either consider the rack fault free, schedule additional maintenance programs for execution within the M&D controller, or schedule supplemental maintenance programs.

II. CONSTRUCTION OF SAFEGUARD DICTIONARIES

2.1 Approach

The dictionary approach to fault isolation was chosen early in the design cycle primarily to satisfy a requirement that craftspeople with moderate skill, working at a large number of installations, be able to quickly accomplish fault isolation. Simulation was considered as the only feasible method for generating dictionaries since there was no hardware time available for fault insertion, and the logic was too complex for manual dictionary generation.

Figure 1 is a block diagram of the Logic Simulation Facility (LSF). For simulation purposes, each rack is divided into several, often overlapping, logic blocks, none of which exceeds 20,000 gates. This maximum gate count is a serious design limitation which occasionally causes functionally integral logic blocks to be subdivided. Had time permitted, this design limitation would have been eliminated. Each SAFEGUARD data-processing system has over 300 maintenance programs designed to detect faults in the logic blocks. The tests within the program are designed manually. Most of these programs have associated fault-isolation dictionaries generated through simulation. A few programs (mostly for rack interface blocks) were not simulated since they were only testing a small portion of a block functionally much larger than 20,000 gates. Before programs are run on the simulator, they are debugged on the hardware to verify that predetermined logic values within compare instructions are correct. By debugging on the hardware rather than the simulator, the possibility that the simulated logic block is incorrectly constructed or initialized is eliminated.

Circuit interconnections and other pertinent wiring information for the computer units are described in manufacturing tape files. The data in these files are used by an automatic wire-wrap machine to wire the

* These chassis (500 to 600 logic gates) are, in turn, repaired by replacing integrated-circuit packages.

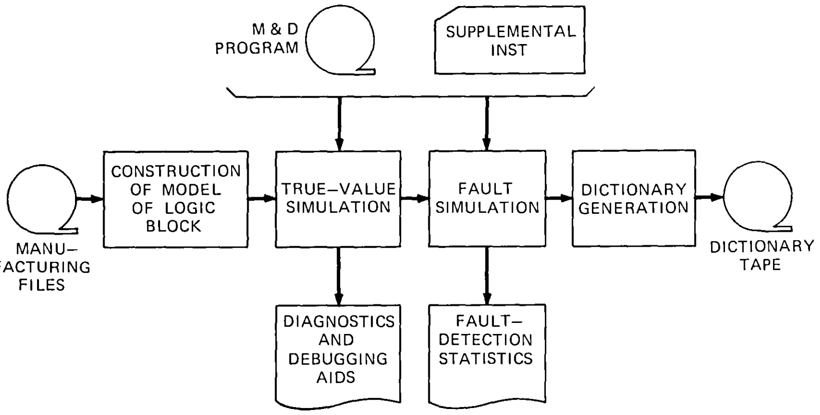


Fig. 1—Logic simulation facility.

chassis and racks. These files are used to construct a simulation data base and to simulate the hardware at the logic-gate level.

In addition to the manufacturing files, there are two primary inputs to simulation: the maintenance program discussed above and a set of supplemental instructions. These supplemental instructions enable the test designer to set any gate in the simulated logic to any logic state. They are particularly useful in initializing gates on the boundary of the logic block which are driven from logic not being simulated. It is through these instructions that the simulated logic block goes from an unknown state to a state representing the hardware at the start of testing.

The true logic value simulation, pictured in Fig. 1, is a simulated execution of the maintenance program in the absence of faults. Since the program has "run clean" on the hardware (i.e., all compare instructions are correct in predicted true logic value), the true logic value simulation is used to find discrepancies between simulation and hardware execution of the maintenance program. Discrepancies are usually caused by erroneous supplemental instructions or by deficiencies in the logic-block data base. These differences are usually resolved through changes to the instructions or data base. Standard aids are provided to assist in identifying causes for discrepancies (e.g., gate timing traces of change from known to unknown logic value).

The LSF fault simulator is a deductive simulator (see Ref. 3). At any given simulation time, each gate in the circuit has a true logic value (possibly unknown) and a fault list (possibly null) associated with it. A gate's fault list contains all faults in the circuit which, if present singly, would complement the true logic value of the gate. Every fault present in a gate's fault list is said to be detectable at the gate. The simulator assumes that only single, hard faults occur in the hardware. Transient failures, most timing faults, and marginal faults are not considered. Unit gate delay is assumed. At each interval of simulation time, the fault-free logic value and the fault list for a gate are computed if either the logic value or fault list of any of the gate's inputs has changed in the preceding time period. When a compare instruction of the maintenance program is simulated, the instruction number and all faults associated with the compared register (i.e., all the faults which, if present singly, would cause a bit to be complemented from its true logic value) are output to a fault tape for later dictionary generation. Thus, for each compare instruction, there exists a list of faults which are detected by that compare instruction due to their causing an incorrect logic value in the compared register.

Statistical programs provide the maintenance programmer with both summary and detailed information on the faults detected and faults simulated but not detected. This output from the simulator, in many cases, is more important than the dictionary (described in Section III) and is a significant advantage of the simulation approach to dictionary generation. The statistical information is used locally to improve the detection quality of a given program. It is used globally in directing efforts to improve detection in certain areas (e.g., to design a supplemental maintenance program) or conversely, to suspend effort in an area already achieving good detection.

2.2 Simulation performance improvements

The initial version of the simulation facility required extensive computer usage for dictionary generation. Estimates indicated full utilization of an HIS 635 computer for a period of about two years. Even this large cost was an underestimate since many programs would have to be simulated more than once either because the corresponding hardware was significantly changed or because the program was significantly modified to improve detection. Therefore, considerable effort was devoted to reducing computing requirements. Some resource-use reduction resulted from internal algorithm and code modification. The four major items below, however, have most significantly reduced resource requirements, with a cumulative effect of approximately a ten-fold reduction.

2.2.1 Fault list paging

Core storage requirements for fault lists can become excessive. This necessitates partitioning of the simulation into n fault runs, each simulating faults in only $1/n$ of the total number of gates. Results for partitions are merged into a single dictionary. Since the entire M&D

Table I — Computer time savings due to software paging

Block	Partitions		Total Elapsed Hours	
	No Paging	Paging	No Paging	Paging
1	100	13	210.0	110.0
2	50	12	19.4	10.0
3	20	6	11.1	3.5
4	6	1	7.3	2.2
5	4	1	5.1	1.9
6	4	1	4.0	1.3

program must be simulated for each partition, the time required for calculating the true logic values is multiplied by the number of partitions. When n becomes large this introduces a very significant overhead. However, it was determined by experimentation that core requirements for fault lists during simulation peak sharply after a few tests and then fall off quickly (particularly after implementation of other performance improvements to be described). An objective of reducing the number of partitions and total elapsed time was then met by a fault-list paging algorithm which minimized the time required during the absence of paging at the expense of time required during demand paging. The number of partitions for very large blocks is not always reduced to one in order to prevent the paging overhead from exceeding the overhead inherent in dividing the block into a few partitions. On the average, the number of partitions required is reduced by about 75 percent while elapsed computer time is reduced by 40 to 75 percent. Table I provides some sample computer time savings due to demand paging of fault lists.

2.2.2 No simulation of conditionals

In simulation, unknown logic values appearing on the output of gates can be due to either one or more uninitialized boundary-access terminals or to a race condition in a flip-flop. The fault list associated with a node whose state is unknown is not unique, since detection of a fault is dependent upon the particular logic value present. Armstrong[5] provides a method for nonexact treatment of fault lists in the presence of unknowns in order to reduce simulation time. The method was successful because the majority of unknowns appear only transiently and are replaced by known states before monitoring is performed. This method flags faults as "conditional" if their detection is conditioned on the logic value actually existing at an unknown input. It provides a more accurate simulator than one which ignores conditionals.

Experimentation was performed on the trade-off involved between computer time required for simulation of conditionals versus decrease in fault isolation by nonsimulation of conditionals. Simulation of conditionals required from four to ten times as much computer time as did nonsimulation of conditionals. An additional 3 to 5 percent of the faults in the test blocks had no chassis isolation or wrong chassis isolation when dictionaries were generated without simulating conditional faults. It was concluded that conditionals should not be simulated so that computer time could be more profitably used.

2.2.3 *Fault elimination*

Fault isolation is essentially a process of applying tests and observing passes and failures (i.e., a fault signature) until only faults on an acceptably small number of replaceable units have the same signature. For example, Table II illustrates fault signatures for three faults. Faults a and b are indistinguishable in signature while Fault c is distinguished from a and b at Tests 5 and 9.

The effect upon isolation of not simulating all faults for all tests was investigated; e.g., one could stop simulating a fault after it is detected once or twice (i.e., fails one or two tests). In the example in Table I, if a fault were no longer simulated after one detection, Faults a, b, and c would now be indistinguishable since they have the same signature through the first detection (i.e., Test 4). On the other hand, if the fault were no longer simulated after two detections, Faults a, b, and c would have the same isolation as simulating all faults for all tests, since Fault c is still distinguished from a and b at Test 5.

Several blocks were simulated varying the number of detections required before a fault was eliminated from simulation. Results showed that eliminating a fault after two detections provided dictionaries with essentially the same isolation as eliminating a fault at three or more detections; yet simulation time (all other factors being equal) was reduced by 80 percent compared with no fault elimination. Table III provides some representative statistics. The net simulation time savings is even greater since eliminating faults after two detections

Table II — Sample fault signatures

Faults	Tests									
	1	2	3	4	5	6	7	8	9	10
a	P	P	P	F	F	F	F	P	P	P
b	P	P	P	F	F	F	F	P	P	P
c	P	P	P	F	P	F	F	P	F	P

Note: P = Test passes in presence of fault. F = Test fails in presence of fault (detects fault).

Table III — Computer time savings due to fault elimination

Eliminations*	Savings†	1‡	2‡	3‡
1	92	83	97	100
2	88	90	98	100
3	83	91	98	100
No Elim.	—	92	99	100

* Number of detections prior to elimination.
† Percent simulation time savings vs no elimination.
‡ Percent detected faults isolated to 1, 2, 3 chassis.

contributes to the sharp peaking of core requirements for fault lists and, therefore, is partially responsible for making fault-list paging possible.

2.2.4 Fault collapsing

Another attempt at reducing simulation time was "fault collapsing," i.e., merging two faults if detection of one guarantees detection of the other. For example, consider the string of invertor gates shown in Fig. 2. The effect of the output of C being stuck in logic value one is indistinguishable at the monitorable output from the effect of the output of A being stuck in logic value one. Therefore, a test will either detect both faults or neither fault. If both faults are located on the same replaceable unit, there is no loss in isolation by "collapsing" one onto the other and simulating only one of the faults. In order not to reduce replaceable unit isolation, strong restrictions are placed on candidates for collapsing. Only faults on strings of gates located on a single chassis are considered for fault collapse. Thus, a fault might be isolated to the wrong integrated-circuit package but not the wrong chassis. Typically, 15 percent of the faults in a logic block are collapsed resulting in computer savings of about 10 percent. One problem experienced with this limited fault collapse is that additional time is required to evaluate the accuracy of the simulator, and to repair chassis based on the ambiguous integrated-circuit-package isolation information in the dictionary. Table IV summarizes the results of the parameter trade-offs.

2.3 Other methods tried and their limitations

A study of a technique for building dictionaries, called Reachability List Dictionaries (R-LIST), was conducted. Figure 3 is a diagram of

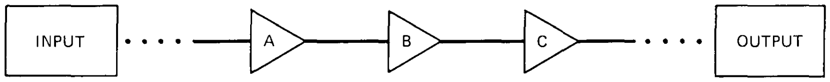


Fig. 2—Sample logic string.

Table IV — Summary of computer time savings vs dictionary degradation

Parameter	Time* Savings (%)	Dictionary† Degradation (%)
Paging	40–75	0
No simulation of conditionals	60–90	3–5
Fault elimination	80–90	1
Fault collapse	10–12	0

* HIS 635 elapsed computer time vs full simulation.
† Percent of faults with no or incorrect chassis isolation compared with a dictionary created without using the parameter.

a simple logic block. An R-LIST is associated with each output gate (e.g., 4, 5, and 6).

The R-LIST contains all gates (or faults) that lie on paths which feed the gate. The R-LISTS may be derived from the total connectivity matrix for the complete block. The R-LISTS may also be obtained by performing a reverse trace to all input (or boundary) gates to the logic block (e.g., gates 1, 2, and 3 of Fig. 3). The R-LIST can be created from the logic block description alone, without any dynamic simulation. Therefore, there was promise of providing a very economical method of generating dictionaries providing the isolation was good. Experiments were performed to determine the isolation capability of dictionaries constructed using these techniques. They showed poor isolation capability because:

(*i*) Lists were much longer than expected and embraced many chassis. Each list contained over 50 percent of all possible fault-producing gates.
(*ii*) Lists overlapped; that is, many of the gates in any one list appeared in all lists.*

Problems associated with automatically generating tests for large, asynchronous, sequential logic are well known.[7] It is difficult to adapt known test-generation algorithms to such circuits. SAFEGUARD designers were successful, however, in supplementing manually generated tests with automatic addition of compare instructions to outputs not already monitored. Usually, outputs were not monitored because the complexity of the circuit was such that the programmer did not realize the full effect of establishing correct logic configurations on control lines. The simulator was modified to "look" at all output points

* The R-LIST technique was further refined and met with somewhat greater success when applied to ESS 1-A.

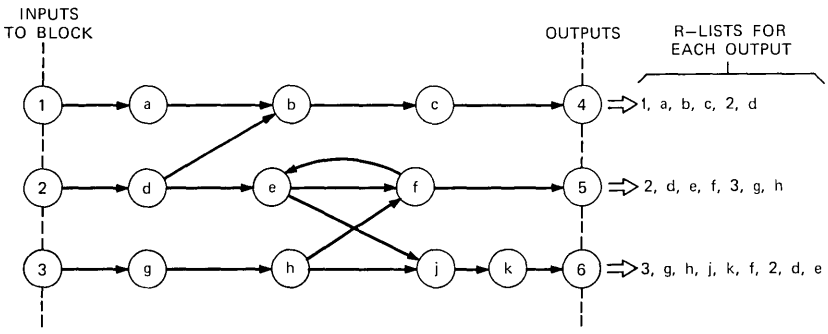


Fig. 3—Sample block with output gate R-LISTS.

for additional propagated faults. This simple technique is being used to increase detection by 3 to 10 percent (an increase which typically required several programmer months).

III. EXPERIENCE WITH DICTIONARIES

Dictionary entries are associated with test compares which could detect a fault. Functionally, a dictionary entry appears in the form shown in Fig. 4. For example, if Test N failed (i.e., observed output was not 101_2) with observed error pattern 110_2, Faults F, G, C, D, and E are candidates for having caused the failure. The test-controller program on the CDC 1700 computer processes dictionary entries corresponding to both matched and mismatched test compares in order to compute a list of faults that have fault signatures consistent with observed test results. The test controller then prints out a list of suspect chassis (with suspect integrated-circuit packages) ordered by chassis with the greatest number of faults on the computed list.

A sample of 31 dictionaries was examined to determine the number of suspect chassis associated with each compare and with each error

TEST COMPARE N

TRUE LOGIC VALUE 101_2

THREE POSSIBLE ERROR PATTERNS

(1) 000_2 WITH POSSIBLE FAULTS
A,B CHASSIS 1
C,D,E CHASSIS 2

(2) 110_2 WITH POSSIBLE FAULTS
F,G CHASSIS 1
C,D,E CHASSIS 2

(3) 111_2 WITH POSSIBLE FAULTS
A,B CHASSIS 1
H CHASSIS 3

Fig. 4—Functional dictionary entry.

pattern within the compare. For example, in Fig. 4, test compare N shows that faults from three different chassis could cause the test to fail. Figure 4 also shows that faults on only two different chassis could cause any of the three possible error patterns.

These results show that chassis lists are usually short (on the average, 95 percent of the error patterns for a dictionary had three or fewer suspect chassis). Figure 5 indicates that both the tests and the logic are functionally designed; i.e., groups of tests are usually exercising logic that has been reasonably arranged on a small number of chassis. This fact contributes to making the dictionary useful even when there is no exact match between an error pattern in the dictionary and the one occurring during the running of the maintenance program, as is shown below. It also contributes to the success of the above-mentioned performance improvement studies.

Additional testing was performed to determine the accuracy of the simulator and the degree of dictionary isolation. Test approaches included limited hardware fault insertion, comparison with another independent simulator, off-line analysis of dictionaries, and vigorous program testing of simulator versions. The results confirm that the simulator accurately generates dictionaries for hard faults, and dictionaries usually isolate detected hard faults to three chassis more than 90 percent of the time (i.e., if one can detect the hard fault, one can isolate it).

Table V summarizes three different ways of evaluating how well the dictionary approach isolates faults. The first column shows the experimental results from actually inserting 102 randomly chosen faults

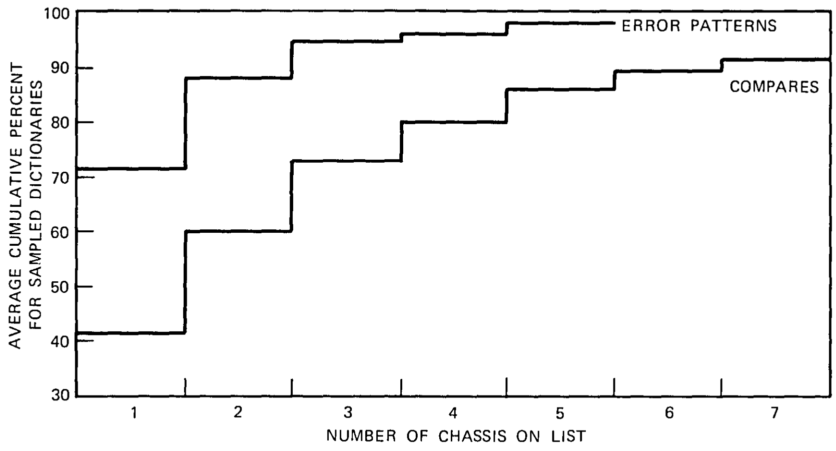


Fig. 5—Typical number of chassis per fault list.

Table V — Preliminary fault isolation evaluation results

Number of Chassis	Fault Insertion (%)	Independent Simulation (%)	Analysis (%)
1	76	70	85
2 or fewer	87	81	96
3 or fewer	93	92	98
4 or fewer	94	94	99
5 or more	4	3	1
No or wrong	2	3	—

into a processor (99 detected). Physical fault insertion exercised the processor dictionaries in their actual environment. Faults were isolated to three chassis 93 percent of the time. The second column summarizes the results of simulating 261 detected faults on an independent simulator and then searching the appropriate 32 dictionaries for isolation. Finally, the third column summarizes the results obtained by analyzing the 300,000 possible detected faults covered in 19 randomly chosen dictionaries. The size of the 19 logic blocks covered by the dictionaries ranges from 12 to 31 chassis and averages 22 chassis. This analysis assumes that when the M&D program is run on the hardware in the presence of a fault, the first two detections of the fault will occur exactly as predicted in simulation and, therefore, will always yield correct isolation (i.e., the isolation list for a fault is exactly the set of chassis associated with the first two detections). The advantage of this type of analysis is easy determination of the approximate isolation for very large numbers of faults. Again, isolation to three chassis is better than 90 percent.

Dictionary isolation evaluation is continuing with emphasis on increased hardware fault insertion, off-line analysis of dictionaries, and initial field experience. Results to date have been generally consistent with those presented in Table V. In fact, dictionaries have been used in the field to isolate to the integrated-circuit package. The feedback to programmers on detection has been instrumental in improving the quality of program fault coverage. Simulation statistics on processor programs, for example, show they now detect 87 percent of the simulated detectable faults. A four-man committee reviews simulator information on undetected faults and makes recommendations for improvement code. This technique has increased detection by as much as 25 percent in some areas. In most cases, the maintenance program was resimulated after the recommended improvement code was added. In such cases, the simulation data base was first made consistent with the latest hardware changes. In a few cases, where the computer time for simulation was large, improvement code was added to the end of the program so that the dictionary remained correct with entries corresponding to the added program instructions at the end of the dictionary.

Hardware changes which cause a divergence from the simulated hardware are a significant problem. These hardware changes eventually cause maintenance programs to noncompare when run against fault-free hardware. Such a condition causes rapid modification of the maintenance program. Often, however, the corresponding dictionary cannot be immediately regenerated. Since it is difficult to quantify the resulting dictionary degradation, maintenance personnel eventually lose confidence in the dictionary and stop using it. Dictionaries seem to be worthwhile for hardware that is modified only occasionally.

There has been much discussion about the need for a "nonexact-match"* strategy to handle such items as marginal, transient, and multiple faults or faults improperly handled due to parameter trade-offs or minor hardware change. The general strategy of on-line processing of dictionary entries allows a very simple algorithm for isolating faults causing exact match. Nonexact match can be handled by interaction between maintenance personnel and dictionary. Simple information requests, such as "List all chassis associated with the first six noncompares or previous six compares," can be answered from the general dictionary entry (see Fig. 4). Such information tells maintenance which logic was being tested at the failed instructions. Since the chassis list is usually short, it is a good starting place for further manual troubleshooting. Thus, maintenance personnel can use the dictionary in homing in on the fault. Not all this interactive capability is currently available. A microfiche print summarizing dictionary entries (i.e., which logic chassis could cause the failure) is being provided to allow such interaction, although less conveniently.

IV. CONCLUSIONS

As others have noted, simulation facilitates detection feedback. Statistics provided by simulation agree with the laboratory experiments (i.e., they are believable). Since the statistics indicate which faults are not detected, they enable the M&D programmer to improve detection, resulting in a better maintained system. Since good detection is required for good isolation, this benefit of simulation should be considered when one chooses a dictionary generation method. It often

* A nonexact match situation results when a fault causes the maintenance program to noncompare when it is run on the hardware and the dictionary entries do not indicate any fault consistent with the observed error patterns.

overshadows the dictionary itself. If the simulator is efficient, the augmented M&D program can be resimulated.

The consistently high quality of the processor-unit dictionaries, for example, indicates the practicality of dictionaries for large logic blocks (20,000 gates) using SAFEGUARD hardware technology.[1,2] Both the fault model and the simulation were simplified, yet isolation remained quite good. (In fact, multiple faults were often correctly isolated.) Thus, dictionaries for large, stable blocks are useful in isolating faults to a small number of chassis. Because the format actually indicates suspect integrated-circuit packages, the dictionary is further useful in repairing the chassis. On the other hand, dictionaries are marginal, at best, for very small logic blocks, blocks with very low detection, or blocks subject to a very high rate of hardware change activity. Dictionaries can be regenerated for blocks experiencing high hardware change order activity providing the computer time required for regeneration is reasonable.

REFERENCES

1. J. R. Hahn, Jr., and F. E. Slojkowski, "SAFEGUARD Data-Processing System: Maintenance and Diagnostic Subsystem," B.S.T.J., this issue, pp. S63–S72.
2. J. W. Olson, "SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control," B.S.T.J., this issue, pp. S41–S61.
3. H. Y. Chang, E. Manning, and G. Metze, *Fault Diagnosis of Digital Systems*, New York: Wiley Interscience, 1970.
4. H. Y. Chang and W. Thomis, "Methods of Interpreting Diagnostic Data for Locating Faults in Digital Machines," B.S.T.J., *66*, No. 2 (February 1967), pp. 289–317.
5. D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," I.E.E.E. Transactions on Computers, *C-21*, No. 5 (May 1972).
6. H. C. Godoy and R. E. Vogelsberg, "Single Pass Error Effect Determination (Speed)," I.B.M. Technical Disc. Bulletin, *13* (April 1971).
7. S. A. Szygenda, "Problems Associated with the Implementation and Utilization of Digital Simulators and Diagnostic Test Generation Systems," International Symposium on Fault-Tolerant Computing (March 1971).

Section III

REAL-TIME SOFTWARE DEVELOPMENT

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Central Logic and Control Operating System

By J. P. HAGGERTY

(Manuscript received January 3, 1975)

The Central Logic and Control (CLC) is the digital computer that controls SAFEGUARD. This paper describes the novel features of the CLC operating system, presents its design rationale, and points out its limitations. Emphasis is on the characteristics that make the operating system suitable for applications other than SAFEGUARD. These include its ability to control as many as ten processors, its ability to initiate the execution of a program within milliseconds of an event, and its ability to detect and isolate faulty hardware racks without manual intervention.

I. INTRODUCTION

The Data-Processing System (DPS) at a SAFEGUARD installation is controlled by a stored program computer, the Central Logic and Control (CLC). CLC software can be divided into a set of applications programs plus an operating system. From the point of view of the operating system, all applications programs are simply the *user* or the user *process*.

Although assemblers, compilers, and linkage editors are usually considered part of an operating system, the CLC operating system provides none of these. All program preparation takes place on a separate support computer, currently an IBM System 370.* The programs compiled and link-edited on this machine, including the operating system itself, are brought to the CLC on magnetic tape as *load modules*.

II. THE ARCHITECTURE OF THE CLC†

The CLC consists of one to ten identical processor units sharing a common memory system, two Input/Output Controllers (IOCs), and two Timing Generators (TGs). Processors are independent of one

* The reasons for this are discussed in Ref. 1.
† A more complete hardware description appears in Ref. 2.

another in the sense that each executes its own instruction stream without knowledge of the instruction stream being executed by any other processor. An *interrupt* causes a processor to switch instruction streams in response to an error condition it has detected, such as arithmetic overflow. Memory is of two types: program store, read-only core from which processors may fetch instructions but not data; and variable store, ordinary core which processors may read or write. Memory racks are shared and not associated with a particular processor so that any processor can reference any memory location. Processors always reference program store by absolute address; they may reference variable store either by absolute address or through base registers.

Data transfer between the CLC and its peripheral devices is performed by an IOC that operates independently of the processors. IOC programs residing in variable store may be initiated either by a processor or by a peripheral device; these programs may perform elementary storage-to-storage operations, such as setting or clearing bits in variable store, as well as I/O.

The IOC controls a variety of peripherals. Some of these are conventional data-processing devices such as the disc drive units, the magnetic tape transports, the card reader, and the line printer of the recording subsystem; the teletypes; and the cathode-ray tube displays of the display subsystem. Other equipment such as the radar subsystem, the missile subsystem, the TG, and the Maintenance and Diagnostic Subsystem (M&DSS) are also considered peripheral devices only because they communicate with the IOC rather than with the processors directly.

The TG, part of the CLC, contains a time-of-day clock incremented every 200 ns. The TG can cause the initiation of an IOC program when a specified time of day has been reached. By suitable IOC programming, this notification may be made repetitive.

The M&DSS is particularly important to the operating system. It can inject logic signals into and sense logic signals within DPS racks at predefined M&DSS test points. Under the proper conditions, the M&DSS can control DPS equipment by means other than their normal interfaces. For example, an M&DSS instruction that places the proper pattern on IOC test points could cause an I/O operation to be performed. M&DSS instructions can originate from various sources, only one of which will be mentioned here: the M&DSS read-only core memory. M&DSS executes instructions from this source in response to one of three stimuli: manual intervention, failure of the CLC operating system to reset a *sanity timer*, or an explicit request from CLC software.

The SAFEGUARD data-processing system includes standby equipment. There is one extra processor, program store, variable store, IOC, and TG. A given peripheral device is either duplicated and wired to a particular IOC or switchable under program control to either IOC. Software can establish a *green partition* and an *amber partition* such that equipment in the green partition cannot communicate with equipment in the amber partition, and vice versa. The amber partition has two purposes. Spare equipment is partitioned amber, so it may be used as a pool of inactive equipment from which replacements for green units are drawn; for example, the operating system can substitute the amber IOC for the green IOC. When it contains sufficient equipment, the amber partition may function as an independent computer. The operating system then executes independently in each partition.

III. THE SUPPORT MODE AND THE PROCESS EXECUTE MODE

The SAFEGUARD data-processing system is used for three different activities with distinct requirements:

(*i*) Tactical execution of a user process.
(*ii*) Debugging of a user process.
(*iii*) Utility operations such as saving the contents of disc packs on magnetic tape.

The operating system reconciles conflicting requirements between these three environments by functioning in the *process execute mode* for item (*i*) or the *support mode* for items (*ii*) and (*iii*).

In the support mode, the CLC operating system reads requests from job control cards to invoke utility programs. Some of these programs allocate space on DPS disc volumes; others install load modules created on the support computer onto DPS disc. Still others temporarily or permanently patch load modules.

Debugging is easier in the support mode than in the process execute mode. In the process execute mode, program testing is hampered because manual interactions such as a teletype input cannot be exactly reproduced for each test run and because the cause of an error is difficult to determine when several processors have been executing simultaneously. In the support mode, on the other hand, the operating system allows simulated manual inputs to be generated as specified by a card deck, each card tagged with the time of day it is to be processed. Also in the support mode, the operating system allows all processors but one to be idled when a programmer-specified condition occurs. Only one user job can run at a time, although that job may use more than one processor. A more detailed discussion of the operating system support mode debugging capabilities appears in this volume.[3]

The second mode of the CLC operating system, the process execute mode, is discussed in depth in Sections V through IX.

IV. RECONFIGURATION, LOADING, AND DPS RECOVERY

Selection of either the support mode or the process execute mode is under the control of the CLC data-processing system operator at the time the system is initialized. The major events following a request for the process execute mode will now be examined.

First, the operating system attempts to identify faulty hardware, such as an IOC that appears unable to reference a particular variable store. Next, it establishes a green partition sufficiently large for the user process (by examining tables stored on disc along with the process), and partitions amber all equipment not needed. Finally, it loads the user process from disc, it enables the sanity timer, and execution begins.

The same sequence of events can also be initiated manually or automatically during execution when DPS sanity is in question, in which case it is called *DPS recovery*. The reason for this operation is discussed in Section IX.

Both manually initiated loading and DPS recovery involve the M&DSS. Each causes the M&DSS to execute a program that idles all processors, causes the IOC to load a portion of the operating system into memory, and restarts all processors. The remainder of the load or the recovery is performed by the operating system as described above.

V. THE PROCESS EXECUTE MODE

Two fundamental constraints are placed on the CLC operating system in the process execute mode:

(*i*) *Timing.* Certain user process computations are required as often as every 6.5 ms.

(*ii*) *Error Control.* The incidence of a hardware or software failure must not cause the operating system to lose control.

The following sections of this paper examine how the four operating system functions of processor management, main storage management, I/O management, and error recovery are performed as a consequence of these constraints.

VI. PROCESSOR MANAGEMENT

The problem of processor management is simply stated: How shall the CLC processors (as many as ten) be best utilized to perform the SAFEGUARD process control calculations within the real-time constraints imposed by system requirements? To provide the necessary throughput, the multiple processors must be permitted to perform certain calculations in parallel, but how shall this capability be provided to the programmer? Shall the programming language allow statements to be executed in parallel as in ALGOL 68? The answer is no. Parallelism is excluded from the language, and instead the operating system is allowed to execute simultaneously as many "independent" programs as possible, as in conventional multiprogramming systems.

Recognizing that it may be necessary to prevent one program from interfering with another through alteration of shared data, the operating system provides functions equivalent to Dijkstra's[4] P and V so that "independent" programs may cooperate, thus becoming no longer truly independent. Programmers can write parallel algorithms involving several programs. The operating system will assign programs (now called *tasks*) to processors so that as many processors as possible are busy. The assignment algorithm is sketched later.

The real-time constraint can be approached in two ways. "Time" often suggests "interval timer," the expiration of which usually causes a processor interrupt, followed by the initiation of the time-dependent computation. This method becomes decidedly unattractive if the time-dependent computation must be performed on more than one processor, for the operating system would have to decide which processors to interrupt, save the previous state of each, initiate new tasks on several processors, and later restore the processors to their original tasks. Therefore, this interrupt-driven approach is discarded in favor of a simpler method that is suggested by the following observation. Assume the time-dependent calculation must be completed within 6.5 ms from the time of request and further that it can be structured as P tasks each having an execution time T of less than 6.5 ms. Then if, among the tasks that are already running at the instant the time-dependent calculation is requested, at least P of them finish within $6.5 - T$ ms, sufficient processors will be available to complete the desired computation. By restricting task run times to the millisecond range, the desired behavior can be produced without timer interrupts because processors become free every few milliseconds.

If a computation cannot be completed in milliseconds, it is divided into pieces (tasks) that can be completed in the allotted time, and each task is executed in turn. This requires the operating system to recognize predecessor conditions, e.g., that Task B cannot run until Task A completes. It is useful to allow more complex situations, such as those represented by Fig. 1. Here, Task A is said to *enable* Tasks B, C, and D, and Task E cannot execute until *conditionally enabled* by both C and D. Enablement is a generalization of the "wake-up" operation of other operating systems.[5]

What conditionally enables Task A? It could be some other task not shown, or it could be the operating system. One particularly important feature of the operating system is that it can be requested to enable a

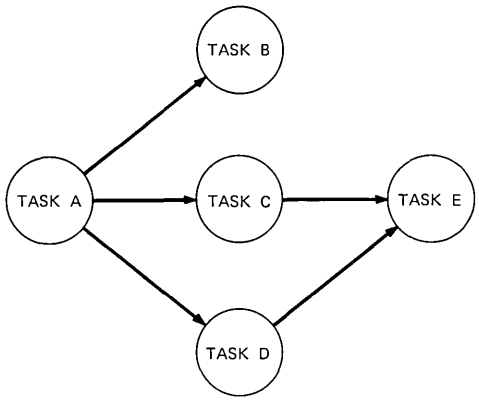


Fig. 1—Predecessor conditions among tasks.

given task approximately every 6.5 N ms ($N = 1, 2, 4, \cdots, 64$). Sets of tasks initiated this way are called *timed arrays*, structures that form the basis for almost all time-dependent computations performed in the process execute mode.

Assuming that each task is assigned a unique *priority* relative to all other tasks, the following algorithm decides which task will run next on a given processor:

(*i*) Of all the tasks whose predecessor conditions have been satisfied but which are not executing yet, execute the task of highest priority.

(*ii*) Allow each processor to perform step (*i*) independently of all other processors.

If each processor performs this operation whenever the task it is currently executing terminates, then no one processor is master over another, and the operating system is not sensitive to the number available. In fact, the number of processors can be increased or decreased during execution.

The way in which the operating system keeps track of the 6.5-ms intervals can now be explained. In Section II it was stated that the IOC can alter bits in memory, that an IOC operation can be initiated by a peripheral device, and that the timing generator may be programmed to signal the IOC at intervals of 6.5 ms. Let the IOC program "satisfy the predecessor conditions" of a task (i.e., set bits in an operating system table), and let this task be of high priority. The above algorithm then ensures that this task will execute as soon after the timing generator command as a task on any processor terminates. Although an exact 6.5-ms synchronism is not possible, the simplification of the operating system achieved by not using timer interrupts for this purpose outweighs the disadvantage of having to account for a slight timing jitter when real-time deadlines are being planned.

The previous paragraph implies that, in the process execute mode, the operating system is itself executed as a set of tasks. This is indeed the case. In fact, the processor management algorithm makes no distinction between operating system tasks and those of a user, nor are system tasks necessarily of higher priority. In this way, execution of the CLC operating system is distributed over all the processors and the loss of a processor simply results in its load being equally distributed among those that remain.

VII. MAIN STORAGE MANAGEMENT

The hardware design of the CLC processor restricts the main storage management that the operating system can easily perform. The design does not allow the creation of a virtual memory since program store and variable store are both referenced by absolute addresses embedded within machine instructions. For the same reason, code is not easily relocatable, and a main storage management technique that assigns the same program to different locations in memory at different times is not feasible. A static allocation for all main storage is therefore implied. With a minor exception for part of variable store, this is the case.

Since programs are placed in fixed locations in memory, it is desirable to make this assignment only once, prior to task execution. The Execution Preparation Facility,[1] executing on the support computer, performs this function, and the load module brought to the CLC is not relocatable. This implies that the CLC memory rack configuration assumed at link-edit time must be available when the load module is read into core, and it is the responsibility of the reconfiguration and loading function of the operating system, described in Section IV, to ensure this.

The operating system provides a limited overlay mechanism. Two or more programs in the load module may be bound to the same address, and one or the other read into core as desired. The operating system performs the disc transfer, but it is the responsibility of the user to request the operation explicitly and to keep track of the current contents of overlay areas. Data base overlays may also be performed.

The operating system provides up to ten pushdown stacks in variable store, one for each processor. The stacks are used in the ordinary way for passing subroutine parameters, saving return addresses, and providing local storage for subroutines. A processor's stack is initialized to empty whenever a new task begins.

The CLC operating system provides no other forms of dynamic memory allocation. All other variable store usage, like all program store usage, must be declared at compile time. The decision not to allocate variable store dynamically meant that the maximum amount of data to be passed between two tasks would have to be decided at design time. This decreased operating system overhead and ensured the existence of a data structure large enough to handle the specified traffic level.

VIII. I/O MANAGEMENT

The traditional I/O management functions of an operating system are I/O scheduling, buffering, I/O completion processing, reservation and allocation of devices, and protection of one user from another. But an operating system can also provide other services such as concealing differences between devices so that one device can be substituted for another or altering the appearance of the device so that it is easier to program. In any case, an operating system should ensure the reliable performance and efficient use of the peripheral devices.

The CLC operating system deals with two general categories of devices. The first set consists of the conventional devices and includes the magnetic tape transports, the disc drive units, the cathode ray tube displays, the teletypes, the card reader, and the printer, and the second consists of the special-purpose devices such as the radar subsystem and the missile subsystem. These latter units are considered first.

For the special-purpose peripherals, the CLC operating system is only concerned with I/O completion and reliable performance. In particular, device characteristics are not camouflaged, and no attempt is made by the operating system to ensure the efficient use of the unit. Buffering is generally limited to providing an input area for devices that send data to the IOC of their own accord, under hardware rather than software control. The operating system functions of I/O scheduling, reservation, allocation, and user protection for this class of peripherals are simple. There is only one on-line unit of each type, and the user must do everything himself. Finally, an attempt is made to ensure the reliable performance of each device by monitoring some error indications it can produce and informing the user if trouble is being reported. These reports deal generally with the IOC-peripheral interface; the user is responsible for sensing and responding to device-dependent error conditions. The operating system was designed this way because the users were not sure at the time of how they wanted to program the special-purpose devices.

While the operating system management philosophy for special-purpose peripheral devices is generally one of minimal intervention, its approach for conventional devices is almost the opposite. Emphasis is placed on I/O scheduling and reliability and, in some cases, on altering the appearance of the device to the user. For example, to increase disc drive efficiency, read and write requests received by the operating system are reordered to minimize access delays. To increase reliability, each disc write is performed to two units so that if a subsequent read on one unit fails, a duplicate copy is available. The magnetic tape transports are another example. In this case, the appearance of the device is altered so that the user sees a tape capable of recording at up to four times the hardware rate of an individual transport. This is accomplished by directing suitably buffered output not to a particular transport but to a pool of four, capitalizing on the ability of the IOC to overlap writes on as many as four transports. The designers of the operating system knew how the conventional peripherals would be used, so they were able to plan more sophisticated support for them.

Neither the conventional nor the special-purpose peripheral devices generate processor interrupts when they complete a request. Instead, every 6.5 ms the operating system tests whether any I/O has completed. It then notifies the user via the conditional enablement of a user task. Since processor management uses no interrupts, neither does I/O management.

In the process execute mode, the CLC operating system makes no attempt to conceal the differences between devices, and programs are usually device-dependent. For tactical execution, this is permissible, but in other circumstances, it is a handicap. This is discussed further in Section X.

IX. ERROR DETECTION AND RESPONSE

The operating system detects errors in many ways and provides both local and system responses to these errors, depending on the circumstances. Local error responses consider the frequency with which an error is reported. If the frequency exceeds a given threshold, then extensive corrective action is assumed to be required. For error conditions that are treated in this manner, the operating system may make a particular response before the threshold is reached, but a different response after it is exceeded. For example, before the threshold is reached, a device reporting errors may be reset; after it is exceeded, further use of the device may be prevented.

This latter action suggests a general technique called "severing." If a peripheral device or a software function is declared severed, the operating system rejects all future requests for that device or function. This procedure is applicable to a variety of error conditions; its intent is to minimize snowballing by preventing a second failure from occurring as a result of the first. In the face of many errors, severing produces a relatively gradual loss of operating system capabilities and is appropriate in situations in which the consequences of DPS recovery cannot be tolerated.

Some operating system functions, especially processor management and I/O management for the special-purpose devices, are never severed. These functions execute moderately elaborate error recovery code that attempts to prevent unrelated calls of the same type from failing.

System level responses are provided in but not initiated by the operating system. A more complete discussion of SAFEGUARD error control can be found in Ref. 6.

X. DEFICIENCIES OF THE OPERATING SYSTEM

A single mechanism for peripheral device substitution, a feature commonly found in general-pupose operating systems, is not in the CLC operating system. Initially, this was felt to be an unnecessary complication because the important peripherals, the special-purpose devices, cannot be mimicked by any other peripherals. Later, several operating-system designers needed particular instances of this capability, and each built his own version. Allowing commands to be read from the card reader rather than from a teletype (in the support mode) and permitting the use of one teletype in place of another (in the process execute mode) are both instances of peripheral device substitution, yet two different mechanisms were coded.

The operating system does not provide for communication between tasks, and it should. An extension of conditional enablement would be to allow a parameter list to be passed by each predecessor task. Communication between tasks does take place, but each programmer devises his own mechanism.

Whenever a particular subroutine was needed by one class of users, it was made part of the operating system and accessible to all users, thus penalizing those who did not require the subroutine by costing them core. A subroutine library established on the support computer would have avoided this.

XI. CONCLUSION

The CLC operating system is not intended to be general purpose and cannot easily be made so. Criteria that might be used to judge the adequacy of a general-purpose operating system do not apply to it, such as the ease of learning its job control language or the number of jobs it can process per hour. Since the real-time performance of the SAFEGUARD Data-Processing System depends not only on the CLC operating system but also on the user process, the operating system would have to be considered a failure no matter how elegant it was if the overall real-time performance of the DPS were not achieved. But since the required performance has been achieved, the CLC operating system can be termed a success.

The operating system's most innovative and greatest success is its approach to processor management. The approach taken provides a rapid response time without the conventional use of processor interrupts. It also sets a logical framework in which it is possible to design, code, and test real-time programs taking advantage of up to ten independent processors.

REFERENCES

1. R. R. Conners, "SAFEGUARD Data-Processing System: Support Software and Support Computers: An Overview," B.S.T.J., this issue, pp. S149–S160.
2. J. W. Olson, "SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control," B.S.T.J., this issue, pp. S41–S61.
3. A.K. Phillips, "SAFEGUARD Data-Processing System: Debugging a Real-Time Multiprocessor System," B.S.T.J., this issue, pp. S133–S145.
4. E. W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages*, New York: Academic Press, 1968, p. 68.
5. E. I. Organick, *The Multics System: An Examination of Its Structure*, Cambridge, Mass.: M.I.T. Press, 1972, pp. 275–281.
6. L. J. Gawron, "SAFEGUARD Data-Processing System: System Error Control," B.S.T.J., this issue, pp. S123–S131.

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Process Design in the Structure of Real-Time Software Systems

By W. S. DOYLE and J. R. GIBBONS

(Manuscript received January 3, 1975)

Process design, structuring the real-time program for the CLC, was one of the difficult aspects of SAFEGUARD software development. Initially, there were no significant guidelines or criteria. In the course of the project, basic process-design rules were developed and significant experience was acquired. Some techniques that emerged are the use of short-running, asynchronous tasks; overlays to minimize storage requirements; and multiple storing of programs to minimize processor queuing.

I. INTRODUCTION

Process design involves defining the characteristics, interrelationships, and organizational structure of the tasks that comprise the operating system and the applications software. It was one of the difficult aspects of SAFEGUARD software development. Initially, there were no specific criteria to be followed. Several iterations were required to converge on the final process design. The purpose of this paper is to present some of the basic guidelines that evolved in the course of the SAFEGUARD project. The guidelines included are those believed to be most workable and most applicable to a wide range of real-time software systems.

II. GENERAL PROCESS-DESIGN GUIDELINES

Major efforts in the process design involved selecting from among the available methods of enablement for tasks, selection of the time frames in which they would execute, and the definition of task priorities. (For a description of tasks and processor management, see Ref. 1.)

2.1 Task structure

Initial investigation of possible process structures led to the use of both synchronous (time-enabled) tasking and asynchronous (event-triggered) tasking. It was clear that critical processing had to be given high priority, and it was generally of a synchronous nature. Asynchronous tasks were to be used to fill the time slots between critical synchronous tasks and to provide a uniform distribution of processing among the available processors. This general approach had to be modified by a few additional considerations. First, low-priority asynchronous tasks must have a short run time or they will hold a processor too long, denying access to high-priority tasks. Second, it is generally more difficult to design and test a process which utilizes asynchronous tasks. Further, it is not always necessary to achieve a uniform work distribution, e.g., during the process initialization and termination sequence. An almost totally synchronous design was chosen for process initialization and termination tasks to facilitate design and testing.

It is inefficient to enable a synchronous task, only to find that the task has no data to process because a peripheral device has not completed its transfer or because other tasks have not generated it. Ultimately, synchronous tasks were utilized when critical and periodic response was required and when the availability of data at the same frequency as task enablement could be guaranteed.

The asynchronous, event-triggered task is enabled by the completion of an I/O transfer or by the successful completion of processing by a predecessor task or tasks. Each predecessor task can conditionally enable one or more successor tasks. A successor task is absolutely enabled, i.e., ready to run, only after all conditional enablement criteria have been satisfied. The predecessor-successor relationship of conditional enablement can also help alleviate data interference problems. Table I depicts some of the process-design questions that were faced and the type of tasks used to answer these questions.

Table I — Process design

Problem Description	Task Description
Support high-frequency, high-accuracy endoatmospheric target track.	Synchronous task whose frequency is at least as high as the update requirements.
Process intersite communications message traffic.	Asynchronous tasks whose trigger for enablement is the arrival of intersite communication messages.
Generate time-ordered, simulated radar replies during an exercise.	Both synchronous and asynchronous tasks. Tasks that generate the replies are synchronous. These tasks conditionally enable an asynchronous task which time-orders and outputs the simulated replies.

2.2 Parallel processing

There were several cases where identical processing had to be repeated for several items in a short time frame. In this case, the throughput requirement exceeded that of a single processor. The solution to the problem was to parallel process, i.e., to define several tasks executing identical code. Since the code was re-entrant, only one program copy was required even though each instance of the task could be separately controlled and separately enabled. Again, the structure of this processing could be synchronous, asynchronous, or a combination of both. It was found necessary to parallel process different types of tasks to take full advantage of the multiprocessor environment.

Obviously, multiple-instance task use may cause processor queuing problems. These can be alleviated by storing one program copy for each task. The critical consideration determining the number of program copies needed is the response requirement on the tasks involved.

2.3 Data interference

One of the primary design goals was to maximize throughput of the processing system. A natural implication of this was an attempt, in the beginning, to multiprocess everything. This immediately triggered task-to-task data-interference problems. Reviewing the task-response requirements made it obvious that not only was it not necessary to multiprocess all tasks, but in many instances it was impossible.

This observation led designers to take a closer look at task time-frame design and the serial-processing relationship among tasks. From these investigations evolved two basic task-design guidelines for avoiding data interference. If possible, competing tasks should be assigned to nonoverlapping time frames of possible execution.* If this could not be done, an attempt was made to establish predecessor-successor relationships among them. These techniques could be used only infrequently when tasks were competing for data.

Since a large number of data-interference problems were not solvable by either of these techniques, attention was directed to data-base design. Many interference problems arose when only two tasks were in competition, one loading the data and the other processing them. In those instances where the competing tasks were accessing a variable number of data items each time executed and the response requirements on the task were not critical, a circular queue with an access mechanism called a take-load pointer was used. With this mechanism, the loading task uses the load pointer to control the writing of data. It never

* A time frame is a time "window" in which a task is allowed to execute.

writes beyond the take pointer. The processing task uses the take pointer to control the reading of the data. It never takes beyond the load point. This technique alleviated about 10 percent of the interference problems.

When two high-frequency tasks with critical response-time requirements were competing for data, a double-buffering technique was useful to avoid data interference. In this case, two tasks both execute at a high frequency and in the same time frame. One loads the data and the other processes it. The competition question was solved by dividing the data area into two identical buffers, one of which was being loaded while the other was being unloaded. When unloading was complete, the buffers were switched. This technique works, but was of limited applicability.

As a final resort to solving interference problems, locking and unlocking conventions were used. These conventions required use of predefined program-logic sequences to lock and unlock data areas. These sequences relied on a special CLC instruction called a "biased fetch" which was implemented for this purpose. (For a more complete description, see Ref. 2.) Locking will always work, provided locking conventions are observed and enforced. Improper use of locking has caused the integration effort many headaches. The improper use of locks will manifest itself in a thousand disguises. However, it was necessary to use locking to solve more than half of the interference cases.

2.4 Discussion

How well is the process working? How close does the process conform to the process-design requirements? These are two questions that were constantly asked. To answer them, a process performance-monitoring capability was implemented. The implementation relied on constant monitoring of "probe" or test points within the process. Implantation of these probes into the process and interpretation of the resulting data proved useful for fine tuning the design and verifying that the basic requirements were being met. This should have been done much earlier in the design cycle. Probes should be capable of furnishing such data as routine and subroutine execution timing; the time differential between when a task is enabled and when it actually acquires a processor; minimum, maximum, and average task run times, etc.

This section would be incomplete without a few words about the position of the process designer. It became obvious that the process designer must participate in program design and integration. He must do this to guarantee that the program designers do not stray from the process-design requirements on program timing and interfaces. He must be part of the integration effort to ensure that the process design is actually implemented in the process. Furthermore, it was found that the process designer required this program design experience and integration experience to be able to accurately interpret performance data and to use it to refine the design of the process.

III. SYSTEM SIZING CRITERIA

Estimates of the number of processors, program stores, and variable stores needed to do the job were continually monitored in the light of the mission to be fulfilled by the system. System sizings were an iterative effort. As requirements solidified and understanding of them improved, as routine, subroutine, and data-base estimates improved, and as simulation tools for forecasting system loading improved, sizing estimates changed.

3.1 System operating points as design input

It was the process designers' responsibility to map system performance requirements into the number of instructions needed to code these requirements, the amount of variable store required to support the data base, and the number of processors needed to meet throughput requirements. The design effort attempted to balance, on a system cost basis, the inevitable trade-offs among these three resources.

To facilitate evaluation of the impact of the various trade-offs on process design, a contour or envelope of possible system operating points was developed. Points on this contour reflected maximum usage of one or more resources and/or maximum processing capability of one or more process functions. It soon became clear that there were not enough resources to support the "worst-case" condition for all process functions. Further, it was not only impossible to support the worst case, but not necessary, since all functions do not peak simultaneously. Once the contour was identified and a feasible and reasonable set of operating points selected from it, trade-offs could be thoroughly examined.

After the operating point was selected, it was the responsibility of the process designers to ensure that the design supported it. It was this effort that required the continual resizing of the system to guarantee that it would fit into the resources available.

3.2 Minimizing core requirements by the use of overlays

As design proceeded, program storage resources were rapidly exhausted. Further investigation showed that there were certain sets of programs that were not required to be in core simultaneously since their functions were mutually exclusive. Another set of programs had such "loose" timing requirements that they could be called in from a peripheral storage device prior to execution. Examples of such sets are hardware test programs, display update programs, and system initialization programs.

3.3 Load balancing

One of the most critical factors that influenced selection of the system operating point was the need to maintain a balance between the capability of the application process and the exercise process; that is, the exercise process must be capable of driving the application process at or above the system operating point.[3]

When planning for load balancing, two factors must be studied. These factors are the "immediate-response" processing requirements, representing a maximum allocation of resources applied for a short time, and the "long-term" or residual processing requirements, representing the load over a typical processing cycle.

Since the process had two basic time frames, one approximately 5 to 10 ms and one approximately 50 to 100 ms, two levels of load balancing were needed, short term and long term. Experience showed the most critical need for load balancing to be at the short-term level. It was also the most difficult to satisfy. Once the short-term problem was solved, the long-term problem disappeared. Short-term balancing was found to be extremely sensitive to changes in routine and subroutine execution times, and tuning the balance was always required.

IV. ALLOCATION OF RESOURCES

Consideration of possible process structures led to three basic alternatives for the allocation of the most critical system resources, processor and radar time. The first alternative is fixed allocation in which the execution time frame of each task is fixed in nonreal time by the process designer. The second alternative is real-time allocation in which the execution time frame of each task is determined dynamically by a synchronous allocation task included in the process. The third alternative is a combination of the previous two.

Initially, fixed allocation with its heavy reliance on synchronous tasking was favored because it appeared to be easier to design and test, and its reactions to traffic were easier to predict. After study, this design was rejected because it resulted in a nonuniform distribution of the work which, it was thought, would result in unacceptable system performance.

The second alternative to a process structure centered on attempting to allocate almost all resources in real time. This technique yields a much more uniform distribution of work among the processors and a better utilization of resources; however, designing and testing this type of process appeared to be very complex. In addition, it was decided that the uniformity of the distribution of work was not as critical as first thought.

Process design eventually included both types of allocation. This combination allowed the process to be designed and tested in a timely manner and yielded a nearly uniform distribution of work, giving reasonable processor utilization.

V. OVERLOAD RESPONSE REQUIREMENTS

SAFEGUARD process designers had to answer the question of what to do when there were more requests for service than could be accommodated. Because it was felt that the inherent overload handling of the priority tasking structure was not sufficient, a predefined, fixed-response technique was developed.

In this approach, a tunable processing load point was defined at which overload-response rules were invoked. The exact rule to be used depended on the outcome of an overload function which "predicted" processor usage for the next cycle. This prediction was done by summing selected system-traffic components weighted by an appropriate factor. Depending upon predicted processor usage, the execution of certain lower-priority tasks was curtailed. The higher the predicted usage, the more tasks were curtailed. Once the system entered overload, it remained there for the duration of the engagement.

This technique eliminated the additional testing and design required to implement a feedback type of overload response. The feedback technique was tried in the prototype system and was found to be impractical.

VI. MULTIPROCESSOR QUEUING PROBLEMS

Minimizing task run times was of critical importance for certain process functions; e.g., endoatmospheric tracking. Generally, functions with critical response times were also those functions selected for multiprocessing. This quickly led to a realization of the impact on task run time of processors queuing for instructions.

A decision had to be made either to use multiple copies of multiple-instance parallel tasks or to divide the program into subunits. The final decision was based on each task's response requirement. For example, in one instance five identical tasks executing from a single program copy ran 77 percent longer than single-processor run time. The same programs were suitably subdivided and partially distributed to five independently addressable storage units and run time was reduced to a level about 25 percent greater than single-processor run time. Of course, if five complete copies were stored in five different independently addressable storage units, there would be no increase in the parallel-tasking time versus single-processor execution. The final decision made was to use multiple program copies only for those tasks that always had to execute at maximum efficiency. This was done to conserve program storage. More commonly, large programs were divided into subunits distributed among program storage units in such a manner as to equalize the number of accesses per storage unit per time interval. This general technique was found to be sufficient for a large number of applications.

VII. SUMMARY

Initially, there were no significant guidelines to process design; these were developed as design progressed. No claim is made that the criteria which evolved in our design are exhaustive, but they should be applicable to a wide spectrum of real-time software systems.

It was good design practice to use short-running, low-priority, asynchronous tasks wherever possible. This helped alleviate task scheduler conflict problems, which arose when there were a large number of high-priority synchronous tasks. It helped guarantee that high-frequency, high-priority tasks would execute at their specified frequency, and it also aided in achieving a more uniform work distribution.

Data-interference problems arise naturally in a multiprocessing environment. The most useful technique to solve these problems was consistent use of software locking conventions; however, improper implementation of these techniques caused problems during integration.

To minimize system overhead and to avoid wasting processing time, tasks should be enabled only when they have work to do. Synchronous tasking should be used only if data are available to be processed at the same frequency as the enablement.

Since it was essential to maintain a balance of capabilities between the application process and the exercise process, it was required that the interfaces between these processes be established as soon as possible and that their integrity be rigidly maintained.

Because it was necessary to measure how well the process was working, it was found that performance probes should be included in the initial design and considerable thought should be given to their correct placement. Performance probes proved invaluable throughout the system-integration process, particularly in helping to identify task-timing and queuing problems. Resolution of these problems requires that the process designer become deeply involved in the test-and-integration effort.

Finally, process design is iterative. For this reason, it is important that the design be kept as simple and straightforward as possible. This standard guideline of programming is even more important in process design because of the inherent complexity of the multiprocessing environment.

REFERENCES

1. J. P. Haggerty, "SAFEGUARD Data-Processing System: Central Logic and Control Operating System," B.S.T.J., this issue, pp. S89–S99.
2. J. W. Olson, "SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control," B.S.T.J., this issue, pp. S41–S61.
3. B. P. Donohue III and J. F. McDonald, "SAFEGUARD Data-Processing System: Process-System Testing and the System Exerciser," B.S.T.J., this issue, pp. S111–S122.

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Process-System Testing and the System Exerciser

By B. P. DONOHUE III and J. F. McDONALD

(Manuscript received January 3, 1975)

This paper considers two problems: how to build the SAFEGUARD software so that it is testable and how to test it as realistically as possible. The first is solved by an iterative process of adding software capabilities, testing them, then adding more. The second problem is solved by driving SAFEGUARD with computer-generated radar echoes.

I. INTRODUCTION

Testing activities play a crucial role in the development of all hardware/software systems. These activities are described in terms of two phases, system integration and system testing. The system integration phase is carried out through tests which determine that all components of the system have been properly connected and are performing their specific function correctly. During the system test phase, the *performance* of the overall system is determined through analysis of the results obtained from some finite set of tests. The tests must reflect, as well as possible, the environment and full range of permissible data and control inputs. Although these phases overlap extensively, much system integration occurs before the system test phase.

It is well known that very difficult problems may be encountered in the system integration and test phases of complex system development programs. The plans and some of the significant techniques used to minimize these difficulties for the SAFEGUARD development are discussed.

Plans for the full SAFEGUARD system tests required large-scale analysis and simulation of the complete system. Since it is not possible to describe all the considerations that went into this planning, discussion is limited to a general description of overall system test planning. However, the relationship between the overall system tests and the data-processing effort are described specifically. Particular attention is given to the system exerciser because of the important role it plays.

II. SYSTEM INTEGRATION AND TEST PLAN

For several reasons, it is vital to prepare a detailed system integration and test plan. First, the time allocated for conducting the integration and test phases is usually not sufficient to demonstrate system performance under all conditions. This is simply an empirical observation. It could be attributed to the lack of detailed understanding of the objectives at the time the overall system development schedules are being formulated. It is always possible to conceive of an infinite number of tests of any complex system. No matter how carefully planned, the number of necessary tests will still be very large and, therefore, require a significant amount of calendar time to conduct. Since the system integration and test phases are the last activities before making the system available to the user, there is always pressure to make these periods as short as possible. The early existence of a detailed test plan is important because it provides strong support in arguing for reasonable system integration and test intervals and allows optimal use to be made of the allotted time.

Second, the system integration and test phases can overlap and, therefore, interact extensively. The tests that are conducted during the integration phase are designed to verify that system components perform as specified. Results from these tests can serve to increase confidence in overall system performance. The scope of future testing can be significantly influenced by this increased confidence. As a result, the testing activities in these two phases should be well coordinated.

Third, there are always schedule difficulties during the system integration and test phases if planning for test tools, techniques, and procedures does not begin long before the actual test period. Development of the hardware/software products can be influenced by test considerations. The test tools can often be developed more economically, and will better serve needs if identified early. Preparation of a detailed plan is the best way to recognize required lead times and avoid such scheduling difficulties.

Fourth, monitoring of progress is particularly difficult during these phases of the development. It is not uncommon to find that progress has been negative (and unknown) during parts of these intervals. A detailed test plan can serve as a very good measuring guide to monitor this progress.

Some general characteristics of a good system integration and test plan are reasonably clear. It identifies the means to achieve a specific set of objectives in a specific time, it recognizes the availability and capability of other tests carried out during the development, and it reflects all appropriate constraints on the use of resources. In Safeguard, certain features of the plan were more significant than others. Four have been selected for more detailed discussion.

2.1 The incremental approach

Everyone recognizes that a complex system cannot be integrated in one step, so an "incremental approach" must be used; i.e., the complexity of the hardware environment, the software, and the test cases must be built up incrementally.

Several factors were considered in arriving at the specific incremental approach for Safeguard. These led to a series of steps of increasing complexity, where each step included a given level of hardware, software, and functional tests. The principal steps were:

(*i*) Integrate all the "control" software; i.e., demonstrate the basic operating control necessary to perform initialization and cycling.
(*ii*) Integrate those software units that are part of critical timing chains.
(*iii*) Integrate additional software, which allows a simple, but consistent, stream of functional processing.
(*iv*) Interface this software with hardware; e.g., radars.
(*v*) Integrate remaining software to provide complete capability.

These principal tests were supplemented with additional parallel testing of various parts of software. Following is a brief description of how these steps were applied to the Missile Direction Center (MDC) application software.

First, the basic control programs were merged with the operating system, and the ability to load, initialize, and cycle was established. Then software dealing with the radar loop was added; i.e., radar management, search, and track programs. Ability to search and track was then established at low traffic levels, while the radar hardware was simulated with software. After sanity was established in the software, the radar hardware was introduced into the testing loop. In parallel with this activity, application programs supporting intersite communications and command and control were tested in a separate test bed. Similarly, both battle planning and missile guidance software were tested in separate software environments. Ultimately, these programs were merged into a single process, and the complexity of the test cases was systematically increased.

The incremental approach can create difficulties. It is obvious that some mechanism must be provided to represent interfaces of programs that are not yet a part of the process. Dummy programs, called "stubs," were provided. The requirements for stubs depend on the nature of the programs they represent and the sequence in which programs are added and tested. If this aspect of the incremental approach is not carefully considered during test planning, the stubs may become nearly as complex as the programs themselves, thus defeating the incremental strategy.

The selection of test cases can affect the efficiency of a test plan in a major way. SAFEGUARD has literally hundreds of individual capabilities and operates over a continuum of threat environments. Each test was carefully designed, using a design-of-experiments approach, so that all capabilities covering the full range of operation could be verified with the smallest number of tests. The test design was also approached from an incremental viewpoint, and was found to require an iterative effort.

The sequence used in identifying the test cases for full system testing of the SAFEGUARD MDC is briefly described here.

(*i*) The peak traffic level to be verified in full system testing was selected.
(*ii*) The types of threats to be countered, and allowable combinations, were delineated.
(*iii*) A sequence of tests starting with a single target and building up to peak traffic was identified. The "single target" was common to all test cases, as were other targets added later. Keeping pieces of the threat environment common provided a basis of test result comparisons—peg points along the way.
(*iv*) A set of high-traffic test cases was defined and all capabilities tested were identified. This exercise was performed iteratively with the goal of identifying a *minimum* set of high-traffic tests that, as a collection, test all system capabilities and cover all necessary threat mixes.

2.2 Success criteria

The system integration and test phases are intended to demonstrate that the various components and the system operate as intended. Tests are designed to subject the system to various stresses and conditions. The crux of test design is the clear specification of criteria that can be used to measure successful operation. It is obvious that this has to be done, but it is not always recognized that the success criteria will affect a test program in so many ways. For example, the efficiency of the test activities is vastly improved if the success criteria, that is, expected results, are available before the execution of the test. The criteria can also affect the data recording and reduction efforts. Since the specification of success criteria is a form of testing, it is not uncommon to uncover problems in either requirements or implementation. All these factors recommend that success criteria be identified early in the development sequence.

This effort was both difficult and large. On the SAFEGUARD project, sources of information that provided a basis for establishing success criteria included results of the test program conducted at Meck Island, desk analysis, and simulations. The greatest amount of data came from the simulations of the system. Various portions of SAFEGUARD were simulated in varying degrees of detail. These simulations were in turn calibrated using analytical and field data results. Where possible, the simulations were then used to predict system performance for each test case. The success of a test was measured by comparing data recorded during the test to predicted values. The simulations were large, initiated early, and served as a basis for system evaluation activities.

2.3 Data recording and reduction

One critical step in testing a system is measuring the system's performance. The basic measurement tool in the SAFEGUARD project was the recording and reduction of test data. Because of the complexity of the software processes and the tightness of schedules and on-line computer time, the ability to process recorded data off-line was essential. Recording and data reduction were not treated as two problems, but rather as two aspects of the same problem.[1] A coordinated approach to recording and data reduction was taken to achieve an efficient solution.

In "high-traffic" testing, or in any mode of testing, in fact, recording should be minimized (e.g., so that the off-line data reduction system is not overwhelmed with data). To meet this goal and still preserve the necessary error isolation capabilities, a "hierarchy" of recording selectability was defined.

The basic approach to recording and data reduction for SAFEGUARD was to construct each process so that the ability to select the desired mix of recording per run or per test could be accomplished with ease. Each process has the necessary capability for all possible recording permanently embedded in the on-line code. Data reduction program activities of sorting and formatting are minimized by the real-time association of sort "handles" with the recorded data. The key to the approach lies in a hierarchical structure in which multiple levels of recording are established. In general, three levels (high, intermediate, and low) are sufficient, although additional levels could be used in special instances.

The basic three levels can be described as follows. A process is divided into process functions. The recording necessary to isolate a test failure to a process function or to a peripheral is the highest level of recording. In general, these highest level data should consist of "counts" or statistics usable to determine logic flow, basic time sequencing, etc. The lowest level of recording consists of a detailed record of the processing of an input by a process function on a single logical pass. The intermediate level of recording is designed to aid the tester in selecting the proper low-level recording options.

A quick-look on-line computer capability was embedded in the software to allow off-line data reduction to be bypassed on occasion. This allows critical data to be "recorded" in on-line memory and output on a printer immediately following test completion. The test teams used quick-look and operating-system debugging aids[2] to support integration. Using quick-look, they determined when and in what portion of the process detailed recording should be performed. In the case of system tests, the system test specification specifies success criteria and prescribes the data to be recorded.

In testing the Meck prototype system, there were several examples of missions in which millions of words of data were recorded. In conducting a test involving missile launches, it is necessary to record all data of any possible interest, for the cost of repeating such tests is extremely high. However, the cost of repeating a test is reasonably economical in the TSCS (Tactical Software Control Site) since no launches are involved. Although tests are not absolutely repeatable, they are essentially repeatable in a functional sense. This means that a hierarchy of recording can be utilized to minimize the data recorded in real time, minimizing the off-line data reduction required. If a test fails, it can be repeated with selective recording performed in the suspect areas of the system. Although this approach forfeits some capability to isolate transient errors, it allows trade-offs to be made in the use of on-line computer time vs off-line data reduction time. With hierarchical recording, better test turnaround and lower overall integration costs were achieved without any serious problem in isolating transient errors.

2.4 Test tools

The need to provide test signals and data to "drive" any system is clear. As the complexity of the system and its operating environment increases, so does the complexity of the driver. It was considered vital to devote considerable resources to the development of a driver, and the effort was started early.

Few ground rules were available to guide its development. As it evolved, both special-purpose hardware and software were required. Because this effort was viewed as one of the more significant ones, the driver, or the SAFEGUARD system exerciser, is discussed in detail in the following section.

III. THE SYSTEM EXERCISER

The primary role of the system exerciser is to support test and integration of SAFEGUARD applications software in the hardware environment in which it was designed to operate. But testing SAFEGUARD against a simulator is difficult for two reasons. First, SAFEGUARD is a complex system involving radars, missiles, and interacting sites; the number of combinations of inputs is immense. Second, in actual operation, some inputs, such as radar noise, are random variables; these inputs should be random during testing as well.

Because of its complexity, it was not feasible to simply assemble the entire system and drive it utilizing the system exerciser. The system was assembled in an incremental sequence. The development of the system exerciser was, likewise, modular in nature. At each building stage, portions of the system exerciser's capability were used to drive that portion of the system included in the test bed. By relating the sequence of capability buildup in testing to the modularity of the system, an efficient development plan was evolved.

During the early stages of SAFEGUARD development, several goals for the system exerciser were established consistent with the primary role. The five most important goals are:

(*i*) As much of the system, hardware and software, should be exercised as is cost-effective. The software heavily interacts with the hardware; hence, confidence in the software/hardware combination can only be established through successful demonstration of their interactions.

(*ii*) The impact of system exerciser implementation on the application-system implementation should be kept to a minimum.

(*iii*) The system exerciser's simulation of the environment should be as realistic as is feasible.

(*iv*) The traffic capacity of the system exerciser should exceed the design level of the application system.

(*v*) The system exerciser should provide the capability to record the outputs of the application system.

During the development, every effort was made to retain sufficient flexibility to allow the system exerciser to be used in other ways, e.g., determining in part the system readiness and verification in an operational time period.

The discussions that follow apply to the MDC and PAR system exercisers. The approach taken for the BMDC exercise was different because of its distinct processing function (control and display) and relatively small size. The MDC system exerciser is the most complex.

3.1 Structure of the MDC system exerciser

Figure 1 shows the normal connections between equipment at an MDC site. During a system exercise, these connections are rearranged under software control as shown in Fig. 2. Data sent by the application data processor to the radar, the missile ground equipment, and other sites are directed instead to the exercise data processor. The system exerciser generates plausible radar returns, missile responses, and messages from other sites, and returns these to the application data processor. The exerciser is separated from the system being tested; it operates in a separate data processor connected to the application data processor through a special digital hardware unit, the Exercise Control Unit (ECU).

Tapes containing target and some environmental data to be used in the simulation are prepared off-line in nonreal time by a program called the SAFEGUARD Threat Action Generator (STAG). The design of STAG and the real-time processes was closely coordinated.

Several decisions were made in the design of the MDC system exerciser. First and foremost, the exerciser software was executed in a data processor distinct from the application data processor. The execution of exerciser programs in no way interferes with the execution of application programs. The alternative of executing the exerciser programs in real time on the application computer had been taken in the pro-

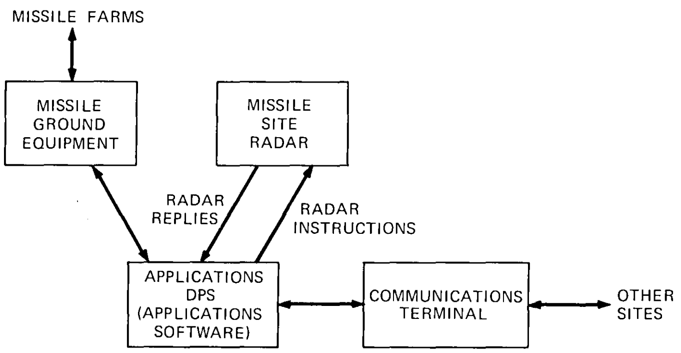
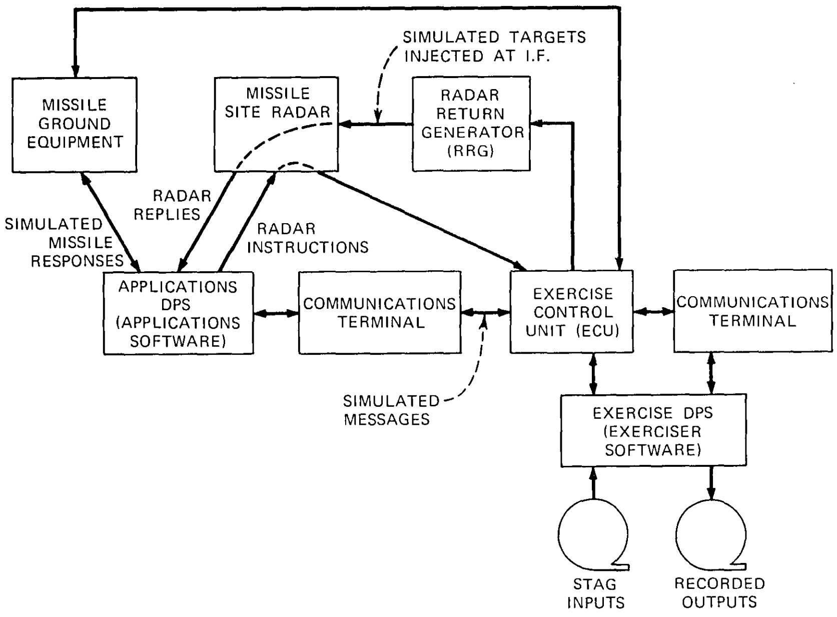


Fig. 1—SAFEGUARD MDC site equipment configuration.



MISSILE GROUND EQUIPMENT
MISSILE SITE RADAR
SIMULATED TARGETS INJECTED AT I.F.
RADAR RETURN GENERATOR (RRG)
RADAR REPLIES
SIMULATED MISSILE RESPONSES
RADAR INSTRUCTIONS
APPLICATIONS DPS (APPLICATIONS SOFTWARE)
COMMUNICATIONS TERMINAL
EXERCISE CONTROL UNIT (ECU)
COMMUNICATIONS TERMINAL
SIMULATED MESSAGES
EXERCISE DPS (EXERCISER SOFTWARE)
STAG INPUTS
RECORDED OUTPUTS

Fig. 2—Safeguard msr site configured for a local exercise.

totype system. Separation of the application and exerciser program systems also allows the development of the exerciser to remain as independent of the application system as possible. The potential for the exerciser programs to corrupt the application programs while operating in a combined form was demonstrated on occasion with the Meck test system.

Experience with the separated application and exercise systems has been favorable. No interference or identifiable differences in queuing or timing between the exercise and application modes was found. For instance, exercises were conducted that involved the tracking of "simulated" satellites. The performance of the application process was comparable when similar "live" satellites were tracked.

At one stage of the design, it was recognized that requirements for exerciser data processing throughput could be reduced by about 40 percent if the exerciser's load could be made more uniform. All that this required was to have the application program distribute in time the data which the application data processor sends to the radar (see Fig. 2). Changes were made to accomplish this without affecting the capability of the application system. Other examples include the setting of "flags" by the application program in data that it sends to the radar. When the exercise intercepts the data, it uses the flags to help expedite processing. This was accomplished without compromising either the applications or exerciser roles.

A second decision made in the design of the exerciser was to utilize as much of the hardware in an exercise as possible. Clearly, the real defensive missiles could not be included, but we note that the exerciser interfaces with the system of missile ground equipment, not just at the software/hardware interface. The full radar could not be included because a real target environment is not available to be viewed and because the cost of injecting simulated signals at the radar face is prohibitive. As shown in Fig. 2, the ECU injects simulated signals into the radar at the IF strip. This has allowed the applications software to be tested with major portions of the radar. This proved to be an effective approach from several points of view. It provided a mechanism to identify numerous problems in the hardware and software at the TSCS (the test bed). These problems included radar instruction sequencing errors, tracking bias errors, miswiring, etc. Corrections were made to both TSCS and site hardware. Software was corrected before it was shipped to the site. As a result of the prior testing at the TSCS, relatively few problems were found with the testing at site. Problems that were found were largely attributed to the detailed characteristics of the hardware not included in the exercise. The number of problems was lower than originally expected.

A third decision in building the system exerciser was to perform as much of the calculation required for simulation as possible before conducting the real-time exercise. Calculations for targets, defensive missile farms, and other sites and of hardware was done off-line, in the STAG facility; and results were placed on tape. The real-time software modified these data as appropriate for the real-time condition. This approach minimized the size and complexity of the real-time exerciser on a nonreal-time, pre-exercise basis. It also allowed programs such as trajectory generators to be used to support exercises for different radars; i.e., both the PAR and the MSR. This reduced the total size of the effort.

Fourth, in designing the exerciser, a number of decisions were made relative to the realism of the various exercise simulations. The approach was usually, but not always, to simulate the effect of a particular phenomenon, rather than the phenomenon itself. For example, in simulating the stream of intersite messages the MDC receives from the PAR, there were several options. The highest degree of realism would be a detailed simulation of the PAR system interacting with the threat environment. A much cheaper option would be to generate a representative sequence of intersite messages per threat. These threat messages would then be combined with a set of PAR status messages and modified in real time as appropriate. For SAFEGUARD, the latter approach was taken because it was economical, yet sufficient.

3.2 Exercising the exerciser

The system exerciser is a complex system, although considerably smaller than the applications system. As the principal tool in integrating the applications software, it had to be stable and reasonably debugged. There were at least two alternatives to test it. On one hand, the testing of the exerciser could be performed in conjunction with the testing of the applications system. On the other hand, the system exerciser could be tested as a stand-alone system. The latter approach was taken for SAFEGUARD, because it allowed greater control and easier isolation of problems.

Testing the exerciser was conceptually simple. We can view the applications software as outputting radar instructions, missile instructions via the missile ground equipment, and intersite messages. Those three classes of outputs represent the stimuli to which the exerciser responds. To test the exerciser, a simple software package called the Exercise Standard Test Process (ESTP), which resided in the application data processor and output these stimuli, was generated.

In simplest terms, ESTP obtains time-tagged data blocks containing radar instructions, missile instructions, and intersite messages from a driver tape. ESTP outputs each data block at the appropriate time. The key part of all this, of course, is the generation of the driver tape.

The most critical output from the applications software to the real-time exerciser is the stream of radar instructions. The real-time exerciser must determine whether or not any tactically ordered radar operations will cause the simulated radar to view any simulated targets. To test this portion of the exerciser, a stream of radar instructions that cause the exerciser to perform its simulation calculations is required. The target trajectories are known, and the expected response of the applications system is known. With this information, the radar instructions to be generated by the applications system are computed. ESTP assumes a "perfect" tracker but does not simulate the application system tracker. With respect to the missile loops and the intersite loops, similar deterministic test methods were used to exercise the exerciser.

Because of the testing done with ESTP, relatively few problems were experienced with the exerciser when it was interfaced with the applications software. Just as importantly, ESTP provided a vehicle for further isolation and debugging of problems that did occur.

IV. CONCLUSIONS

Some lessons learned from SAFEGUARD system integration and test activities can possibly be applied to other projects. They are summarized as follows:

(*i*) Prepare a test plan early; even though it cannot be complete initially, it should address those items that could affect design, or require long lead time.

(*ii*) Consider an incremental approach to testing. Several iterations will be required to decide what form the incremental buildup should take. Details will affect the program development schedules.

(*iii*) Start the identification of tests early. Don't delay the specification of success criteria. This specification requires lead time and coordination with other activities and can go a long way toward getting design problems resolved early. Make every attempt to minimize the total number of test cases. The expense of doing the necessary analysis, test specification preparation, etc., is large and often underestimated.

(*iv*) Make adequate provisions for an exerciser. Consider separating but not isolating the exerciser from the applications system. Try to incorporate as much of the hardware in the exercise configuration as possible. Test the exerciser to create a stable base for system testing.

REFERENCES

1. E. S. Hoover and R. A. Jacoby, "SAFEGUARD Data-Processing System: The Data Reduction System," B.S.T.J., this issue, pp. S181–S189.
2. A. K. Phillips, "SAFEGUARD Data-Processing System: Debugging a Real-Time Multiprocessor System," B.S.T.J., this issue, pp. S133–S145.

Copyright © 1975 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

System Error Control

By L. J. GAWRON

(Manuscript received January 3, 1975)

Errors occur even in well-designed, well-tested systems. This paper describes how errors are detected and controlled in the SAFEGUARD system and makes recommendations pertaining to the design of error control in large-scale, real-time control systems.

I. INTRODUCTION

SAFEGUARD is a *fault-tolerant* system. It can perform its tactical function even in the presence of many types of errors, including latent design errors, hardware failures, and operator mistakes. This paper describes some of the automatic error-control features of a generic SAFEGUARD Data-Processing System (DPS) and also the important role of manual control in maintaining the operational integrity of the DPS.

II. AVAILABILITY-RELIABILITY REQUIREMENTS

What are the availability and reliability requirements of the SAFEGUARD system? How are they satisfied? What is the role of error control?

As it pertains to SAFEGUARD, availability is the probability that the system is capable of performing its tactical functions—surveillance, tracking, intercept, etc.—at any given point in time. Reliability is the conditional probability that the system will function through the duration of a missile attack provided that the system is available at the beginning of that attack. The product of availability times reliability is required to be high to provide adequate assurance that the system can, at any time, quickly detect a missile attack and successfully defend against it. During peacetime operation, the emphasis is on availability so that the system can perform continuous surveillance and be ready at all times to wage battle against offensive missiles. During a battle, the emphasis is on reliable operation which includes avoiding significant interruption of tactical performance for any reason, even in response to errors.

Availability and reliability are both enhanced through the use of highly reliable, individual, hardware and software components, as well as through the use of inherently fault-tolerant hardware and software systems. For example, the DPS hardware design features extensive component redundancy and multiprocessor control. (The availability and reliability advantages of multiprocessor computers are commonly accepted today.[1]) The software design also has many features that minimize its vulnerability to errors. For example, it has decentralized system control. This means that total control is not contained in any single, and thus highly vulnerable, software module. It has distributed software execution control, i.e., all processors are treated equally. There is no single controlling processor, which would have an inherently greater vulnerability to errors. Also, the software makes minimal use of particularly vulnerable data structures such as linked lists. In addition to the use of highly reliable components and a fault-tolerant design, thorough testing is also performed to ensure that all components, as well as the total system itself, function as intended.* Thus, error prevention is one of the principal means of satisfying the availability-reliability requirements of the system. The other is error control.

Error control enhances system availability by aiding in rapid detection and replacement of faulty components. The DPS contains redundant components and, in conjunction with the software, it is self-diagnosing. The DPS is normally configured into two distinct partitions: one, called the green partition, is the primary computer system; the other, called the amber partition, is a secondary computer system containing the redundant units. When a faulty green partition unit is detected, a reorganization or reconfiguration of the DPS may be initiated either by the DPS itself or manually by a DPS operator in order to replace the faulty unit with its redundant counterpart. However, such replacements generally require interruption of tactical performance for several seconds.

Error control also enhances reliability by confining errors to minimize their effect on tactical performance, and thus minimize the need for such replacements during a battle. The remainder of this paper describes in greater detail how error control helps to satisfy SAFEGUARD's availability-reliability requirements, especially as they apply to the DPS.

* Software-debugging and system-testing methods are described in Refs. 2 and 3.

III. SYSTEM ERROR-CONTROL STRUCTURE

How are errors detected in the SAFEGUARD system? How are the effects of errors confined? How does the system recover from errors? This section discusses the general approach to solving these problems. The following two sections describe in more detail the two principal aspects of error control, namely error detection and error response.

Figure 1 illustrates the basic system error-control structure. Errors may be detected by hardware, by software, or by the DPS operators. Software detections include hardware-reported errors. Likewise, manual detections include both hardware- and software-reported errors.

Software provides the principal responses to hardware and software errors. There are two principal classes of error responses: local responses and system responses. Local responses are attempts to confine or correct errors at the point of detection. System responses replace faulty hardware or software components and restore basic system

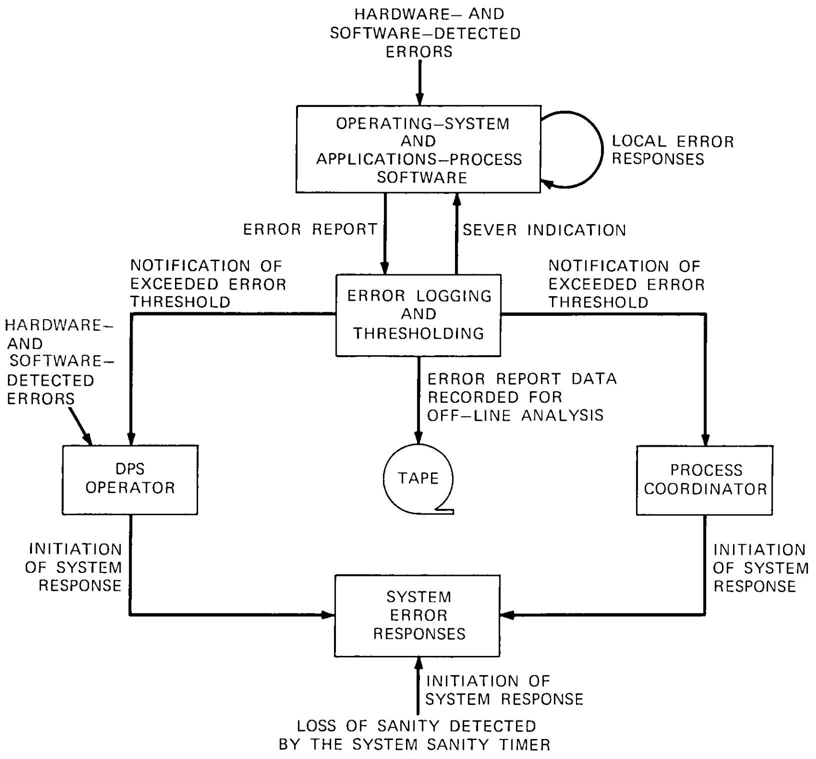

HARDWARE– AND SOFTWARE–DETECTED ERRORS
OPERATING–SYSTEM AND APPLICATIONS–PROCESS SOFTWARE
LOCAL ERROR RESPONSES
ERROR REPORT
SEVER INDICATION
NOTIFICATION OF EXCEEDED ERROR THRESHOLD
ERROR LOGGING AND THRESHOLDING
NOTIFICATION OF EXCEEDED ERROR THRESHOLD
HARDWARE– AND SOFTWARE– DETECTED ERRORS
ERROR REPORT DATA RECORDED FOR OFF–LINE ANALYSIS
DPS OPERATOR
TAPE
PROCESS COORDINATOR
INITIATION OF SYSTEM RESPONSE
INITIATION OF SYSTEM RESPONSE
SYSTEM ERROR RESPONSES
INITIATION OF SYSTEM RESPONSE
LOSS OF SANITY DETECTED BY THE SYSTEM SANITY TIMER

Fig. 1—System error-control structure.

sanity. System responses generally require a brief (several-second) interruption of tactical operation.

During normal peacetime operation, both local and system responses contribute to system availability by correcting errors and replacing faulty components. During battle-mode operation, the emphasis is on local responses to assure reliable operation by confining and correcting errors and to avoid the need to interrupt tactical operation for the purpose of performing system responses.

Specific local responses depend on the type of error detected. Several examples of such responses are described in Section 5.1. In addition to any specific response that might be performed, one common local response is to report the error to a centralized error logging and thresholding function. This function logs (records) the error-report data onto tape for use in off-line error analysis. It also keeps a record of error occurrences. If a report causes an error count or an error rate for the associated class of errors to exceed a prespecified threshold, then several additional common local responses may be taken. One such response is to return a sever indication to the program that reported the error. Severing is a method by which a program is permitted to degrade the operation of certain noncritical parts of the SAFEGUARD system by simply removing them from service. Its purpose is to avoid recurrence of errors. Typical components that could be severed are operating-system modules, such as data recording, or certain CLC peripherals such as printers, tape units, TTYs, etc. In addition to severing, another common local response to an exceeded error threshold is to notify a DPS operator and/or the highest-level software-control function called the process coordinator.* Either may then initiate a system response.

In general, system error responses may be invoked manually, by the process coordinator, or by a special hardware device called the system sanity timer. (Use of the sanity timer is described in Section 4.1.) System responses involve reinitializing the software and/or reconfiguring the DPS to remove faulty components. One of the principal system responses is DPS recovery which includes both DPS reconfiguration and software reinitialization. System error responses are discussed in greater detail in Section 5.2.

IV. ERROR DETECTION

4.1 Hardware detection

Error-detection circuitry is an integral part of the DPS. For example, the processors detect errors such as arithmetic overflow or attempts

* The entire collection of operating system and application software that execute on a single CLC partition is called a process.

to store data into nonexistent memory locations. When such errors are detected, a processor interrupt is generated and the processor transfers execution control, via the operating system, to the program's local-level interrupt-response code. Peripherals detect various types of input/output (I/O) errors, e.g., data-transfer parity errors. Such errors are reported to the software via I/O status returns.

In addition to the error-detection logic, which is a part of basic circuit design, the DPS also contains hardware devices specifically designed to aid in error detection. One such device is the CLC's status unit. It reflects the hardware status of each processor, memory rack, and peripheral, as well as of the radar and missile equipment. This status information obtained from the hardware is accessible to the software and displayed to the operators. Typical status-unit indicators are "processor disabled," "tape unit power marginal," "missile equipment internal error," etc.

Another special error-detection device is the Maintenance and Diagnostic Subsystem (M&DSS) sanity timer. This timer must be reset by the operating system's task scheduler every 50 ± 10 ms as an indication of basic system sanity, i.e., that the software is still executing on the CLC. If the operating system fails to reset it within the correct time interval, the sanity timer will automatically initiate DPS recovery.

4.2 Software detection

Just as error-detection circuitry is an integral part of the hardware, error-detection code is an integral part of the software. For example, the operating system performs input-validity checks on call parameters and the weapons process performs data-reasonableness checks on important data such as radar return signals.

The software also performs several types of hardware diagnostic tests. The operating system performs diagnostics on the DPS equipment; the weapons process performs diagnostics on the radar and missile equipment. For example, whenever the operating system reconfigures the DPS, it performs normal path diagnostics to verify that each green-partition CLC unit functions properly. Also, during tactical execution, CLC units and peripherals in both partitions undergo additional tests. For example, the operating system contains programs called real-time exercisers which test each green-partition memory rack every five minutes. They compare the entire program-store contents with a program-store image on disc to verify that no programs have been modified. They "read test" each variable store rack in its entirety, and they "write test" the first two words and the last two words of each variable store rack by storing test-pattern data into these words and then fetching the words to verify their contents. These four words in each variable store rack are reserved for this testing purpose. The weapons process contains continuously running radar tests that verify the basic functional operation of the radars. It also contains manually invokable radar tests and missile tests, which are more extensive diagnostics and which are used when faults are suspected in this equipment.

Extensive M&DSS diagnostics, capable of isolating faults to the chassis level, are also performed on amber CLC units and peripherals. All DPS units are periodically reconfigured out of the green partition (replaced by their redundant counterparts) in order to undergo such testing in the amber partition. The purpose of these tests is to minimize the probability of failure in green-partition units by detecting potentially faulty units before they actually fail. M&DSS tests are scheduled by the CLC operating system and are initiated manually. Processors may be reconfigured without terminating execution and are scheduled for M&DSS testing hourly. Other CLC units and the I/O subsystem require an interruption of tactical execution in order to be reconfigured. The entire I/O subsystem is scheduled for M&DSS testing every four hours. CLC units other than processors are not automatically scheduled for M&DSS testing; however, such tests may be initiated on those units manually at any time.[4]

In addition to hardware diagnostic tests, a system exerciser[3] is used to periodically test much of the total hardware/software system.

4.3 Hardware- and software-reported errors

The hardware and the software report many of the errors they detect to the DPS operators. For example, the operators' consoles have many hardware- and software-controlled error-indicator lamps. A system-status panel displays much of the information in the CLC's status unit, thus indicating the operational status (working, faulted, off-line, etc.) of the CLC units and peripherals. Software also notifies the operators of exceeded error thresholds via error-report messages. With the wide variety of error-status information available to him, a DPS operator often better comprehends the system's error environment than do either the hardware or the software and, in many cases, he must determine whether or not a system level response should be initiated.

V. ERROR RESPONSES

5.1 Local responses

Local error responses are attempts to automatically confine or correct errors at the point of detection. They are important in all modes of operation, but especially in the battle mode where they are a significant factor in short-term system reliability.

Programs commonly use the centralized error-logging-and-thresholding function to report, record, and threshold errors they detect. They also perform many kinds of specific local responses designed to correct or confine the effects of a specific type of error detected. The following are several typical examples of such responses.

A program's response to a processor interrupt might be to reinitialize a critical portion of its data base using default values, to unlock any locked data sets, and to exit. If an I/O error is detected, a program might retry the I/O operation. If a radar return-tracking signal fails a data-reasonableness check, a program might employ an algorithm to "coast" the object's track for one radar cycle.

Suppose repeated error indications in the status unit for a peripheral device cause an error-report threshold to be exceeded. If the peripheral is not essential for tactical operation, the peripheral device manager could sever it, thereby degrading system operation but avoiding recurrence of the errors and also avoiding the possibility of propagating the errors into other parts of the system.

In the case where memory errors detected and reported by the real-time exercisers exceed a threshold for a certain memory rack, the only local response is the error-logging-and-thresholding function's notification to a DPS operator and to the process coordinator. Either may then initiate a system response to replace the rack with a spare. Such a replacement might be done during surveillance-mode operation, but not during a battle. During battle-mode operation, the software's local responses must be able to recover from any errors that might occur either in the memories or in other parts of the system.

5.2 System responses

System level error responses are used to reinitialize the system or to replace faulty components. They are invoked automatically by the system sanity timer or by the process coordinator in response to certain errors that cannot be easily corrected at the local level. In many instances, they are invoked manually in response to errors or combinations of errors reported by the hardware or the software. System responses are performed by the operating system but they are never initiated by it. System-error responses contribute to system availability, but they may be inhibited during a battle to prevent interruption of tactical operation.

There are three basic system level error responses: reinitialization, reconfiguration, and DPS recovery. Reinitialization involves reloading the system's entire data base. It can be initiated by the process coordinator to restore severed software components. Reconfiguration involves swapping DPS units between the green and amber partitions. It provides a method for the software's process coordinator or for an operator to replace faulty or severed hardware units in the tactical (green) partition with their redundant counterparts from the amber partition. However, DPS reconfiguration is most commonly used by an operator to switch units from the green partition to the amber partition for M&DSS testing. The most commonly used system-error response is DPS recovery. It is the easiest to use because errors do not have to be localized beforehand. It is also the only system error response which may be invoked either by hardware (the sanity timer), by software (the process coordinator), or manually by a DPS operator.

DPS recovery reinitializes the entire hardware/software system in approximately 10 to 20 seconds, depending on the CLC configuration size. Once initiated, DPS recovery proceeds automatically under the control of the operating system. It involves the following steps:

(*i*) Terminating process execution.
(*ii*) Saving the system image (including the data base, the contents of the status unit, and the contents of the processor registers) on disc for possible later analysis.
(*iii*) Running normal path diagnostics, and reconfiguring the CLC to eliminate faulty units if necessary.
(*iv*) Completely reinitializing the software by reloading all programs and the entire data base with fresh copies from disc.
(*v*) Resuming tactical execution.

VI. EXPERIENCE/RECOMMENDATIONS

The following are a few key points and recommendations based on the SAFEGUARD experience with error control. The recommendations are believed to be generally applicable to designing error control into large-scale, real-time control systems.

(*i*) A system's error-control guidelines and error-control structure must be defined early. They are required early in the design if the system is to have a consistent approach to error control.
(*ii*) Error logging must be provided as one of the first software functions. It is an invaluable debugging tool.
(*iii*) Certain error-control features, e.g., audits, must be considered early to make implementation feasible. SAFEGUARD might have made greater use of data-base audits if the data base had been designed with audits in mind.
(*vi*) Testing local error responses is difficult, but it is important for reliable operation. To enhance reliability, keep local responses simple and testable. To help simplify testing and to help reduce the amount of code devoted to local responses, categorize errors to minimize the number of different local

responses required. Many natural opportunities for testing local-error responses occur during early software testing. To take advantage of these opportunities, local-error responses must be implemented during early software development.

(*v*) Error responses should be easily modifiable. The desired responses may change as operational experience with a new system provides additional information about error occurrence rates. In the SAFEGUARD system, centralized, table-driven error-thresholding functions and system error-response maps permitted tailoring many of the local and system error responses as experience with the system grew.

(*iv*) Hardware and software status returns should be "response oriented." They should include a simple code indicating what to do about an error, that is: retry the operation; reset the device or correct a parameter first, then retry; don't retry, the device is broken; etc. More detailed status information to further identify the nature or cause of the error may also be included, but it should be independent of the response-oriented status. The detailed status may be recorded by software for off-line analysis.

(*vii*) Manual error control or manual override should be provided even for automatically operating or self-repairing systems. Manual control is essential for "bringing up" systems—even automatic systems. It is also invaluable when automatic systems fail to operate, or when self-repairing systems fail to repair themselves.

REFERENCES

1. P. H. Enslow, Jr., Ed., *Multiprocessors and Parallel Processing,* New York: John Wiley, 1974.
2. A. K. Phillips, "SAFEGUARD Data-Processing System: Debugging a Real-Time Multiprocessor System," B.S.T.J., this issue, pp. S133–S145.
3. B. P. Donohue III and J. F. McDonald, "SAFEGUARD Data-Processing System: Process-System Testing and the System Exerciser," B.S.T.J., this issue, pp. S111–S122.
4. J. R. Hahn, Jr. and F. E. Slojkowski, "SAFEGUARD Data-Processing System: Maintenance and Diagnostic Subsystem," B.S.T.J., this issue, pp. S63–S72.

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Debugging a Real-Time Multiprocessor System

By A. K. PHILLIPS

(Manuscript received January 3, 1975)

The debugging of SAFEGUARD software was performed in phases, each with a unique environment, problems, and debugging tools. The unique aspects of each phase are described here with special emphasis on the debugging tools used. Although the multiprocessor configuration introduced new kinds of software "bugs" and complicated the debugging problem, the real-time character of the system had a greater overall impact.

I. INTRODUCTION

This paper describes the debugging approach used on SAFEGUARD. The debugging effort is presented in terms of three testing phases: (*i*) unit and module, (*ii*) software integration, and (*iii*) system level. The tools and techniques required for each phase receive special emphasis. Although the multiprocessor configuration introduced new kinds of software "bugs" and complicated the debugging problem, the real-time character of the system had a greater overall impact. The debugging experience gained from SAFEGUARD is applicable to other large, real-time systems, whether multiprocessor or not.

1.1 The debugging problem

The basic steps for debugging a large, real-time multiprocessor system are essentially the same as for other software: detect the error, isolate the cause, and provide a fix. Underlying this sequence are two fundamental prerequisites: the ability to make an error repeatable and to be able to collect the data required to isolate the problem. Repeatability and data gathering, while taken for granted in simpler environments, are severely affected by real-time and multiprocessor system characteristics. Real-time execution limits the ways in which data may be collected. In fact, the very mechanism used to capture data may perturb the timing enough to cause other problems or to make the original error disappear. The multiprocessor attribute introduces further complexities. Active system components not involved in the error may destroy critical debugging data before it can be collected. Certain problems may manifest themselves in extremely complex interactions requiring closely timed, coordinated, and parallel occurrences of events. New classes of errors are introduced: timing changes due to memory queuing effects on processor speed; shared-data accessing conflicts; and intermittent, phantom "clobbers" of data. Although the great majority of errors found (e.g., incorrect register usage, destroyed data, and bad interfaces between programs) are similar to those encountered in simpler systems, those errors unique to this special environment are among the hardest to find and correct. Two other factors compounded the SAFEGUARD debugging problem. One was the parallel development of both hardware and software. The other was the amount of software involved, of which the real-time portion alone contained approximately three-quarters of a million instructions.

II. PHASE I—UNIT AND MODULE TESTING

The purpose of this phase is to test all logic paths in each program and to test the interfaces between programs. In many instances, hardware simulators extend the testing domain to encompass hardware interfaces as well.

2.1 Environment

Most of the unit and module testing occurred on an IBM support computer. A simulator called STACS (SAFEGUARD Tactical Simulator) provided the primary testing vehicle.[1] Various special-purpose test drivers and hardware simulators interfaced with STACS and enhanced its value. By eliminating the real-time and multiprocessor factors, STACS reduced the testing effort to a more common situation: program developers systematically testing their programs in a batch-oriented environment.

As soon as the CLC became available, the operating system was transferred to it for unit testing. This transition was greatly facilitated through the use of support programs that executed on the Maintenance Data Processor (MDP). Prior to entering the software-integration phase of testing, it was necessary that operating-system support capabilities be thoroughly tested and verified on the CLC. This requirement necessitated the early development of a basic set of debugging aids called DEBUG.

2.2 Debugging tools

2.2.1 The STACS simulator

STACS fully simulates the CLC processor and most of the conventional CLC peripheral units such as tapes, discs, consoles, and TTYs. It also simulates many of the CLC operating-system capabilities; in some cases, it uses the actual operating-system programs. A number of special-purpose test drivers simulate the hardware, extending the STACS testing capability. In some cases, these drivers are written in high-level languages such as FORTRAN or PL/1. These languages have the advantage of being stable, already known to many programmers, and well suited to the problem at hand. The ability to link to user-written drivers of this kind is an important consideration in designing a simulator. A good example of what can be done under STACS was the testing of the I/O manager of the CLC operating system. Although the module contained complex and widely distributed hardware interfaces, STACS allowed thorough debugging to occur on the support computer. The transition to the CLC produced few problems.

STACS provides a variety of debugging aids including register initialization, execution traces, conditional register and data snaps, and post-execution dumps. An interrupt generation capability permits error interrupt occurrences to be simulated at any specified location in a program. Coupled with the STACS simulation of the CLC operating system interrupt handling, this allows exhaustive testing of program interrupt response code. Special commands to simulate manual inputs enable man/machine interactions, which are normally asynchronous and not exactly reproducible, to be reduced to a single repeatable form for testing purposes. Run-time statistics accumulated by STACS (e.g., the number of instructions executed and the number of memory accesses) assist programmers in estimating program execution times and memory queuing loads.

The ability to temporarily patch programs and data sets proved extremely valuable. STACS supports a simple, instruction-level patch capability. To modify a program, the programmer specifies the instruction to be inserted and its offset within the program. Patching frees the program tester from time-consuming source recompilations and provides a great deal of flexibility. For example, one STACS run might contain many test cases, each created by using patches to change test data between program executions. The patch capability also permits verification of the correctness of the instructions or data being changed. Such verification eliminates two common problems: patching the wrong location and patch conflicts due to more than one patch at the same location.

2.2.2 *MDP support program*

Support programs executing on the MDP played an important role in the transition of the operating system's support capabilities to the CLC. These programs utilize the independent access paths of the Maintenance and Diagnostic Subsystem (M&DSS)[2] to interface with various CLC units. Along with the capability to load and execute bound code, they provide a set of single-processor, nonreal-time debugging aids including traces, snaps, and dumps, as well as a temporary program and data set patch capability. Attempting to debug the operating system's support capabilities without such a set of basic tools, which are provided by a separate support computer, would represent a formidable task. Later, these programs provided a capability that allowed a complete and uncorrupted snapshot dump of the system to be taken in the event of a system "crash."

2.2.3 *DEBUG—a single-processor, nonreal-time tool*

DEBUG represents the CLC operating system's first package of debugging aids. Although it includes some multiprocessor capabilities, which will be discussed under phase II testing, its design is more oriented toward a single-processor, nonreal-time environment. Its programs are not reentrant, its I/O is not concurrent with processor execution, and some of its capabilities require overlays from disc. DEBUG output may be directed either to printer or tape.

DEBUG capabilities include many of those provided by STACS and the MDP support programs. They include register initialization, traces of jump instructions or subroutine calls, conditional register and data snaps, dumps after termination, and program or data set verification and patching. A TTY interrupt capability allows an operator to interrupt program execution, request debugging actions, and then cause execution to resume. Using the breakpoint hardware of the CLC processor, DEBUG provides a breakpoint capability, which allows a trace of all accesses to a specific variable store memory location. Its patching capability became the standard approach across SAFEGUARD for fixing problems, thus eliminating the need for source-code redelivery and rebinding except at widely spaced intervals. Consistent with this philosophy, DEBUG capabilities require no special compile-time changes. For example, to cause a snap or program breakpoint, DEBUG temporarily inserts an illegal instruction into the program. When a processor encounters the illegal instruction, it interrupts, and DEBUG gains control, performs the requested debugging service, and then executes the instruction which has been replaced. The debugging "hook" exists only for the duration of the run.

2.3 Lessons learned from phase I

Phase I testing would have benefitted from better compatibility between STACS and the CLC operating system, and more complete hardware simulation by STACS. Ideally, the transition from the support to the target machine should be transparent. However, except for the patch commands, the command languages as they exist are completely different. Programmers must become familiar with a new command language prior to beginning testing on the CLC. In addition, due to the way it simulates CLC memory, STACS requires programs and data sets to have a memory allocation different from their eventual CLC mapping. Thus, rebinding was required before the transition to the CLC.

The status unit is a good example of a device which should have been simulated but was not. The status unit is a special-purpose hardware unit used to collect status information from the CLC and its peripherals.[3] Both the operating system and the application software contain numerous references to this device, and the effort required to simulate it would certainly have been worthwhile.

The ideal situation would be to leave phase I testing with only timing, multiprocessor, and some interface errors remaining in the software.

III. PHASE II—SOFTWARE INTEGRATION TESTING

The purpose of phase II testing is to integrate the software, starting with a simple nucleus of tested code and adding increments until all of the various software components are included. Testing is at an external interface level, which may involve the complex interaction of many programs and hardware units.

3.1 Environment

Phase II testing was performed on the CLC, primarily in a "hands-on" environment. There were efforts to move toward batch operations, but the complexity of the system and its unstable character during this phase limited this approach. Independent test-and-integration groups performed the bulk of the testing. For example, in the operating system area, eight to ten people were engaged full time in the debugging effort. The DEBUG patch capability allowed quick fixes to problems until the next source code update was made. During this phase, single-processor, nonreal-time testing gave way to testing in a multiprocessor, real-time environment. At regular intervals, operating-system releases provided new capabilities to the application software. Special drivers were used to simulate the missiles and radar, later to be replaced by the system exerciser[4] when it became available. During this period, test-and-integration personnel, using the DEBUG patch capability, "invented" many debugging tools as they were needed. As the debugging environment became more constrained, the debugging approaches attempted to minimize timing impact. Consistent with this evolution, the debugging tools will be presented in order of decreasing timing perturbation.

3.2 Debugging tools

3.2.1 Time suspension

As mentioned earlier, although DEBUG's basic design is not intended for a real-time, multiprocessor environment, it does include a few capabilities for dealing with both of these complicating factors. Not surprisingly, its approach, a form of time suspension, attempts to collapse the system to the simpler, single-processor environment for which it is designed.

The time-suspension strategy involves stopping the system, performing a debugging operation, and then restarting the system. To stop the system, DEBUG first stops the timing generator and then causes each processor, except the one controlling the time suspension, to be interrupted and to enter an idle loop. At this point, the controlling processor performs the requested debugging operation, e.g., a memory dump to the printer, which may consume many seconds or even minutes. To restart the system, DEBUG first restarts the timing generator and then the processors. Each processor restores its previously saved registers prior to resuming execution.

Time suspension suffers from several serious drawbacks. Using interrupts to stop processors is a serial operation, requiring 10 to 20 μs per processor. This permits scores of instructions to be executed, and proves particularly unsatisfactory in a "stop-on-error" situation. The fact that all processors cannot be stopped instantly leads to several difficulties. For example, some processors may be stopped with critical data sets locked, causing lock recovery code to be erroneously triggered on one of the processors still running. An even more serious difficulty is that time suspension does not work with synchronous peripherals such as the radar. DEBUG cannot correctly stop and restart the radar's internal clock and, therefore, cannot preserve its timing relationship with the data-processing system. Early in phase II testing, when synchronous peripherals and large processor configurations were not used, time suspension proved helpful.

3.2.2 System image save

The "system image save" is one of the most important data-gathering tools, providing a complete snapshot of the system. Preceding the save, the system is collapsed to a single-processor, nonreal-time state. Following the operation, the software must be reloaded prior to restarting the system. The system image includes the entire data base, all processor registers, the contents of the status unit, and the contents of internal hardware registers. The information is written to tape or disc, the entire operation requiring only a few seconds. Test-and-integration personnel invoke this capability manually when they suspect the occurrence of a serious error. During phase II testing, the automatic invocation of it by DEBUG in response to an error interrupt was important. In phase III testing, a system-image-save automatically occurs as a first step during system-error-recovery operations.

3.2.3 Real-time simulation

Real-time simulation on the CLC is another useful technique for reducing the debugging effects of a time-constrained environment. Two SAFEGUARD approaches deserve mention: one employs the timing generator and the other eliminates it entirely. The operating system manages processors by dividing time into discrete units, called phases. The length of a phase is determined by the timing generator and can be increased by simply programming the timing generator such that the phase length is longer than it normally would be. This approach does not eliminate the timing generator's time constraint, but does provide a continuum of execution rates from nonreal time to real time.

The other approach used on SAFEGUARD employs a software mechanism instead of the timing generator to control software execution and phase length. In order that I/O jobs may terminate properly, a minimum time between phases is enforced. This approach eliminates the timing generator's time constraint completely, allowing a task's execution time to extend as long as necessary, e.g., for many seconds or minutes in the case of a dump of processor registers on the printer. An additional benefit is greater repeatability since the elimination of the hardware clock reduces many of the run-to-run variations which normally occur. However, because real-time simulation precludes synchronous peripheral interfacing, its use was confined to the early portion of phase II testing.

3.2.4 DARTS—a low-perturbation tool

The intent of Debugging Aids for Real-Time Systems (DARTS) is to provide debugging capabilities in a multiprocessor, real-time environment with a minimum of timing perturbation. This environment includes normal timing-generator and radar operation. The underlying assumption is that debugging actions can be performed during normal processor idle time. The design of DARTS resembles in many respects that of the real-time portion of the operating system: reentrant programs that are permanently resident during execution; a real-time component driven by tables constructed in nonreal time; and service times low enough to be measured in microseconds.

DARTS permits the establishment of program breakpoints at which desired debugging actions can occur. These actions include both data-collection and data-manipulation services. Actions can be conditional, depending on register contents, data values, the operating system phase, the arrival of a specified point in time, or the completion of a specified time delay. Breakpoints can be enabled or disabled during execution, providing added flexibility. A manual input simulation capability permits complex man/machine interactions to be reduced to a list of DARTS commands. This feature offers a number of significant benefits. First, repeatability is increased since the simulated inputs can be timed precisely. In addition, the number of operators required is reduced, the possibility of operator error is virtually eliminated, and run times are shortened considerably. DARTS also provides an interrupt-simulation capability which proved extremely useful in debugging the extensive interrupt-response code within the operating system.

Instead of dumping captured data to the printer, DARTS either accumulates it in circular buffers or writes it on tape using the operating system's recording capability. At termination, information in the circular buffers can be dumped in chronological order.

DARTS provides a flexible, easy-to-use, high-level language with which test-and-integration programmers can create their own debugging tools. It incorporates many of the ideas and techniques learned during the early SAFEGUARD debugging experience.

3.2.5 Event traces and error logs

The operating system provides a number of historical traces and logs of key system events, including both normal occurrences and errors. These data-collection capabilities are extremely valuable in debugging and performance analysis. The normal path traces include task executions, status-unit bit changes, and manual inputs. For each error occurrence, the operating system generates a four-word entry containing the time of the error, its category, and two data words that are dependent on the particular kind of error. The event-trace and error-log information is accumulated in memory and, periodically, is written to tape using the operating system's recording capability. The information remaining in memory becomes an important portion of any system image save which may be made. It reflects the key system events leading up to a serious error occurrence.

3.2.6 Data recording

The operating system provides a flexible and powerful data-recording capability which permits continuous data collection onto tape with a capacity of one hundred thousand 32-bit words per second. Numerous recording calls are permanently embedded both in the operating system and the application software. These calls may be easily augmented using DARTS. Both software and manual controls permit individual recording categories to be turned on or off. Thus, the recording stream can easily be adjusted to meet the needs of particular test situations or suspected errors.

In addition to recording the various event traces and error logs described earlier, the operating system supports special recording capabilities relating to processor interrupts and CRT displays. Specifically, on a processor interrupt, the operating system records the processor registers and stack information. The stack contains temporary data variables and information sufficient to recreate the chain of programs leading up to the interrupted program. These interrupt-related data become increasingly useful in phase III testing when continuous operation in the presence of errors, including interrupts, becomes commonplace. The operating system provides the capability to record CRT displays. This output can be reduced using special programs on the support computer, producing a "hard" copy of displays. Verifying the correctness of displays in this manner is more convenient than taking photographs.

3.3 Lessons learned in phase II

The most obvious lesson from phase II testing is that debugging approaches suitable for nonreal-time, single-processor systems are not adequate for a system like SAFEGUARD. Specifically, the philosophy of minimum perturbation as exemplified in DARTS is far superior to the time-suspension technique used by DEBUG. For time suspension to be feasible, hardware mechanisms to allow abrupt stopping and restarting of all active system elements (e.g., processors and clocks) must exist.

The second lesson is that debugging aids must be developed early, well ahead of the software which will use them. Waiting for experience to provide feedback on what tools are needed does not allow sufficient time for their development. A solution to this dilemma is to provide the test-and-integration personnel with the tools to construct debugging aids as the need arises. The patch capability is the simplest example of this approach while DARTS represents its easy-to-use culmination. An analogous problem occurred in developing individual operating-system tests. Often the test team would identify new areas requiring testing. However, the amount of time required ruled out the normal test-development cycle. The solution was a software facility that allowed quick test generation using a simple, high-level command language.

IV. PHASE III—SYSTEM TESTING

The purpose of phase III testing is to verify that the software and hardware work together as a system in an environment that resembles as closely as possible the expected operating conditions.

4.1 Environment

During phase III, "hands-on" testing continued, primarily in a real-time, multiprocessor environment. The completed system exerciser became the test driver for the process. The duration of test runs increased and, in some instances, testing extended for periods of many hours. As confidence in the extensive error-recovery code in the system increased, "stop-on-error" modes of testing declined. Errors provided unexpected opportunities to verify the software error response. Load testing and process tuning became important. Netted tests which involve multiple site interactions occurred frequently. Across SAFEGUARD the number of official patches grew into the thousands, requiring extensive control- and quality-assurance measures. The debugging tools developed in phases I and II remained available, permitting changes to the software to reach the test groups in a well-tested state. Although most of the debugging aids described previously continued to be used, the tools that were permanently part of the applications software and that normally caused the least timing perturbation were the most important. These included the event traces, error logs, data recording, and the system image save. Data recording and reduction were the tools that had the most widespread use during this phase of debugging.

4.2 Additional debugging tools

4.2.1 CLC hardware monitor

The CLC monitor is an external hardware monitor which includes its own memory, extensive logic to count and filter data, and two tape units. Although its primary use has been to gather system performance measurements, it has proven valuable in debugging two areas. One is the kernel of the CLC operating system, where normal debugging tools cannot be used. The other includes extremely time-critical portions of the system where the insertion of debugging "hooks" causes an unacceptable perturbation. The mechanism for transferring the software event information to the monitor is a single store instruction, which increases task execution time by approximately

1 μs. A number of these monitor instructions are permanently embedded in the software.

4.2.2 Operating system testing during phase III—the system test "cycler"

A special test process called the system test cycler tested the operating system in an environment quite different from that of phase II testing. It exemplifies the kind of testing done in phase III. The cycler allows continuous testing of the operating system over periods of many hours. Special logic within the cycler exercises many of the conventional data-processing peripherals (e.g., tape and disc) and the operating system software which manages them under extremely heavy loads. Using a TTY command, test personnel can insert simulated hardware faults into memory units and processors, verifying that the operating system can detect and recover from the errors. Most of the error-recovery mechanisms provided by the operating system can be exercised using the cycler, either manually or automatically. Besides uncovering numerous software and hardware problems, the cycler provided a test-bed for verifying many of the changes made to the operating system during phase III.

4.2.3 Visual error-detection aids

During phase III, visual error-detection aids became increasingly important. In a system such as SAFEGUARD, where no observable activity normally occurs, visual signs are needed to inform the operator as to system "health." Error indicators may prompt him to enable recording, or they may serve as clues as to which portion of the total recording output should be reduced. In addition to error messages, wall display boards, and various error light indicators, the operating system provides a CRT memory dump display. This allows areas of memory or the status unit to be viewed. In this same category is a printer trace[4] of key events which was extensively used during the phase II testing of the application software. It provided a window through which the system tester could observe the continuous functioning of the process. Although an important testing capability, it was never made a permanent part of the system.

4.3 Lessons learned in phase III

The most important deficiency uncovered during phase III testing was the absence of sufficient visual indications to determine what was really happening inside the computer. One solution proposed, but never implemented due to lack of available memory space, was a "vital signs" CRT display. Such a display might show the accumulated errors on various units, the amount of I/O and processor activity, or key radar and missile information.

V. RECOMMENDATIONS

Table I lists the capabilities discussed in this paper. If one capability could be singled out as the key to the SAFEGUARD debugging success, it would be the ability to patch programs. It eliminated the need, except at widely spaced intervals, for time-consuming source-code redeliveries and system reverification. In addition, patching provided a flexible, easy-to-use tool through which new debugging aids and test tools could be created.

The importance of unit and module testing cannot be overemphasized. A high percentage of the bugs found in the later phases could have been eliminated in phase I. Therefore, it is highly cost effective to provide extensive unit and module test facilities. Programs which bypassed phase I testing, either because of extensive hardware interfaces or schedule constraints, generally became long-term problems during later phases of testing.

The early consideration of three vital areas is mandatory: error logging, data recording, and other special debugging aids. On SAFEGUARD, error logging and data recording could have simplified debugging if they had been available earlier. The tendency to postpone consideration of these areas because they are not critical capabilities

Table I — Use of debugging tools by testing phases

Debugging Tools		Testing Phases		
		I	II	III
CLC simulation on support computer (STACS)		√	√	
Unit debugging aids	MDP programs	√		
Dump capability	MDP programs		√	
Unit debugging aids on CLC	DEBUG	√	√	
Program patching	DEBUG	√	√	√
Time suspension	DEBUG		√	
Real-time simulation			√	
Low-perturbation aids	DARTS		√	
Manual input simulation	DARTS		√	
Error-interrupt simulation	DARTS		√	
System image save			√	√
Event traces and error logs			√	√
Data recording			√	√
CRT memory display				√
Printer trace of key events			√	√
CLC hardware monitor				√
System test cycler				√

should be avoided. In the case of other specialized debugging aids, it is clear that waiting for actual testing experience to reveal what tools are needed is unsatisfactory.

Although it may seem obvious, the availability of an experienced nucleus of people may be the best guarantee of success. The Meck test system prototype effort which preceded SAFEGUARD provided a sizeable pool of real-time, multiprocessor experience, which proved invaluable in testing the SAFEGUARD system.

REFERENCES

1. R. R. Conners, "SAFEGUARD Data-Processing System: Support Software and Support Computers: An Overview," B.S.T.J., this issue, pp. S149–S160.
2. J. R. Hahn, Jr. and F. E. Slojkowski, "SAFEGUARD Data-Processing System: Maintenance and Diagnostic Subsystem," B.S.T.J., this issue, pp. S63–S72.
3. J. W. Olson, "SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control," B.S.T.J., this issue, pp. S41–S61.
4. B. P. Donohue III and J. F. McDonald, "SAFEGUARD Data-Processing System: Process-System Testing and the System Exerciser," B.S.T.J., this issue, pp. S111–S122.

Section IV

SUPPORT SYSTEMS

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Support Software and Support Computers: An Overview

By R. R. CONNERS

(Manuscript received January 3, 1975)

This paper gives a critical overview of the development of a selected set of support software programs. These programs comprise part of the support environment required to make the SAFEGUARD system application software and hardware operational.

I. INTRODUCTION

In the development of the SAFEGUARD system or any other large-scale programmed system, one may distinguish between two types of software—application and support. Application software does the job for which the system is intended. In SAFEGUARD, this means driving the radars, tracking objects, firing missiles, etc. Support software includes all other software required to make both application software and hardware operational.

This paper, and others that appear in this section, cover a limited set of support software programs. Included in this set are

(*i*) CENTRAN—A compiler for a high-level extendible language of the same name.
(*ii*) SNX—A macro assembler.
(*iii*) XPF—A binder for preparing CENTRAN and SNX output for execution.
(*iv*) STACS—A simulator for execution of XPF output.
(*v*) SDRS—A set of programs for reducing, i.e., decoding and formatting, data recorded during CLC execution.

All these programs have several things in common. First, their purpose is to assist the development and debugging of SAFEGUARD application software. Second, for reasons to be discussed in detail later in the paper, these programs operate on computers other than the Central Logic and Control (CLC).*

* The CLC is discussed in Ref. 1. Discussions of support software that operate on the CLC may be found in Ref. 2.

This paper critically examines some key decisions that shaped the programs and the environment in which they operate. In doing this, the groundwork will be laid for the other papers in this section. The paper is divided into two sections: the first covers support software; the second covers the computers on which the support software programs were developed and on which they run.

What is the value of such a critical examination? The frequency with which programming projects fail or repeat the mistakes of their predecessors leads to the conclusion that the knowledge required to manage program development is largely based on experience. Perhaps the communicating of experiences, successful and unsuccessful, may help to transmit some of the knowledge gained on the SAFEGUARD project.

Why is SAFEGUARD a good choice for such an examination? Because its development and that of its prototype span a ten-year period that overlaps the development of the "science" of programming management. Because during those ten years many intensely creative people were involved, and since the magnitude of the project was enormous, their creativity was not constrained. Because the SAFEGUARD effort encompassed a multiplicity of organizational structures within Bell Laboratories and its various subcontractors. Because it seems that, at one time or another, practically everything was tried or seriously studied.

During the ten-year period examined, one can distinguish three successive phases of support software development. Each phase was built on the lessons learned during previous phases, and each phase had its own characteristic spirit. The first phase was part of the NIKE-X antiballistic missile research and development effort in the mid-1960s.[3] The second phase was the development of support software for the Meck test system, a prototype system intended to validate some of the results from the first phase. The third phase was the development of support software for SAFEGUARD, which has applied the results of NIKE-X research on an even wider scale. The topic of this paper is SAFEGUARD support software, but since it has been shaped by a synthesis of NIKE-X and Meck test system experiences, it is with these that the story must begin.

II. SUPPORT SOFTWARE FOR NIKE-X AND THE MECK TEST SYSTEM

In the early days of NIKE-X, support software designers envisioned an environment in which the application programmer "typed" high-level language programs at a terminal under control of a time-sharing system operating on the CLC. Programs would then be processed by a global optimizing compiler and executed, with results routed to the programmer at his terminal. In retrospect, these ideas are perfect examples of the optimism that pervaded the computer industry in the mid-1960s. As the industry lost its innocence, these ideas crumbled. The CLC time-sharing system, Program Development Facility (PDF), was ultimately dropped when it became apparent that there was not enough CLC time to allow concurrent development of hardware, support software, and application software. PDF development had lost not only its means but its end as well, for application software development for the Meck test system had already passed the unit testing stage where PDF could have been most helpful.

A global optimizing compiler, NIKE-X Compiler Language (NICOL), was abandoned after completion of three of the four planned stages of its development. A working, but incomplete, compiler (NICOL 3) had been built for the CLC, but there was insufficient CLC time for support software development and compiler maintenance. This, combined with a long time estimated for completion of the optimizing phase given the available manpower and with technical problems encountered in maintaining NICOL 3, indicated that NICOL would not be able to meet the needs of the project. Meanwhile, coding for the Meck test system had been done in assembly language so the compiler was needed only for SAFEGUARD software development.

A message in the NICOL and PDF stories will recur in this paper. Support software goals must be realistic, particularly in the sense that they be attainable at the time they are required. Essential features must be available on schedule. The purpose of support software is, after all, to support the objective of building systems. Building state-of-the-art support software as well is laudable, but only if it contributes to the main objective.

These experiences led to several important decisions that influenced support software development for SAFEGUARD. First, it was decided to move as much support software activity as possible to a computer other than the CLC. This had the desired effect of making the CLC more available for application software testing, but it also resulted in a sizable support computer requirement which is discussed in Section V of this paper.

A second major decision was that application program development would be done primarily in batch rather than in time-sharing mode. This decision was reached over the course of several years and was based on many contributing factors. First, experiences with MULTICS and TSS[4] were disappointing, in the sense that they did not support the expected number of users at the predicted time. Second, other available, or nearly available, time-sharing systems appeared to have serious limitations: CP/CMS was not file-compatible with IBM System

360. TSO, which IBM advocated in place of TSS, was limited in capability and lacking in human engineering. No system provided adequate availability and reliability. Third, interactive support software would have to be written and it appeared easier, and therefore faster, to write support software that would operate in the batch mode. Finally, the exact trade-offs between batch and time-sharing program development were not yet known (and perhaps still are not). It was speculated that time-sharing might improve programmer productivity, but it would do so at the expense of additional computer requirements. It would have been difficult for management, already faced with large development costs, to commit itself to time-sharing without a better understanding of its cost effectiveness.

This shifting support software philosophy affected the development of application software for the Meck test system. From the ashes of PDF and NICOL emerged a set of support software tools that were sufficient to get the job done. Application programs were coded in assembly language. Linking and binding the various application programs was an intricate and awkward process. Computer program source was stored and distributed on cards. The key to effectiveness was an ironclad set of control procedures that made the system work. Despite a lack of sophisticated support software, the Meck test system has consistently met its objectives.

III. BUILDING SUPPORT SOFTWARE FOR THE SAFEGUARD SYSTEM

The decision to deploy the SAFEGUARD system had a direct impact on support software. It would be necessary to improve the programs and procedures used to develop the Meck test system, since SAFEGUARD was a larger and more complex undertaking. However, there was very little time to implement these improvements and still meet tight development schedules.

One of the first decisions made was to "borrow" software. Under the aegis of the ESS project, Bell Laboratories had developed a modular assembler called SWAP[5] that was specifically designed for portability—as long as there was an IBM System 360 available. Borrowing SWAP, converting it to generate CLC machine code, and relabeling it SNX 360 yielded a fast, efficient assembler at minimal cost. However, it was necessary to provide people to maintain and modify SNX 360 as requirements grew. SNX 360 was adopted by both SAFEGUARD and the Meck test system.

What about a programming language for the deployed system? Assembly language had been used exclusively for the Meck test system and was favored by most of the "old pros." Management and support software designers were concerned because experience indicated that a programmer could write and maintain a fixed number of statements per month, no matter what language was used. The use of high-level languages promised greater productivity. Management also knew that system requirements would be fluid in the early stages of development, requiring frequent changes. This supported the argument for the use of a high-level language.

The only high-level language available for the CLC was NICOL, however. At that time there was a working, but incomplete, NICOL compiler of unknown reliability generating inefficient code. It ran slowly and was very difficult to maintain. Estimates of the time required to complete the final phase of NICOL development, an optimizing compiler, were unacceptably long. This was due in part to the inherent difficulty in generating optimal code for a machine whose instruction set had not been designed with an optimizing compiler in mind. In addition to the flaws mentioned above, NICOL did not allow programmers to "get at the machine," i.e., access the CLC hardware registers. This alone was enough to make NICOL unsuitable for a major part of CLC software, the operating system. What was needed was a language that was easy to use and available immediately, could produce optimal code, allowed programmers to access the CLC registers, was efficient, and was sufficiently rich in syntax and semantics to serve the needs of system programmers, tracking and filtering analysts, and radar and missile control programmers.

What evolved from this need was CENTRAN,* a high-level extendible language. The compiler for CENTRAN was coded as a SNX 360 macro package. While this implementation permitted early compiler availability, it did result in long compilation times. Later improvements increased the speed, but not to the point where it was comparable with most compilers.

CENTRAN offered the programmer his choice of language level, from assembly language to something resembling a subset of PL/1, and statements of varying levels could be intermixed. If the programmer felt that the language syntax was inadequate, he could extend it with relative ease. Extendibility allowed for development of the language in stages, so a minimum facility could be made available to users almost immediately. Reference 7 contains a retrospective look at some of the design issues in CENTRAN.

As a postscript to the programming language issue, it is interesting to note that use of CENTRAN was decreed rather than sold. There are several "extreme" reasons why this had to be. On one side, CENTRAN designers felt that the CLC programmers' attachment to assembly

* CENTRAN is described by its inventor under the name ETC. See Ref. 6.

language was excessive* and that the programmers' cries of inefficiency were misdirected. Efficiency concerns, argued the CENTRAN designers, should be directed first to the design of the program and data base, rather than to the programming language. Many programs need not be coded with scrupulous efficiency. Further, with a knowledge of a few basic facts of CLC architecture, the programmer could write CENTRAN statements that would generate code just as efficiently as an experienced assembly language programmer could. The programmers countered with arguments of their own. They claimed, with some justification, that CENTRAN was unreliable. CENTRAN taxed SNX 360 macro capabilities as they had never been taxed before. If a programmer made a syntax error, his compilation would occasionally abort without a diagnostic or produce a SNX 360 diagnostic that was meaningless to him. Programmers also complained about CENTRAN documentation, again with justification. It is impossible to write adequate documentation, construct courses, and reeducate 600 programmers overnight. These things take time to evolve, and while they do, programmers suffer. The battle over CENTRAN raged for some time and became quite bitter. But in the end, CENTRAN was used and programs were written.

Programs were indeed written, several thousand of them, in fact. A better source code storage and control mechanism was needed to replace the card-oriented Meck test system approach. A disc-based filing system was under development, but not near completion. Bell Laboratories accepted the offer of the use of an IBM proprietary system that had been used in the development of IBM programs.

This product is a comprehensive disc-based source-object listing filing system which offers programmers many of the features required in the software development process; for example, an editor for changing source lines, a means of temporarily changing source for testing, and a mechanism to facilitate delivery of completed code. In addition, the system helps to protect the programmer from his own or others' mistakes by allowing limited access to libraries. Initially, users were not enthusiastic about the system, and management pressure had to be applied to ensure its use. Complaints centered around reliability and documentation deficiencies. In retrospect, however, the decision to use the system proved to be a good one, primarily because of the procedural discipline it forced on the programmers.

* Part of this attachment is really an unwillingness to give up comfortable, familiar coding patterns. B. N. Dickman relates an anecdote that illustrates this vividly. When he joined the project in 1967, he found that CLC programmers were hard-coding base registers in their instructions. He implemented a USING pseudo-operation similar to the one in IBM System 360 BAL. But he found it difficult to get programmers to use this most helpful and completely noncontroversial feature.

CENTRAN and SNX 360, like most compilers and assemblers, produce relocatable object code. A program was developed to allocate CLC memory, build the control tables needed by the operating system, perform binding functions, structure the overlays, and perform a host of related services.* This program was named the Execution Preparation Facility (XPF).

XPF was possibly the most volatile of the support software programs. As new capabilities were incorporated into the CLC operating systems, XPF had to be changed to reflect these new capabilities in the bound versions of users' programs. This could have been a massive coordination problem, but the XPF designers found a creative solution. First, they implemented XPF in PL/1, which facilitates changes, and made clever use of its preprocessor to automate change as much as possible. Second, they planned a series of releases with incrementally expanding capabilities and coordinated them with the development schedules for the CLC operating systems.

XPF is discussed in Ref. 9. This paper by Van Sciver focuses on the use of PL/1 and its consequences, stressing the additional flexibility which its use affords.

All the support software discussed above is classical in the sense that its operation is well understood by every student of computer science and that most technical problems involved have been studied theoretically and translated into cookbook solutions. Because of its complexity, the SAFEGUARD application requires an additional support facility operating in an area where the theory is not well understood. The basic question is how does one validate a real-time multiprocessing system as complex as SAFEGUARD? How does one really know what has happened inside the CLC? To answer the latter question, the capability of data recording is provided by the CLC operating system. Recording makes it possible to transmit data of the designer's choosing to tape at the rate of three million bits per second during CLC execution. These raw data would fill more than 150 printed pages for each second of CLC execution. Clearly, some automated techniques are required to help the designer through this morass of data. This is the function of reduction programs that group and format the information for orderly and meaningful presentation. These programs are called the SAFEGUARD Data Reduction System (SDRS).

A lack of clearly specified requirements makes designing data reduction programs difficult. The designer of an assembler, a compiler, or a simulator can take much for granted. A large body of knowledge exists about these programs, and techniques for implementing them

* The rationale behind the choice of functions may be found in Ref. 8.

have been studied extensively. The designer of a data reduction system has little theoretical knowledge from which to draw. Although the data reduction problem was documented as far back as SAGE, very few people have extensive experience in debugging large-scale real-time systems, and even fewer people understand the importance that real-time debugging considerations play in data base design. So the data reduction designer gets little help from any one, and must build the most flexible and basic package possible in the hope of meeting user requirements as they are discovered.

SDRS is a generalized information storage and retrieval system, part of which was borrowed from another Bell Laboratories application and adapted for SAFEGUARD.[10]

IV. SOME CONCLUSIONS ABOUT SUPPORT SOFTWARE

What made certain support programs more successful than others? Obviously, the more successful ones met the needs of the users. They were available when they were needed, were flexible enough to react as requirements changed, and were reliable. Various methods were used to achieve these objectives. High-level languages were used to retain flexibility. In fact, flexibility was considered so important that efficiency was sacrificed. Software was borrowed shamelessly, but with the knowledge that it would have to be maintained. High-risk state-of-the-art approaches were avoided. Incremental implementations were planned so that programs could be used as quickly as possible. Strict testing and release procedures were adopted to ensure quality. Programs were "frozen" after release and became subject to change-control procedures. Stringent control was placed over the interfaces between the facilities to ensure integrity. All these techniques helped to build a successful support software system.

V. THE USE OF SUPPORT COMPUTERS IN SAFEGUARD DEVELOPMENT

As indicated earlier in this paper, a basic decision was made to move as much support software work as possible from the CLC to a commercial computer. This led to major involvement in the use and operation of commercial computers. Items to be discussed include the computers currently being used and how they were selected, their locations, and some of the techniques used to improve cost effectiveness.

In the mid-to-late 1960s, NIKE-X computing was performed exclusively by the Bell Laboratories, Whippany, General-Purpose Computer Center, which operated an IBM 7094 and a GE 635. NIKE-X support software development work had been divided between the GE and the IBM systems. PDF and NICOL development was being done on the GE 635, while the CLC assembler and rudimentary binding facilities were tied to the IBM 7094. The IBM 7094 was to be phased out when MULTICS became available. However, when it became necessary to choose a computer for installation at Meck Island, the GE 635 PDF-NICOL package was not ready. The only alternative was to code programs from the Meck test system in assembly language. Since there was no CLC assembler for the GE 635, it was necessary to install an IBM System 360/65 (which could emulate the IBM 7094 at lower cost than the original) on Meck Island. Once the commitment had been made for Meck, the IBM 7094 at Whippany could not be released, since it was required for compatibility with Meck. The GE 635 then began to disappear from the mainstream of software development activities. It remains in use full time, principally to produce fault-location dictionaries for SAFEGUARD equipment.[11]

Meanwhile, the Bell Laboratories IBM 7094 was also replaced with an IBM System 360/65 and the CLC assembler and program preparation facilities were run under emulation. Gradually, these programs were converted to run on the IBM System 360/65 in its native mode for increased efficiency. As each conversion occurred, Meck test system dependence on IBM System 360 hardware increased.

Support software for SAFEGUARD further increased the dependence. SNX 360, the IBM proprietary library system, and parts of SDRS were borrowed from other IBM System 360 installations. CENTRAN was built as a macro package on top of SNX 360. XPF was coded in PL/1. Conversion to a non-IBM computer would have severely delayed the operational date of the SAFEGUARD system because virtually the entire support software system would have had to be replaced.

As the project grew, the number (and size) of support computers grew with it. Table I is a summary of the dedicated support computers used for SAFEGUARD and the Meck test system.

The prime purpose of these computers is to provide whatever service is necessary for program development. The challenge in operating them is to do the job in a cost-effective manner. Some methods used

Table I — SAFEGUARD-dedicated support computers (November 1973)

Computer	Location
IBM 370/165	Madison, N.J.
IBM 370/165	Morris Plains, N.J.
IBM 370/155	Concrete, N.D.
IBM 370/155	Nekoma, N.D.
IBM 360/50	Colorado Springs, Colo.
HIS 635	Whippany, N.J.
IBM 360/65	Whippany, N.J.
IBM 360/65/40	Meck Island

to achieve this will be discussed in the following paragraphs. Because the list of such methods is potentially limitless, the discussion focuses on items that might be considered surprising or controversial.

System tuning usually comes to mind first when one contemplates increasing cost effectiveness. Two kinds of system tuning can be distinguished immediately, synthetic and analytic. Synthetic tuning consists of an algorithmic application of such familiar standbys as data set placement optimization and channel balancing. Synthetic tuning is a necessary but not a sufficient condition for a tuned system. The slack is taken up by analytic tuning, which is an attempt to view the computer as a system, to ferret out the bottlenecks through use of the analyst's bag of tricks, to rank the bottlenecks in terms of their system impact, and to propose and implement solutions.

Different types of analysts can be identified by their approach. The most common and perhaps most effective type is the mystic, who appears to find problems by using a logic compounded of experience and intuition. Other types of analysts are the theoreticians, who try to construct classical proofs that one course of action is better than another, and the simulators, who attempt to model the computer with a series of parameters that can be varied to determine an optimum course of action. Neither the theoreticians nor the simulators were well represented on SAFEGUARD. What one did find on SAFEGUARD were the chartists. The chartist believes that it is possible to build one or more computer performance reports that will tell where the bottlenecks are and where tuning is required. One has only to decide which numbers to include in the charts and how to interpret them.

The chartist approach was very successful, but not exactly in the way it was intended. The most significant information derived from the charts was that the biggest system bottlenecks were the users themselves. Reading the reports seemed to show who was a good programmer and who was not, who was getting his job done and who was not, and who was hogging resources at the expense of others. This caused some rethinking about what it really meant to improve throughput. In a global sense, if the programmers do not get their job done, what difference does it make if turnaround time has been reduced or CPU utilization increased? To improve throughput, both the users and the system must be tuned.

Realization of this fact resulted in a campaign to search out those who were using the computer inefficiently. This endeavor was called "bird-dogging."[12]

Another method used to increase the cost effectiveness of SAFEGUARD computer-center operation was the use of plug-compatible components when their use offered equal or superior performance at lower cost.

This had the additional benefit of supporting the government's objective of fostering competition within the computer industry. SAFEGUARD experience gained in replacing IBM 2314 disc storage, IBM 2401 tape drives, and IBM System 370/165 memory does not differ substantially from that reported in the literature. However, certain key points should be mentioned.

First, it is important to have a solid grasp of the economic situation. A commitment to install plug-compatible equipment is also a commitment to expend the funds and the manpower to select replacement equipment and then to make sure that it works. SAFEGUARD has found this to be a nontrivial investment. One must be relatively certain that new announcements or price reductions will not eliminate the expected savings. IBM's announcement of 2319 disc drives cut heavily into the net savings that accrued from SAFEGUARD's switch to plug-compatible 2314s. On the other hand, the conversion to plug-compatible IBM System 370/165 memory has been enormously successful from an economic point of view, primarily because it was decided to purchase the IBM System 370/165s rather than convert to System 370/168s.

Second, it is important that the vendor have a good local service organization with trained, competent people backed by a hierarchy of expertise including the design engineers themselves. Looking back over several years' experience with plug-compatible equipment, the major problems encountered always seem to be due to difficulties within the service organization.

The third and last technique for improving computer-center cost effectiveness is the use of facilities management companies to operate SAFEGUARD computer centers. The IBM System 370/165s at Madison and Morris Plains, N. J., and the IBM System 360/50 at Colorado Springs, Colorado, were all operated this way. There are many advantages to such an arrangement. Perhaps the most significant one is the emphasis that the facilities management companies place on service. In the case of both companies used by SAFEGUARD, the results have been outstanding. This is partially because facilities management companies exist to provide service and partially because of the nature of Cost-Plus-Award-Fee (CPAF) contracts,[13] which make it financially advantageous for the subcontractor to do his best. Another advantage is that Bell Laboratories need not recruit and hire additional computer operations specialists. Finally, there is the issue of cost. Experience on SAFEGUARD has shown that, when all considerations are taken into account, a facilities management company can provide excellent cost-effective service.

The principal disadvantage of subcontracting computer-center operations is that the company selected occasionally takes a parochial view toward project needs. For example, one company felt that it was extremely important to reduce average turnaround time. What they should have tried to do was to ensure that jobs received turn around in accordance with their importance to the project, regardless of the effect on the average turnaround time. Here again, the CPAF contract is a valuable tool for ensuring that the subcontractor's goals remained aligned with the contractor's goals.

VI. SOME CONCLUSIONS ABOUT SUPPORT COMPUTERS

It is impossible to point out in a few pages all the important lessons from almost ten years of computer-center management experience. In this paper, the emphasis has been put on high-return items—bird-dogging, plug-compatible equipment, and facilities management. Other items, such as hardware and operating system change control and inter-location compatibility, were addressed but have not been discussed. Despite overall success, solutions to many problems were not found; e.g., how does one get users to estimate their computing requirements correctly, or what is an accurate measure of performance improvement? Perhaps our experiences can help others to increase cost effectiveness.

REFERENCES

1. J. W. Olson, "SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control," B.S.T.J., this issue, pp. S41–S61.
2. J. P. Haggerty, "SAFEGUARD Data-Processing System: Central Logic and Control Operating System," B.S.T.J., this issue, pp. S89–S99.
3. N. H. Brown, M. P. Fabisch, and C. J. Rifenberg, "SAFEGUARD Data-Processing System: Introduction and Overview," B.S.T.J., this issue, pp. S9–S25.
4. "No. 1 Electronic Switching System (ESS)," B.S.T.J., *43*, No. 4, Parts 1 and 2 (September 1964).
5. M. E. Barton, "The Macro Assembler, SWAP—A General Purpose Interpretive Processor," Proc. AFIPS FJCC, *37* (1970), pp. 1–8.
6. B. N. Dickman, "ETC—An Extendible Macro-Based Compiler," Proc. AFIPS SJCC, *38* (1971), pp. 529–538.
7. B. N. Dickman, "SAFEGUARD Data-Processing System: CENTRAN: A Case History in Extendible Language Design," B.S.T.J., this issue, pp. S161–S172.
8. J. P. Haggerty, "SAFEGUARD Data-Processing System: Central Logic and Control Operating System," B.S.T.J., this issue, pp. S89–S99.
9. P. A. Van Sciver, "SAFEGUARD Data-Processing System: Systems Programming in PL/I," B.S.T.J., this issue, pp. S173–S180.
10. E. S. Hoover and R. A. Jacoby, "SAFEGUARD Data-Processing System: Data Reduction System," B.S.T.J., this issue, pp. S181–S189.
11. C. J. Rifenberg, "SAFEGUARD Data-Processing System: The Dictionary Approach to Digital Maintenance," B.S.T.J., this issue, pp. S73–S85.
12. J. P. Kuoni, "SAFEGUARD Data-Processing System: A Means to Effective Computer Resource Utilization," B.S.T.J., this issue, pp. S191–S196.
13. W. H. Mac Williams and J. E. Peterson, "SAFEGUARD Data-Processing System: A Cost-Plus-Award-Fee Contract for a Large Software Development Program," B.S.T.J., this issue, pp. S238–S244.

Copyright © 1975 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

CENTRAN—A Case History in Extendible Language Design

By B. N. DICKMAN

(Manuscript received January 3, 1975)

The history of the design and implementation of CENTRAN, an extendible language, is presented as an example to language designers. The history is viewed in the context of four groups of factors: environmental issues, general design issues, specific design issues, and implementation issues. The paper concludes with an evaluation of the design decisions that were made.

I. INTRODUCTION

There are many papers about the syntax and semantics of computer languages. There are some papers about the compilers for these languages. But there are few papers describing how and why a language was designed and how it was implemented. In presenting the design history of CENTRAN,* we attempt to provide a method that language designers may apply to improve the writing of software.

Previous attempts at building a language for SAFEGUARD either attempted to provide a shell† language like PL/1 (NICOL), the entirety of which could be implemented or understood only with extreme difficulty, or attempted to provide a complete syntactical uniformity of the machine language structure, like PL360. The attempt to provide syntactical uniformity failed because requisite hardware uniformity does not, in fact, exist. At the assembly language level, the syntax of a language cannot be more uniform than the structure of the object machine.

CENTRAN can be viewed as an extendible language in which several levels of language features exist. At the lowest level, CENTRAN is the assembly language. At the next level, CENTRAN provides a uniformity for the machine by completing incomplete data paths and by providing

* CENTRAN and ETC are different names for the same language (see Ref. 1).
† A shell language attempts to provide all the features users would ever want.

uniform register usage. At this level, CENTRAN is still almost one-to-one with the machine code, but provides a more concise syntax for the machine operations by means of, for example, polymorphic operators. At the next level, machine dependence may still exist in the form of hardware register references, but CENTRAN functions as a true compiler. At the highest level of use, CENTRAN programs can be as machine independent as those written in PL/1.

The extended language of CENTRAN approximates PL/1 in control structure and FORTRAN in data structure. In addition to the control structure of PL/1, CENTRAN has CASE, BREAK, and ITERATE statements. BREAK allows a program to exit a DO loop or group gracefully (without use of a GO TO statement); ITERATE causes the next iteration of a DO loop or group to begin. The data structure is similar to that of FORTRAN except that there are based variables, simple structures, and partial word variables. The base language has been described in Ref. 1.

II. LANGUAGE DESIGN PROCESS

The many factors which control the design and implementation of a language can generally be classified into four groups, the designer having increasingly greater control over the resolution of the factors in the later groups.

The four groups are: environmental factors (external resources and constraints), general design issues (decisions to be made based directly on environmental factors), specific design issues (decisions of a topical nature to be made based on the resolution of general design issues), and implementation issues. The resolution of the issues posed in earlier groups are factors in the resolution of issues in the later groups.

2.1 Environmental factors

This group consists of factors over which one generally has little or no control.

2.1.1 Necessity for a new language

First, there is the basic presumption that yet another language is necessary. The need for a new language hopefully arises from external considerations, rather than out of some inner need of the designer or as a result of the "not invented here" syndrome. There must be good justification for designing a new language rather than choosing all or part of an existing language.

It was clearly desirable to write at least some of the SAFEGUARD software in a language higher than machine language. There were many cases in which the possible inefficiencies in code generated by a compiler could be tolerated. There were also many cases in which it was desirable to produce working programs inexpensively, regardless of the cost in running time and core, e.g., drivers and other test programs. Furthermore, if assured of good programming leverage (object-to-source-code ratio greater than one) from a language, and concise generated code from its compiler, it would be desirable to write all software in that language.

In the SAFEGUARD project, the compiler for the existing high-level language, NICOL, was unstable, and it was felt advisable to develop a language intermediate to NICOL and the assembler language as insurance. Selling CENTRAN as an "intermediate level" language (rather than a high-level language) avoided the presumption of NICOL's demise and avoided promising too much prematurely.

2.1.2 Manpower and implementation schedule

Two rigid constraints on the implementation of a language in an industrial environment are the manpower available and the implementation schedule: PL/1 cannot be implemented on a FORTRAN budget. Furthermore, the feasibility of using a high-level language must be proven before a commitment will be made to the implementation. A working compiler, with programs written in the language, is the most persuasive proof of feasibility.

For CENTRAN, the requirement existed to produce something useful within six months because the project was well under way and user software development could not wait. Only two full-time people and one person half time were available for design and implementation. There was no promise of increased manpower or lengthened schedule. Only one of these people had previously designed and implemented a compiler. It was necessary that the structure of the compiler be clean enough and simple enough for the available manpower to implement. The extendibility features of CENTRAN played a role here in assuring that the basic structure of the compiler could be implemented in a short time. Using the SWAP[2] macro facilities to write the compiler also contributed to the quick implementation of the language.

Within three months, a skeleton compiler was written that was able to successfully compile sample programs with which to show the feasibility of CENTRAN. A computer listing can be powerful magic, even among the initiated, and compiler development support was soon forthcoming.

2.1.3 Hardware

The hardware on which the programs are to be run is more of a constraint in the design of a language than is usually realized. Going from one generation of hardware to another has revealed machine dependencies and influences in language design. It has often been said that there should be more feedback to hardware design from language design, but until the state of software technology reaches that of hardware technology, hardware will be a fixed factor in language design.

The language designer has the final word on how the hardware appears to the user. He has the satisfaction of knowing that one purpose of a computer language is to compensate for "errors" in hardware design, such as to make the machine seem more uniform in structure than it actually is or to make explicit by syntactic equivalence the classes of machine operations. For example, the designer may use "+" to add a constant to a variable as well as to add two variables, even though the "+" may be implemented as two different machine operations.

The SAFEGUARD Central Logic and Control (CLC) computer was the target machine for CENTRAN. At a low level, CENTRAN supplied a uniformity to the CLC instruction set that did not in fact exist. For example, there were no machine operations to move data from certain registers to others without first moving the data to an intermediate register. CENTRAN "completes" incomplete data paths by generating the appropriate code. Of course, at the highest level of CENTRAN use, no references to hardware registers are necessary.

2.1.4 Software environment

The degree to which the software environment (e.g., loaders, binders, and operating system) is a fixed factor may affect the mechanics of program production and perhaps even the design of the language itself.

At the time CENTRAN was being designed, a large body of support software already existed. It was tedious matter to reassemble all SNX programs each time the object module format changed, and so it was decided that CENTRAN would conform to SNX object module format. As a result, certain desirable language features could not be included (e.g., multiple location counters) because they could not be represented in the object module.

2.1.5 User population

The two attributes of the intended user population, programmer proficiency and programmer background, affect the design of the language. For CENTRAN, the user population (in addition to Bell Laboratories people) consists of several subcontractors. The programmers exhibit a wide range of ability and experience.

Programmers have an emotional investment in the first language they learn; it is difficult to teach a programmer a second language. On the SAFEGUARD project, most of the experienced programmers were assembly language programmers and had a strong bias toward writing in machine code. This phenomenon has been noted in a more general context by Weinberg.[3] CENTRAN attempted to make the transition to a high-level language more palatable by keeping the machine accessible if so desired. The assembly language, SNX, is actually a proper subset of CENTRAN.

CENTRAN may have made the transition to a high-level language too easy—some programmers still think in machine language when organizing their programs, leading to a potential rigidity of structure and lack of language leverage.

2.2 General design issues

While the environmental factors generally are not under the control of the language designer, some degree of design creativity can be expressed in the resolution of the general design issues. These issues are: whether to create a new language or adapt an existing one, what the degree of machine independence and the language level are to be, how important ease of learning and ease of use are, whether the language should in some sense be "complete," and whether the language design should express present technology or the state of the art.

2.2.1 Creation of a new language or adaptation of an existing language

In determining whether to create a new language or adapt an existing one, the designer must beware of contracting either or both of two diseases: the "not-invented here" and the "it's-more-fun-to-design-my-own-language" syndromes.

In the case of a language for SAFEGUARD, the language compiler for NICOL 3 was found to be nonviable. An alternative seriously considered was to code, debug, and unit-test all programs in PL/1 using IBM computers and then to hand-compile the programs into SNX so that they could run on the CLC. This may well have been the course taken if CENTRAN had not been produced on schedule.

There was, however, an "almost existing" low-level language, the CLC assembly language SNX. It included the SWAP macro facilities, possibly the most sophisticated in existence (see Ref. 2), most of the interfaces with the operating system, and an object module generator that almost met requirements. By building on the existing SNX assembler, the designer and implementers gained a certain built-in compatibility with existing SNX SAFEGUARD programs, familiarity with the format, and most important, because of manpower and development-time constraints, free maintenance. However, the approach lost block structure (since the assembler did not have it), efficiency with respect to compile time (since the macro facility is completely interpretive), and control over lexical analysis.

Thus, an existing assembler was used as the base language for an extendible compiler. This allowed maximum use of existing software.

2.2.2 Degree of machine independence and language level

The two concepts, language level and machine independence, although related, are not equivalent. The language level is best described in terms of the degree of clarity and conciseness possible. Machine independence is usually defined in terms of the degree of portability of a program written in the language, i.e., how easily a program may be transferred from one machine to another. A language may be very machine dependent and of a high level.

Since there were no plans for successors to the SAFEGUARD system, machine independence was not a major factor in the design of CENTRAN. The level of the language, however, was a factor. As was mentioned in the discussion of the environmental factors, at the time CENTRAN was being designed there was a perceived need for an intermediate-level language. At the same time, it was apparent that certain high-level language features would soon be needed. CENTRAN's extendible design made it feasible to satisfy both of these requirements.

2.2.3 Ease of use and ease of maintenance

A language may be constructed with consistency, uniformity, and good debugging features, all of which makes it easy to learn the language and to write programs. Languages of this sort are ALGOL 68 and SNOBOL 4.

Program maintenance is aided if the purpose of a program written in the language is easy to comprehend, even though the syntax and semantics are nonuniform. Languages of this sort are PL/1 and FORTRAN.

Are ease of use and ease of maintenance related? Programs may be easy to write but incomprehensible once written, e.g., programs written in PAL, QED, and APL. Programs may be difficult to write but easy to read once written and debugged (e.g., FORTRAN, PL/1, and COBOL). Programs may be difficult to write and difficult to maintain (e.g., machine language programs and IBM JCL).

Another aspect of ease of maintenance that should be considered in language design involves binding time: binding addresses to variables and programs, disc locations to files, and generated code to source statements. In general, the later binding occurs, the easier programs are to maintain. "Patching" is usually easier, as is having independent compilation of subroutines and independent order of compilation. Late binding does, however, increase the cost in link, load, or run time. In CENTRAN, since the object module format was fixed, the language designer had no control over when binding was to occur.

2.2.4 Present technology or state of the art

A decision is made, unfortunately often only implicitly, as to whether the language is to advance the state of the art in language design and implementation or is to represent what present technology can accomplish.

Why design a language if it is not state of the art? Often, there is no need to invent a new language merely to fulfill user needs for a special-purpose language. It may be sufficient to select those features which are needed from existing languages. In a production environment, due to schedule constraints and caution on the part of management, state-of-the-art language may be considered undesirable. A state-of-the-art language and compiler represent more of a design investment and more of a risk.

CENTRAN was never sold as state of the art. However, CENTRAN still had to be implemented as an extendible compiler so that incremental implementation would be feasible. There was no time to do anything else.

Extendibility allowed the circumvention of general design issues by delaying their resolution, possibly indefinitely. If the language is not sufficiently machine independent, extend it to a machine-independent level and code only at that level. Completeness? Extend it as necessary. Efficiency? Start from the machine language; what could be more efficient?

Except for the extendibility features and treatment of machine registers, the extended CENTRAN language is not state of the art. Of course, the extendibility features of the base language, register allocation, and subroutine interface primitives may be considered state of the art, but the average user does not use these features.

2.3 Specific design issues

Specific design issues include: control structures, data structures, program-development features (e.g., tables of variables and attributes, listing format control), and extendibility features (e.g., programmer-defined subroutines, functions, macros, and data types) to be incorporated into the language.

The model chosen for the extended language for CENTRAN was PL/1. It is believed that this was the best decision, provided that a new language could not be designed from scratch. However, there are several reasons why ALGOL 68 (see Ref. 4) would be a better choice as the model. (It should be noted that the ALGOL 68 Report was not available when CENTRAN was being designed.) Perhaps the most important reason is that "expression languages" (in which most statements, as well as what are commonly thought of as expressions, return values and can occur anywhere expressions can occur) can allow the programmer to express himself in a degree of clarity not possible in other languages. Furthermore, an expression language is especially desirable for efficiency and clarity if the compiler does not do any common subexpression analysis, and the language gives the programmer access to hardware registers for the purpose of improving efficiency.

In particular, one of the results of modeling the extended language on ALGOL 68 would have been the choice of distinct representations for equality comparison and assignment. Then assignation could return a value, facilitating, for example, the use of register variables.

System macros (a set of utility macros used, for example, to interface with the operating system) were SNX-style and should have been CENTRAN-style. While implementation of a CENTRAN representation for all system macros was vetoed as not worth the effort, program bugs were induced by syntactical and semantic nonuniformities.

No thought was given in language design to program patching. Patching on the CLC was necessary, primarily due to the logistic problems involved in recompiling programs on the IBM machine and transporting them to the CLC. Little thought was given to data reduction because there were no requirements specified at the time. Requirements for patching and data reduction should have been considered. We pay the piper: for patching, one must patch in SNX or recompile; for data reduction, few symbolic data structures are allowed.

On the positive side, in addition to permitting the compiler to be built quickly, the extendibility mechanism confers additional advantages. The extended language was planned so that extensions could be made to semantics rather than syntax. Some documentation for the extension is free, since description for new syntax is not required. Some user education is free when new semantics can be associated with old syntax.

Extending a language is trivial if all extensions consist of new syntax not meant to interact with old syntax. That is how some language designers and users of extendible languages extend a language. The difficulty is to maintain uniformity, especially when the extension is not orthogonal to the old language. The classic problem here is to add complex arithmetic to a language, extending the semantics of the existing arithmetic operators, rather than creating new ones. Reference 1 describes how this may be accomplished in CENTRAN.

2.4 Implementation issues

2.4.1 Compiler speed and degree of optimization

There always seems to be a trade-off between the speed of a compiler and the optimality of the code produced. In an academic environment, where there are many student jobs, there are many compilations and few executions. In that case, a fast compiler designed without regard to object code efficiency is acceptable. In a production environment, presumably little time is spent in compilation in comparison to the execution time for production programs. Here, highly optimized code is desired.

One way to circumvent making the trade-off is to write two compilers, but this introduces obvious problems, not the least of which is potential incompatible language implementations.

CENTRAN is a slow compiler. This is due primarily to its interpretive nature. While some performance improvements were made after the compiler was written, stability requirements outweighed compilation speed requirements, and extensive improvements have not been made. The lesson learned is that if a program works, it is not likely to be rewritten just to improve its efficiency.

The design goal for CENTRAN was to optimize on the statement level only, producing the best code possible for statements such as "$a = b$ operation c," where a, b, and c are simple variables. Sufficient manpower to produce a global optimizer was not available. Users would rather have more features in CENTRAN than have a globally optimized program. The local optimization design goal of CENTRAN was achieved, leaving global optimization to the user (aided by effective counseling).

Since the expression parser produced nonoptimal code, users were warned against using complex expressions if they had severe running time or space constraints. This was done also to protect the implementers against the possible wrath of users complaining about inefficient code. However, the lack of optimization of code produced by the expression parser was oversold, and programmers get much less leverage from CENTRAN than they could.

2.4.2 Compiler structure

After the questions of degree of optimization and speed of the compiler are resolved, there remains an issue that is the primary concern of the designer: compiler structure. Related to the structure of the compiler is the question, "In what language should the compiler be written?"

Several alternatives were considered in the implementation of CENTRAN. First, as indicated earlier in the discussion of environmental constraints, it certainly was not feasible to create a language completely independent of SNX. There were no resources to implement a new output-module generator, interfaces to the operating system, and machine-operations listing. The compiler at least had to be assembler-ended; the output of CENTRAN had to be an input to the SNX assembler. The question then became that of the degree of interaction between the compiler and the assembler.

Why was CENTRAN not implemented as a preprocessor to or a co-routine with SNX? The answer is that it was not clear at the time how the interface could be achieved. It still is not clear that this can be done successfully. The assembler was not designed to interface externally with a language processor. Other problems to be considered include the possibility of duplicate symbol tables, duplicate language processing, the loss of the macro facility, and the introduction of nonuniformities.

A compiler-compiler was not used to implement CENTRAN because there was none available and creating one would have meant maintaining two languages.

The method of implementation of CENTRAN consists of a combination of recursive descent and precedence tables. The arithmetic, logical, and relational expression parsers are driven by precedence tables; everything else is recursive descent with a vengeance. All the statements generated by the compiler (even those generated by the table-driven parser in the expansion of a CENTRAN statement to machine code) are legal CENTRAN source statements. There is no "canonical" intermediate-level language inaccessible to the user of the extended language. Each machine operation is (textually) generated in only one place. All CENTRAN code generating statements are eventually expanded into a set of CENTRAN statements, each generating exactly one machine instruction.

III. LANGUAGE USAGE

3.1 Who is using the language?

CENTRAN is the official language for the SAFEGUARD project. Except for programs which had been written in assembly language before the availability of CENTRAN (parts of the CLC operating system), all SAFEGUARD programming is done in CENTRAN. Programmers may not use machine language without management approval. No cases are known where it was necessary to "drop down" into machine language. In a large sample, no programmers had machine language interspersed.

3.2 How are the extendibility features being used?

As might be expected, most extensions are made in terms of macros used to generate CENTRAN syntax. Some programmers, however, have extended the language in data structures, where it is weakest.

IV. CONCLUSION

4.1 The designer-implementer-educator-user relationship

From our experience in the development of the system, we can draw several conclusions that might be helpful to others. We as designers along with the implementers, educators, and users should not be disjoint groups. We should be involved as an implementer to keep in touch with reality. We should also be involved as an educator (if a feature is difficult to explain, maybe there is something wrong with it), and a user (uniformity in extension is best achieved by knowing how language is being used). The implementer should act as both educator and program counselor to get feedback on bugs being "programmed around" and to establish priorities for fixing them.

Several things about the implementer-user relationship should have been learned earlier in CENTRAN development. First, the release cycle should be rigidly controlled as soon as possible, no matter how short the cycle. It does not pay to give fixes to bugs informally. Next, old versions of the compiler should not be kept around and certainly not maintained. The maintainers are blamed for bugs that no longer exist, and much time is spent rediscovering causes for problems long since resolved.

Notices of new releases must go to everyone, not just supervision. Users often underestimate the impact on schedules of changes due to improvements to the compiler, even though the improvements were requested.

Insofar as the designer-implementer-educator-user relationship is concerned, we, as designers, should have contributed more to the structure and content of the CENTRAN courses. Frequent symposia (e.g., "Advanced Topics in CENTRAN Programming") should have been held, with compulsory attendance.

4.2 Lessons learned

Most of what has been learned in the design and implementation of CENTRAN has been covered in previous sections. Some of the more critical aspects are worth reiterating.

CENTRAN should have been an expression language. This would not only have aided the production of more efficient, clearer, and more concise code, but would have provided a greater degree of uniformity to the language.

We should have given more thought to data types required for data reduction. Maintenance of CENTRAN programs (especially patching) should have been given greater priority in the design of CENTRAN.

Variability in the backgrounds and experiences of programmers should have been anticipated. Not enough consideration in the design of the language was given to the characteristics of the user population, and not enough emphasis was placed on continuing education.

Several of the CENTRAN design approaches were advantageous. CENTRAN was implemented by a small group of programmers. This approach avoided communication and other problems typically encountered in a large group of programmers.

The register allocation mechanism, subroutine interface primitives (Ref. 5), and extendibility mechanism designs worked well, as exhibited by CENTRAN's short development time. The ability to have partial word variables has been found useful. The structured programming features have been used extensively. The ability to program at several levels in one language made the language suitable for systems and applications programming. Finally, and most important, the design of the extended language is sufficient for the implementation of SAFEGUARD software. The SAFEGUARD programs have been successfully implemented in CENTRAN. Several studies of the suitability of CENTRAN for SAFEGUARD have been made outside of Bell Laboratories, and all have arrived at positive conclusions.

REFERENCES

1. B. N. Dickman, "ETC—An Extendible Macro-Based Compiler," Proc. AFIPS SJCC, *38* (1971), pp. 529–538.
2. M. E. Barton, "The Macro Assembler, SWAP—A General Purpose Interpretive Processor," Proc. AFIPS FJCC, *37* (1970), pp. 1–8.
3. G. M. Weinberg, *The Psychology of Computer Programming,* New York: Van Nostrand Reinhold, 1971.
4. B. J. Mailloux, J. E. L. Peck, and A. Van Wijngaarden, "Report on the Algorithmic Language Algol 68," Kibernetica, *1,* 1970.
5. B. N. Dickman, "Subroutine Interface Primitives for ETC," ACM, SIGPLAN Notices, *7,* No. 12 (December 1972).

SAFEGUARD Data-Processing System:

Systems Programming in PL/1

By P. A. VAN SCIVER

(Manuscript received January 3, 1975)

This paper deals with the development of a large systems program in a high-level language. The reasons for selecting a high-level language, the most extensively used features, the benefits derived, and the significant problems encountered are described.

I. INTRODUCTION

This paper highlights the important aspects of developing a large systems program in a high-level language. The Execution Preparation Facility (XPF) performs the linkage editor function on the SAFEGUARD project. When XPF was originally designed, the decision was made to develop it in PL/1. The paper examines the most extensively used features of PL/1, describes the problems encountered during development, points out the lessons learned, and discusses the benefits derived from the use of a high-level language. An appendix provides XPF development productivity data and comparisons.

II. FUNCTIONAL DESCRIPTION

XPF is the last major step through which software must pass on its way to execution on the CLC. Some functions performed by XPF can be compared to those of the operating-system linkage editor in that XPF prepares the output of the language processor for execution, sets up the overlay environment, and produces memory maps and cross-reference listings.

The output of XPF, called a thread, is a collection of programs and data sets and their associated control tables bound to absolute addresses. The thread also contains installation, debugging, and data reduction information. Inputs to XPF are user-supplied commands, execution time parameters, assembler or compiler output, a partitioned data set called the system file that describes the CLC operating system, and, in an update mode, the results of previous XPF runs.

The major functions of XPF are construction of control vector tables (CVTs) for interthread linkage, allocation of CLC resources, primary memory and disc storage, binding of thread units, and construction of operating-system control tables (PCTs). In addition, XPF produces a series of printed listings describing memory configuration, process structure, forward and back referencing among units, PCT construction, and thread summary data.

III. PHYSICAL DESCRIPTION

XPF consists of 246 subroutines, 95 percent of which are written in PL/1. The internal structure is modular. Functions are performed by 24 independent modules that overlay each other. The XPF load module consists of 130 overlay segments and requires 2.5 megabytes of disc storage. The access method used to retrieve object code, while not actually a part of XPF, is also included in the XPF load module.

XPF operates in a 400-K region, of which 260 K is occupied by the overlaid load module. During execution, 12 internal files are used for work space and intermodule communication. Since the disc space needed for these files varies with the input, space allocation is controlled by catalogued procedure parameters. The actual execution of XPF is controlled by the execution time parameter field on the user's JCL execute card. Most modules execute at the option of the user and are controlled through this field. The mode of execution (regular, debug, or update) is also directed by execution time parameters.

IV. DESIGN DECISIONS

Since most systems software is written in assembly language, one question arises: Why was a high-level language used for this facility? Three major factors contributed to this decision.

(*i*) Development time was short. It was felt that the anticipated ease of writing in a high-level language, coupled with extensive utilization of compiler-provided debugging capabilities, would help provide the desired results within the allocated time. This proved to be the case, and each of ten XPF releases was produced on schedule.

(*ii*) A high degree of flexibility was required. XPF, the operating systems, and the applications processes were to be developed concurrently. Since XPF is the software that links the operating systems and the applications processes, responsiveness to the design requirements of both groups was a necessity. A high-level language was judged to be best equipped to provide the required flexibility. This approach proved valid. In practice, when a SAFEGUARD design problem could have been solved by changing the CLC operating system, the applications processes, or the XPF, XPF was usually chosen.

(*iii*) The execution of XPF was expected to be I/O limited. Therefore, potential compiler-generated CPU inefficiency was not a major consideration.

Since XPF was to be developed and executed under OS/360, the contenders for a high-level language were PL/1, FORTRAN, COBOL, and ALGOL. PL/1 was an easy choice. The bit-handling capabilities of the language were well known, and many members of the development group had PL/1 experience.

V. HOW PL/1 WAS USED

This section records those features of PL/1 used most extensively in the development of XPF.

External variables were used to store relatively small amounts of data needed throughout XPF execution. Since the external variables were located primarily in the root segment of the load module, their use in intermodule communication aided in segmentation and structuring of the overlay tree.

Static storage was used extensively to take advantage of what would have been dead space in the short legs of the overlay tree. The judicious use of static storage minimized the amount of memory required for execution. Static variables require special attention in an overlay environment. Every time a segment is brought into memory, each static variable is reinitialized, but in subsequent calls to the segment that do not require overlay, the variables retain their current values.

Three types of I/O were utilized. Stream-oriented I/O was used for printed listings and debugging output. Sequential-record-oriented I/O was used for intermediate files for communicating between, at most, two modules. The TITLE option was used with these files to allow many modules to utilize the same disc area, thereby reducing overall resource requirements. Regional I update files were used to satisfy global communication requirements, e.g., paging of data and storage of object and bound units.

Area variables were utilized by many modules. Each record entered into the update files consisted of a single area variable. Individual data entries were allocated within the area and entry addresses assigned. The use of areas avoided excessive I/O by allowing large amounts of data to be stored on a single record. The utilization of PL/1 area management greatly reduced the amount of user-supplied code necessary for record formatting and control mechanisms.

In the shorter legs of the overlay tree, area variables declared with the static attribute were employed, realizing the advantages described earlier.

Based variables were used extensively, especially in the areas of I/O. Based structures were declared in the calling programs, and file managers returned pointers to the requested items.

List processing was a major requirement in XPF design. Frequent sorts of these lists were required. The use of linked lists prevented excessive data movement during sorts, since only the pointers needed to be modified to change the order of the tables.

The bit handling features of PL/1, an important aspect of the decision to use this language, were used extensively. Since the CLC uses ASCII character representation, characters had to be interpreted as bit strings.

Label arrays were utilized in command processing. Since many commands contained common keywords and fields, processing was broken down to that level. Keywords and fields were interpreted and assigned number values that were used as indices into label arrays for keyword processing.

The PL/1 preprocessor played an important role in XPF development. Preprocessor statements and procedures were placed on a file that was accessed via "% INCLUDE" by all procedures. Four key functions were performed by the preprocessor:

(*i*) Declarations of global data such as external variables and based variables used in I/O were stored on the file and brought into each procedure that utilized them. This assured identical variable declarations throughout XPF.

(*ii*) Declarations of utility and file manager entry points and their associated parameter attributes were also placed on the file. This helped assure the consistency of parameters passed to these subroutines.

(*iii*) Certain constants such as area sizes, array dimensions, and conversion constants were subject to frequent change while optimal values were being ascertained. Programs referencing these constants did so via preprocessor variables. When modifications were necessary, the values of the preprocessor variables were changed on the file and the referencing programs were recompiled.

(*iv*) Preprocessor procedures were provided for frequently used in-line code.

VI. HOW ASSEMBLY LANGUAGE WAS USED

While 5 percent of the subroutines in XPF are written in assembly language, these amount to less than 0.2 percent of the total number of machine instructions. Assembly language subroutines fell into two categories: data conversion subroutines originally written in PL/1 and recoded in assembly language for reasons of storage economy or efficiency, and subroutines written in assembly language to provide facilities not directly available in PL/1.

An example of the first is a TRANSLATE function that converts ASCII to EBCDIC, and vice versa. This function was not supported in PL/1 Version 4. By recoding in assembly language, a 20-K-byte subroutine was reduced to 500 bytes and made much faster.

An example of the second is a routine to access a partitioned data set of twenty or so members. Had this routine been written in PL/1, one DD card for each member of the data set would have been required.

VII. MAJOR PROBLEMS ENCOUNTERED

The most serious problem encountered during development was an obscure but critical bug in object code generated by PL/1 Version 4 that became important when a new computer with a larger memory was installed. XPF would ABEND if loaded in the upper third of memory because of bad code generated for bit-string operations. This made it necessary to convert XPF to Version 5 of PL/1. Incompatibility between these versions required complete recompilation and some recoding. Six weeks of effort were required to complete the conversion.

Another major problem was directly related to this conversion. Half-word storage, implemented in Version 5, caused structure alignments to be altered. Since boundary alignments were not required on the development computer, some problems were not detected. It was later discovered that XPF would not work on certain models of the IBM System 360. The most expedient method of correcting the problem was to declare the offending structures unaligned. Portability of XPF could probably have been ensured in advance by constantly being aware of the consequences of PL/1 defaults.

The XPF execution problem causing the most impact was excessive I/O usage generated by the OS overlay manager and not by PL/1. Dramatic reduction in load module accesses was accomplished by overlay restructuring.

VIII. LESSONS LEARNED

In addition to the initial decision to use PL/1, throughout the development of XPF many design and implementation decisions concerning the use of PL/1 were made. Some of these proved to be sound, and others had unfortunate results. This section deals with the results of these decisions.

The extensive use of the PL/1 preprocessor proved to be an excellent control mechanism. The inclusion of macros, entry point declarations, and global variable declarations via preprocessor procedures greatly facilitated intermodule communication. This standardization guaranteed the integrity of interfaces.

As originally expected, the liberal use of PL/1 debugging aids was an invaluable development tool. The large number of logic errors detected through ON conditions such as SUBSCRIPTRANGE and STRINGRANGE underlines the value of their use.

PL/1 provides no debugging aids for pointer variables, used extensively in XPF, so it was frequently necessary to examine a dump to ascertain the exact nature of a problem. Since no error control philosophy within XPF had been established, dumps could not be produced at will. A global error control mechanism was instituted. By placing a single ON ERROR block in the main procedure and removing them from lower-level subroutines, the problem of inappropriate or inadequate response by these subroutines was eliminated.

No global coding conventions were established at the beginning of the project. This resulted in various methods of implementation of the same basic requirements, some of which were more efficient than others. A subset of PL/1 should be extracted that is most efficient for the particular application. Programmers should be warned to avoid certain implementation methods and encouraged to use other more efficient ones.

Since XPF was required to execute in a 400-K region (the maximum size for an express run), the use of small independent subroutines that could be overlaid was encouraged. In the longer legs of the overlay tree, this philosophy proved valid. However, in the shorter legs of the tree, this introduced unnecessary inefficiencies because of operating system overhead. The increased use of static storage in the shorter legs decreased the effect, but the use of fewer subroutines would have been more efficient.

The use of assembly language subroutines, though dictated by reasons of efficiency and necessity, presents some disadvantages. Since parameter definition is compiler-dependent, assembly language subroutines must be coded to meet the parameter passing standards and conventions of a specific compiler. In PL/1 these proved even more limiting since assembly language subroutines must be coded for a specific version of the compiler. When such subroutines are utilized, this dependency on a particular version of a compiler should be explicitly documented.

The assembly language complications are the most obvious of the compiler dependency problems. However, as noted previously, incompatible compiler versions, the resulting recompilations required, and possible machine-dependent errors are also problems. Unless a private, unchanging compiler is used, time must be reserved in the development schedule for this type of updating activity.

IX. DISCUSSION

Flexibility was one key factor in the decision to use a high-level language, and it proved to be one of the primary assets of the development technique. Since XPF was written in PL/1, it could be fine-tuned with less effort than if it were written in assembly language. Sections of code could be rewritten in a relatively short period of time. This made it feasible to experiment with implementation methods until optimal code was produced.

One benefit of development in PL/1 that was not considered in the original decision was the ease with which transfer of responsibility is accomplished. Partial turnover of personnel occurred throughout the project. The transfer of code responsibility to new personnel was accomplished very smoothly with no apparent decrease in productivity. Since PL/1 can be largely self-documenting through the use of meaningful variable names and standard operation symbols, it is easy to read and understand. This ease of understanding was the primary reason for the smooth personnel transitions.

Perhaps the most important advantage of developing a system in a high-level language is that the compiler provides area management, storage allocation, error control, data access, and I/O interfaces. The programmers can devote their time to acquiring expertise in the unique requirements of the system.

APPENDIX

Over a period of two years (by Release 8), XPF grew to approximately 32,000 PL/1 plus assembly-language statements. Almost all the

Table I — Comparative productivity, Release 9

	SNX Assembler	CLC Simulator	XPF
Total no. of subroutines	72	90	231
Total no. of source statements	69,788	63,737	34,344
No. of subroutines added or changed	40	30	84
Percent of total subroutines	55.5	37.7	36.3
No. of source statements added or changed	3,286	2,336	7,342
Percent of total source statements	4.7	3.6	21.3
Man-months programmer, management, librarian	23.5	10.5	37.0
Statements per man-month	140	222	198
Man-months, programmers only	18.4	9.0	30.0
Statements per man-month	177	259	244

changes for each release were planned increases in capability, although some, of course, were fixes for bugs. The total effort to produce the first eight releases, debugged and tested, was 222 man-months. The average productivity over this time is therefore about 140 statements per man-month.

Table I compares Release 9 of XPF to the corresponding releases of the SNX assembler and the CLC simulator, both written in assembly language. The simulator was a considerably easier task than XPF for this release because the simulator was only receiving maintenance, while 21.3 percent of XPF was rewritten to add major new capabilities. Nevertheless, the total number of machine instructions produced (per man-month) by the XPF group was greater because they were coding in PL/1, whereas the assembler and simulator groups were coding in assembly language.

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

SAFEGUARD Data Reduction System

By E. S. HOOVER and R. A. JACOBY

(Manuscript received January 3, 1975)

In evaluating and certifying the SAFEGUARD ABM system, it was necessary to interrogate and analyze the massive volume of data generated during tests. The number of different reports, listings, and plots was so large and the variety so great that a flexible data reduction system had to be placed at the disposal of the user community. At the same time, the system had to be highly efficient and quickly available to be used at all. The SAFEGUARD Data Reduction System was designed to accomplish these objectives.

I. INTRODUCTION

Since any operational system must be tested, certified, and evaluated, provision of the means for doing so must be part of its design. The first step in certifying that a process is performing as specified is the recording of certain significant data during test runs. These data must be reduced and presented in a variety of ways. The SAFEGUARD Data Reduction System (SDRS) fills this role by providing a flexible and highly efficient facility to serve the needs of the test teams.

The fundamental capabilities of SDRS had to be available when the testing began. The design for the real-time recording programs and the data reduction programs had to be coordinated, since the recording program serves as input to the reduction program. Short reduction program development schedules made necessary the use of certain preexisting designs and code, which had to be worked into the resulting system without compromising the other requirements. To accomplish this, a number of deliveries were planned, starting with the simplest and most basic features. Users who tried the first system were able to give SDRS designers useful feedback for future deliveries.

This paper discusses the experience gained in formulating requirements, organizing the program, developing the facility, and interacting with users.

II. REQUIREMENTS FOR THE SAFEGUARD DATA REDUCTION SYSTEM

To test, certify, and evaluate the behavior of a SAFEGUARD process, it is necessary to record data from memory during the real-time execution of the process. Debugging and integrating can conceivably be accomplished by stopping the process and taking post-mortem memory dumps, but this destroys the real-time sequence of events. Since the SAFEGUARD processes contain hundreds of thousands of lines of code, testing would be exceedingly cumbersome if it were necessary to stop the test to record data. Because calls to the recording subroutine are planned for and remain always in the code, a wealth of internal data can be recorded without disturbing the real-time behavior of the process. Recording occupies some CPU time but, aside from this effect, the process performs in the same way whether or not recording occurs. Real-time recording is essential for a practical testing program.

2.1 Requirements for recording

Thousands of events for which data might be taken occur during a test. The specific data needed depend on the purpose of the test. The CLC operating system[1] allows the applications process to record up to 100,000 32-bit words of data per second, as many as eight reels of tape during a 16-minute test.

Unless care is taken in recording design, even this large capacity can be exceeded. Designers tend to do more recording than is needed. This happens because the recording decision must be made long before testing begins. By recording almost everything, designers protect against overlooking some data items that may be wanted. As a result, a burden is placed on data reduction to select from a large mass of data only those items the user needs.

The data as recorded by the CLC operating system are organized into physical records of variable length. Each physical record contains a header and one or more logical records. The header preceding each logical record categorizes the data to follow. Records of one type would contain the most common and essential items, while records of another type might contain more voluminous data.

2.2 SDRS requirements

Consideration of the content, structure, and volume of the input and the expected use for the output dictate requirements for the data reduction system. Very large quantities of data are recorded from over 1000 different data sets.

Many data structures are implemented in the processes, but four typical ones were selected for processing by SDRS. Many more complicated structures, if properly recorded, can be handled by SDRS.

To be correctly interpreted by SDRS, the physical attributes (floating point, integer, etc.) of each item of recorded data must be defined. Since a majority of these items must also be defined for CENTRAN compilations, SDRS avoids possible incompatibilities by accessing the CENTRAN declarations.

To reduce data items not defined in CENTRAN and to allow quick response to patches in the CLC applications processes, SDRS also provides utilities to define and modify attributes. Experience with earlier data reduction systems, which provided only manual methods of data attribute definition, led to the requirement for both automated and manual methods.

There is a need to format the data so that they may be interpreted with ease. Some factors to be considered are discussed here.

Ease of interpretation of the recorded data requires that methods be supplied for selection of only the necessary subset of the data for presentation. This allows the user to generate exception reports and summaries rather than printing or plotting every data value.

Raw data may be recorded in one form, but they may be much more useful in another. Presentation in engineering units is often helpful. User-defined computations can be made by SDRS to facilitate evaluation of data.

In some cases, related data are scattered over a series of records and even over a series of tapes. The difficult task of correlation is performed by a file handler that can associate data for a user.

Some forms of presenting data are more useful than others. Since it is impossible to predict what particular listing or plotting form will best serve a given user, the users need the ability to format their own reports for presentation. Specifically, the users choose from among four basic ways to present data: formatted reports, tabular listings, line and point plots, and histograms. Users specify titles, subtitles, column headings, plotting axes, and scaling parameters for plots.

To gain user acceptance, SDRS (which is not a real-time facility) had to provide features that could satisfy quick turnaround time requirements with minimum effort.

Since most users developed their requirements as they began testing, SDRS had to provide users with plots and tabular listings which they could then easily modify to suit their later needs. SDRS provides users with a high-level command language in which to specify more complicated reduction requirements. Sufficient default conditions are supplied so that a simple set of user commands will result in a listing of all data items in a logical record.

One group of five users designed 737 tabular listings and reports in 12 months, averaging 12 per man-week. The elapsed time to get a simple tabular listing to work was about two days.

Because thousands of corrections had to be made to applications processes, there was a premium on quick turnaround time in processing the recording tapes. Since many testing teams submitted reduction runs simultaneously, it was essential that SDRS process a large volume of records efficiently.

It would be possible to reduce data on the CLC (this will be done during the post-installation and test phase). However, the CLC installations are limited in number, and time on the CLC is normally devoted to testing. Therefore, a decision was made early that data reduction should be done off-line on commercially available computers. Testing is performed for SAFEGUARD in several locations. In each of these locations, an off-line computer of the IBM 360/370 series is used for data reduction. The decision to use an off-line facility was correct. Time on the test machine is limited and is in much demand for the primary testing task.

A facility such as SDRS cannot be developed with all capabilities operational at the time an initial capability is needed by users. The designer can capitalize on this. If he designs a modular system and an open-ended user command language, he can first deliver a simple system. Feedback after the users have tried the first system can improve the design of later extensions. In fact, while users were presented with a proposal for SDRS and were invited to give their comments, most suggestions were obtained only after the first version of the system was working.

2.3 What was learned in setting requirements

The needs of users with a great many reduction requests to maintain were not foreseen. Since SDRS made it easy to request a large number of different reductions, the administration of requests required automation. One user group, supporting a single process integration, generated over 7000 SDRS statements. User groups usually solved this problem by developing their own administrative programs.

It was important to restrict users to one specific command language to give them the ability to turn out data reduction requests quickly. Some users, familiar with FORTRAN or PL/1, undoubtedly would have liked full control over program execution and the use of data types available in those languages. Restricting the number of data types supported increased the speed with which the system was developed.

III. SYSTEM ORGANIZATION

3.1 Design considerations

The development of a system organization and design philosophy for SDRS was influenced by a variety of factors. These included user requirements, schedule constraints, experience of the designers, and successes and failures of earlier systems.

The designs of several previous data retrieval and analysis systems were analyzed to determine their applicability to the system requirements. This analysis uncovered serious shortcomings in most of these systems for the high data volume requirements of SAFEGUARD; however, several common strong points were noted in each.

A general solution to the data correlation problem was attempted with the Mission Data Reduction (MDR) system, which was developed for use in the Meck test system. The significant features of this system were a command language user interface, general data sorting capabilities, general data conversion capabilities, and data presentation capabilities in the form of reports, plots, and tabular listings. The major shortcomings of MDR were its complete dependence upon sorted data to produce any output as well as a requirement to convert all data before selecting the subset of interest. Both these characteristics introduced exorbitant overhead for sequential processing.

A more specific approach to the problem was used in the systems that were successors to MDR. These systems added a data attribute dictionary, an efficient sequential-file data extractor, and specific data correlation capabilities in the form of special-purpose subroutines to the basic capabilities of MDR. The major shortcomings of these systems were limited selectivity during the extraction phase, a requirement to convert a large percentage of the total data before selecting the subset of interest, and limited file generation capabilities that required many passes over the raw data to extract all data of interest. An additional shortcoming was a dependence upon manual maintenance methods for the data attribute dictionaries.

A common factor in each of these systems was uncovered: The designers had little or no control over the format of the data files that they were required to process. In general, the files were written without regard to the eventual processing and correlation requirements. With few exceptions, these characteristics of the data to be processed introduced a great deal of complexity and overhead into the systems.

Once it had been determined that none of the available systems would meet the requirements adequately, a design was proposed that would provide an initial capability with incremental growth potential. The user community required delivery of the first release within nine months and incremental releases at two-month intervals.

To meet these schedules, it was necessary to achieve a balance between the development of new programs and the adaptation of existing programs. The advantages of short development time offered by use of existing programs had to be weighed against their extendibility, flexibility, and maintainability. In addition, the efficiency requirements for the data presentation capabilities had to be considered.

The requirements obtained from the user community were analyzed to determine possible commonality. An attempt was made to determine the general characteristics of the data that would be generated by the users. An estimate of the probable volume of data to be generated by these users was also made.

All users requested the same basic capabilities. These included printing reports in a variety of forms, correlating and sorting data on a variety of criteria, plotting data, and specifying the conditions under which this processing was to be done. The large number of special processing requests made it obvious that the development of a general-purpose facility was necessary.

3.2 Basic functional components

The system consists of four basic functional components as indicated in Fig. 1:

(*i*) The data attribute definition component defines the characteristics of data items by examining their CENTRAN declarations.
(*ii*) The sequential data base retrieval component provides data collection, selection, and presentation capabilities for sequentially organized data files.
(*iii*) The hierarchical data base generation component allows the relatively efficient creation of direct access data files.
(*iv*) The hierarchical data base retrieval component provides data collection and selection capabilities as well as sequential data base generation capabilities for direct access data files.

3.3 Lessons learned

The overall efficiency of SDRS is difficult to measure since the users of the system specify what the system must process. This introduces into the evaluation of SDRS performance such factors as user expertise, user knowledge of data characteristics, and user analysis of needs. Several design decisions were made to minimize the impact of these factors on system performance.

Provision of methods for the user to perform many data presentation operations on a single data retrieval pass was the primary characteristic of the design that provided efficient processing capabilities. This approach, although somewhat obvious for processing sequential data bases, is equally applicable to the processing of direct-access data bases. This is true because minimization of the number of times data are retrieved will minimize elapsed time and system overhead.

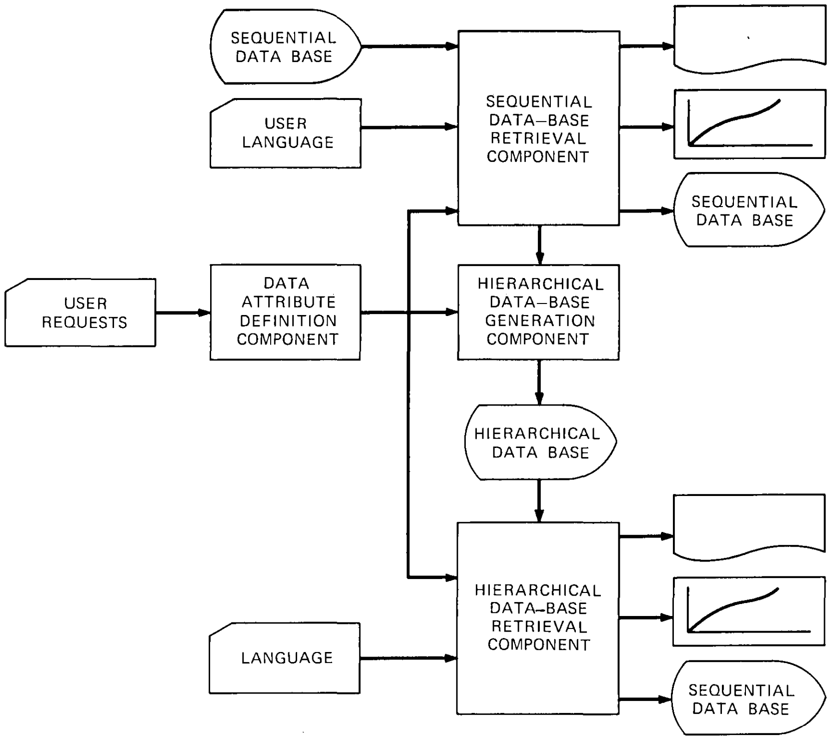

SEQUENTIAL DATA BASE
USER LANGUAGE
SEQUENTIAL DATA-BASE RETRIEVAL COMPONENT
SEQUENTIAL DATA BASE
USER REQUESTS
DATA ATTRIBUTE DEFINITION COMPONENT
HIERARCHICAL DATA-BASE GENERATION COMPONENT
HIERARCHICAL DATA BASE
HIERARCHICAL DATA-BASE RETRIEVAL COMPONENT
LANGUAGE
SEQUENTIAL DATA BASE

Fig. 1—Basic functional components.

Structured data recording makes it feasible to develop a simple efficient data-filtering algorithm. This algorithm enables SDRS to discard all records not requested by a user by interrogating fields in the header and ignoring all data in the record.

Limiting the number of data structures supported makes it feasible to design algorithms that extract and convert a minimum amount of data. Minimizing the number of data conversions was especially critical in the sequential data base retrieval module because of the large volume of data. In this module, data conversion is delayed until after all user conditions had been satisfied.

Assembly language was used in coding those critical paths of the system that would process large volumes of data. The extra time required to develop these programs was offset by the increased efficiency derived from this approach.

IV. DEVELOPMENT CONSIDERATIONS

The development of SDRS and the delivery of the system to the user community with capabilities consistent with the requirements were accomplished on short schedules. Several development procedures and techniques were used by the design group that may be applicable to other development efforts faced with similar problems.

The critical need of many users for a simple data-printing capability was the basis for the design of the first release of SDRS. This release consisted of the basic versions of the data attribute definition component and the sequential data base retrieval component.

Although simple from a user capability point of view, this initial release of the system was designed to be extendible. Emphasis was placed on development of a design that would allow inclusion of additional processing capabilities without major perturbations.

The development of an outline for further functional capabilities was begun in parallel with the development of the initial system. This outline served as a vehicle for planning capability development and delivery.

Formal design specifications for each system component were not written. However, detailed interface specifications were developed. This made it possible for individual routines to be designed in parallel.

The development of SDRS on schedule would not have been possible without the use of a time-sharing system. Although time sharing is relatively expensive, development times can be minimized when rapid correction of troubles and extensive testing are required.

The size of the system required that extensive testing be done to verify system performance. Although testing is possible in a batch environment, the effectiveness of the system test team was greatly enhanced by the availability of immediate test results and on-line debugging capabilities.

V. USER INTERACTION

Before SDRS was designed, users were asked to submit their requirements. The SDRS design group then proposed to users an initial set of requirements. Only after the initial release was meaningful user feedback received. Whenever possible, suggested improvements were incorporated into subsequent versions.

A system as complicated as SDRS requires user education. Two methods were used for this purpose: A user's manual was written and counselors were provided. The role of the counselors was to teach correct and efficient SDRS use and to collect feedback for improvements.

The final service to users is proper test and maintenance of the system. Users were not asked to be guinea pigs. They were allowed to try a new SDRS only after a complete set of tests were run. During two years of use, only 81 troubles were encountered in 105,000 lines of source. Because of the error messages and the modularity of the system, it was easy to identify and fix problems.

Total effort expended in user services has been 18 percent of the manpower of the SDRS group. This is considered a minimum effective support level.

VI. CONCLUSION

The primary lesson learned from the development of SDRS is that user data base design is critical. Recording and reduction efficiency is achieved by designing data bases to minimize the requirement for further correlation and restructuring.

The real achievement of SDRS lies in simultaneously accomplishing the objectives of flexibility and efficiency. Many systems attain one goal or the other: SDRS attempted to do both. Two design decisions contributed to the success of this effort. First, recorded data not wanted by the user are ignored by the system. Second, once data are retrieved, they are processed in as many ways as needed.

REFERENCE

1. J. P. Haggerty, "SAFEGUARD Data-Processing System: Central Logic and Control Operating System," B.S.T.J., this issue, pp. S89–S99.

SAFEGUARD Data-Processing System:

A Means to Effective Computer Resource Utilization

By J. P. KUONI

(Manuscript received January 3, 1975)

"Bird-dogging" is the process of tracking down computer center users who are either having problems and therefore are not getting their job done or who are using a disproportionate share of the computer's resources. Analysis of utilization data for the SAFEGUARD support computer centers has shown that the problems caused by these users can be of alarming magnitude, leading some observers to believe that bird-dogging is the single most effective system performance tuning activity that can be performed. Bird-dogging is an integral component in reliable project scheduling and effective cost control. This paper discusses the methods now used to identify problem users and some experiences gained from the effort.

I. INTRODUCTION

This paper describes the function of bird-dogging as the main tool for achieving the most efficient use of the computer. Specifically, through analysis of computer utilization data (which may be sampled on a daily, weekly, or monthly basis), the use of computer center resources and the problems of its users are monitored in detail. This is followed as needed with a program of counseling. The purpose of counseling is to better educate computer users to employ effectively the computing resources available to them (hardware, operating system, and application software). Counseling also provides feedback to the designers of application software to allow implementation of designs that would permit better utilization of the hardware and operating system features.

Some segments of the bird-dogging campaign are conducted on a daily basis for short-term gains, and other segments take the form of more extensive investigations yielding long-range gains. The latter activity more closely approximates the traditional system tuning.

Bird-dogging has been actively supported at several project support computer centers since the fall of 1971. Manpower allotment during this period is estimated to be two or three full-time technical staff members at each location. This total includes manpower employed to develop programs for automated report generation.

II. WHY BIRD-DOG?

Although many installations are committed to ongoing efforts in the traditional areas of systems performance analysis, few are engaged in bird-dogging campaigns. Why, then, are the SAFEGUARD project centers actively supporting this activity? There are two main reasons: schedule reliability and cost control.

First, schedule reliability. During the years of developing the system's software, timely completion of the hundreds of interlocking software modules has been critical for project delivery. It has been imperative, therefore, that everyone, even the below-average programmer, complete his or her responsibilities on time and successfully. To increase confidence in meeting project schedules, those who are unable to make it on their own must be helped.

Second, cost control. Bird-dogging helps reduce costs through short-term immediate benefits and long-range improvements. For example, bird-dogging usually produces immediate benefits by reducing the resubmittal rates of "problem" programs, which increases the turnaround potential of other programs competing for the limited computing resources.

In the long run, for example, many users having similar problems may reveal that the documentation of how to use a particular feature is inadequate. Following through on individual problems to gain insight into underlying causes is often worthwhile and carries considerable long-range benefit.

III. UTILIZATION DATA

To permit monitoring the center's users, several types of utilization data are obtained from a series of automated reports and other sources.

3.1 Automated reports

The bulk of bird-dogging data is generated by several special-purpose report programs developed by project personnel. Most of these programs use the System Management Facilities (SMF)* data as input. A brief description of each report and its use follows.

* SMF is an optional feature of the Operating System (OS) (Ref. 1), which collects system, job-management, and data-management information.

The stat (statistics) card report shows detailed accounting information about each job run on the computer, sorted by supervisory group and department. Information such as CPU time, lines printed, region size, disc and tape setups, read-in time, and purge time are shown for each job processed. This report is produced and examined daily and gives indications of overall throughput, average turnaround, distribution of work among departments, and unusual jobs. It also provides a reference for the day's activities.

The abnormal end (ABEND) report provides data about each job that aborts. Information such as failure code, programmer name, job name, and CPU time is provided. These data are also printed and examined daily to give indications of particular users who consistently have problems, specific programs that frequently fail, and repeated ABEND codes that may be symptomatic of system problems.

The usage report provides detailed characteristics of the high-usage programs executed by each department. It also shows a rank order list of these high-usage programs. These data are used to pinpoint programs to be considered for performance analysis and improvement, as well as to pinpoint possible inefficient or unusual use of a program by a particular department.

The high-resource report and the exception report highlight users whose jobs exhibited certain high-resource characteristics such as exceptionally long turnaround time, extended use of central processor time, great volume of printed output, very large use of core memory, and utilization of several setup devices, or those jobs that experience a job control language error after significant expense of resources.

3.2 Other sources of data

In addition to the various automated report programs that provide utilization data, there are several other important sources of bird-dogging data. Direct problem program monitoring and feedback from operations personnel are the two most significant sources.

Program monitoring is achieved through use of a proprietary software monitor that provides valuable execution profiles of user programs. Several monitors are on the market; the project centers are using Boole and Babbage's Program Evaluator (PPE).[2] Experience to date indicates PPE is easy to use, well documented, and consistently helpful in providing areas for program performance improvements. PPE indicates where and how the monitored program spends its time and how compute-limited or input/output-limited the program is. The effects of subsequent improvements to the program are readily apparent by remonitoring.

Operations personnel can provide valuable bird-dogging data. In many cases, user problems may not appear in the automated reports, or problems do appear but their magnitude is hidden.

IV. CASE STUDIES

This section presents several cases that typify many of the long-range studies undertaken as a result of the analysis of weekly and monthly computer utilization data.

4.1 Study 1

For a period of several months, the types and frequencies of ABENDs at the computer centers were investigated. It was found that 15 to 20 percent of all jobs submitted eventually ABENDed and 25 to 30 percent of the total Central Processing Unit (CPU) time was spent executing these jobs. The ABENDs were grouped into four categories:

(*i*) Those that were a result of insufficient estimates of the computer resources required by the job (resources include CPU time, memory, and I/O estimates).
(*ii*) Those that reflected problems of a data base nature.
(*iii*) Those that resulted from a program check condition.
(*iv*) Those that were symptomatic of a hardware malfunction.

The most striking observation from this study was that the inability of users to correctly estimate the computer resources required for their job appeared to be by far the biggest obstacle to successful job execution. As a result of this and other related studies:

(*i*) The support software user manuals were revised to include algorithms for estimating required computer resources.
(*ii*) Modifications were implemented to OS that allowed selected critical modules to complete execution even though the actual CPU time consumed has exceeded the programmer's estimate.

As a corollary to the problem of insufficient estimates, system performance was often degraded by serious overestimation. An educational campaign was initiated by distributing to all project programmers an informational bulletin that clarified the specification of job and of job step region parameters.

Because of the changing nature of the project and its computation requirements and the scattered implementation of study recommendations, objective measurements of subsequent improvements have not as yet been attempted.

4.2 Study 2

The usage report indicated heavy use by one department of a "home-grown" data reduction routine. By revising the program only slightly, CPU time was dropped from 110 to 8 seconds per execution.

4.3 Study 3

Analysis of the execution profile for the CENTRAN compiler demonstrated that a much higher than average number of accesses to the CENTRAN symbol tables were required during the compilation of large programs with certain characteristics. By specifying additional core memory in the region size over the default, overall resource requirements were reduced (and, hence, cost to process the job was reduced).

Detailed data for each CENTRAN compilation were available through the automated reports. The 75 programmers who were responsible for programs with exceptional characteristics were contacted over a period of several months and were requested to allocate additional memory for their compiles. Most individuals complied and experienced a decrease of turnaround time (by reduced elapsed time), with an attendant system cost reduction.

4.4 Study 4

The exception report provided a list of jobs requiring high resource use. With the cooperation of the users, these jobs were scheduled for evening or weekend shifts. Rescheduling of these jobs eliminated them from competition with other jobs for limited prime-shift computing resources.

4.5 Study 5

It was observed by operations personnel, and later confirmed by examination of reports that correlated turnaround time and resource usage, that certain users were taking advantage of a loophole in the computer centers' job-scheduling algorithm. The slightly higher priority assigned by the algorithm to jobs requiring a setup led to the submittal of jobs with unneeded setups. A job-scheduling adjustment corrected the problem.

4.6 Study 6

The usage report indicated that the SAFEGUARD Data Reduction SYSTEM (SDRS) was the largest single user of CPU resources, consuming 20 to 30 percent of all CPU time. Analysis of the facility with PPE indicated that much of this time was spent communicating with the operating system. Interrupt recovery capabilities were provided for each type of input data. These required many different recovery routines that necessitated specifying different interrupt exit addresses to OS many times. The same capabilities were preserved by some minor restructuring of the program and the addition of logic to determine the appropriate interrupt recovery. Post-modification benchmarking revealed an average 60-percent savings of CPU time for this program.

V. CONCLUSIONS

It is the belief of the project centers that bird-dogging is the single most effective tuning activity that can be performed. Bird-dogging is an integral component in reliable project scheduling and effective cost control. As in other areas of system tuning, although the fruits of individual events and incidents seem indisputable, the successes (or failures) of bird-dogging can seldom be proven objectively by quantitative measure. Justification, therefore, remains mostly in the subjective domain.

The bird-dogging effort has been hindered by design errors and limitations in the SMF portion of the operating system and by the lack of commercially available SMF data reduction systems suitable for project needs.* Hence, considerable manpower was expended in developing a series of automated report programs.

The computer centers have found a software monitor, in this case Boole and Babbage's PPE, helpful in providing data for program performance improvement. Every bird-dogger should have something of this sort available.

The ultimate success of any bird-dogging program depends heavily upon the degree of cooperation received from the user community and its management. Care should be taken from the outset to present suggestions and criticism in a positive manner. Helping users to help themselves will contribute to improved confidence in meeting schedules and to lower computer center costs.

REFERENCES

1. IBM System 360 Operating System, System Management Facilities, Order Number GC28-6712-5, International Business Machines Corporation.
2. Systems Measurement Software (SMS/360), User's Guide for PPE, Boole and Babbage, Inc., Cupertino, California.

* To this author's knowledge, there is only one "off-the-shelf" SMF reduction system available, the SMF Selective Analyzer, FDP-5798-AAR, IBM Corporation.

Section V

DEVELOPMENT TOOLS AND TECHNIQUES

SAFEGUARD Data-Processing System:

An Experiment in Software Development

By R. D. FREEMAN

(Manuscript received January 3, 1975)

This paper describes a type of flowchart review used as a program-development technique in which each programmer is required to give a box-by-box explanation of a detailed flowchart of his program to a small group of critical colleagues. Such reviews appear to have caught all the major software design errors before the code was written. It also cut the software-development time by at least 25 percent, representing a return of at least 10:1 in terms of software-development time saved as a result of the week of the group's time spent in the flowchart review sessions.

I. INTRODUCTION

This paper describes a program-development technique used in the programming of the sensor (i.e., radar) control portion of the early-1973 release of the software used in the Meck test system. For this release, the sensor control was completely redesigned and reprogrammed. Reprogramming provided an opportunity to experiment with techniques in program development. Of the techniques that were tried, "flowchart reviews" had the largest effect on the development effort.

II. BACKGROUND

Sensor control serves as the software interface between the Central Logic and Control computer and the phased-array Missile Site Radar (MSR) at the Meck Island test site of Kwajalein Atoll in the central Pacific. The most complex job done by sensor control is to resolve conflicting requests for radar usage, e.g., target search and target track. This is accomplished by changing the time at which one of the requested MSR transmit/receive order pairs is executed by an amount small enough not to degrade the validity of the resulting data. Since it is naturally desirable to obtain the maximum amount of data from the radar, the rules for performing this radar-order-conflict resolution are inherently complex. The memorandum analyzing these rules is about

100 pages and demonstrates that the resulting radar-order-conflict-resolution algorithm meets all the system requirements.

As the test missions at Meck Island became more complex, they began to strain the original version of sensor control. There were problems in program execution time and also in the limitations of the radar-order-conflict-resolution algorithm designed into the original version. It was therefore decided that sensor control would be rewritten, essentially from scratch, using a new data structure and an improved conflict-resolution algorithm.

The development of the new sensor control required about six man-years of work, including algorithm design and analysis but excluding any detailed documentation that might be written in the future. For reasons that are mainly historical and beyond the control of the sensor control group, the programming was done in assembly language. Every few tenths of a millisecond of execution time was important. The new sensor control requires about 5000 lines of assembly language code (plus a somewhat larger number of comment lines) and executes in about half the processor time (about 1.5 to 2 ms) of the old sensor control.

III. THE PROCESS OF FLOWCHART REVIEW

During the reprogramming of sensor control, flowchart reviews were used to find software-design errors or possible improvements before the code was written. As a sensor control group policy, before coding was started, the programmer wrote very detailed flowcharts and data-set layouts.* The flowcharts were to be sufficiently detailed that, given the flowchart, coding the routines would be almost a mechanical process. In particular, every decision point and all possible branches of control were to be shown. On the average, there were fewer than a half-dozen lines of code per flowchart box. The data-set layouts were in complete detail, i.e., down to the level of the bit. Given these layouts, coding the data sets was strictly mechanical. There were no specific format requirements for the flowcharts and data-set layouts except that they be easy to read.

As soon as the flowchart and data-set layouts for an area were complete, a review meeting was held. These review meetings were always attended by the group supervisor and several group members. Those group members specially knowledgeable in the area covered in a particular flowchart review were specifically asked to attend. Other mem-

* A "data-set layout" is a pictorial representation of the structure of a data area. Fields within memory locations are shown left to right across the page and consecutive memory locations are shown top to bottom down the page.

bers of the group were encouraged to attend. Flowchart reviews were also open to anyone else who was interested but, in practice, no one outside the group chose to attend. Except for the supervisor, people attending had either given flowchart reviews themselves or were scheduled to give them. The programmer whose flowchart was being reviewed, therefore, had a technically critical, but sympathetic, audience. Although the discussion of technical alternatives sometimes grew quite spirited, criticism of a programmer's design was never sarcastic and there was no gloating when an error was discovered.

At the beginning of each review meeting, copies of the flowchart and data-set layouts were passed out to all participants. Copies were not passed out ahead of time, nor were they later given to anyone who missed the review, primarily because it was unlikely that they would be read.

Usually the programmer began the flowchart review by giving a brief overview of how his code was structured. No high-level flowcharts were used. However, it proved quite easy for a programmer to point out what sections of his detailed flowchart represented what major functions and, in effect, to create a high-level flowchart in the course of the discussion. If the data-set structure used by his program was at all complex, the programmer usually gave a summary of the data structure at this point, leaving the definition of the specific fields for later. Occasionally, there was some discussion of alternative data structures at this point. Usually, however, any alternatives to the data structure designed by the programmer were suggested during the detailed discussion of the flowchart. This was probably because the functional structure of the data base had been one of the earliest decisions made and was a basis for an improved radar-order-conflict-resolution algorithm.

After this overview had been completed, the programmer explained his flowchart in detail. This explanation consisted simply of starting at the beginning and going through it box by box in the same order as the code they represented would be executed. If the descriptive phrase enclosed by a box was not self-explanatory, the programmer gave a brief explanation of what the code would do. For a few more complex algorithms, the programmer set up an example on the blackboard and carried it through during the discussion of the flowchart. The flowcharts were sufficiently detailed so that it was not necessary to describe how the code represented by a box in the flowchart would do the specified function; this was self-evident. However, it was often necessary to stop after reviewing all the individual boxes associated with a particular major function and to discuss whether the design represented by the flowchart would in fact carry out the desired function for any valid input and retain sanity for all possible inputs passed to sensor control. Also, the participants in the flowchart review interrupted the programmer with a question or comment on the average of once for every two to three boxes in the flowchart.

The participants in the flowchart review, although sympathetic, were expected to take an aggressive "I'm from Missouri and you have to prove it to me" attitude toward every assertion that the programmer made. If the programmer said that a data field began at a particular bit in a particular word, more than half the participants would turn to their data-set layouts to verify that. If the programmer said that the various inputs to a given internal subroutine could be divided into three classes, the other participants would try to think of a fourth. If the programmer said that the inputs from another module were in a particular format, the person responsible for that module would be asked to verify this. If it could not be verified on the spot, e.g., because the module owner was not present, it would be checked later. This aggressive questioning of the programmer's every assumption by his colleagues was undoubtedly the key to the success of these flowchart reviews. The programmer would sometimes catch a minor error, e.g., branch conditions reversed for a decision point, as he explained his flowchart to the group. However, the more significant problems were almost invariably found by the other participants.

Discovery of many more significant problems found during these flowchart reviews often resembled the way a lawyer sometimes (at least, on television) finds a major flaw in a witness's story during cross-examination. Instead of anyone at first noticing the basic problem with the design, someone would notice a minor problem. Two or three people, including the programmer responsible for the code, would then propose obvious patches to the design to handle this special case. The discussion of this minor problem would, however, have focused the group's attention on that particular area of the design. During the discussion of the best way to patch the design to handle this minor problem, someone would notice a second problem. Now that the group had seen two problems related to the same aspect of the design, comments would come thick and fast, with interruptions every few sentences. In a few minutes, this whole area of the design would be thoroughly explored and any problems would be obvious. Often, the person who noticed the second minor problem, and hence triggered the discussion leading to the discovery of the basic problem, was neither the person who noticed the first problem nor the programmer responsible for the code.

Another interesting feature of these flowchart reviews is the way two or three people, usually including the programmer responsible for the code, would occasionally seize the conversational initiative and draw the group down a side path. These side paths would often explore an alternative design in a manner not unlike a chess player exploring the consequences of a particular move. One person would suggest a modification to the original design; a second person might suggest that if you were going to make the first change, the design could then be improved by changing another feature. Another person might then suggest a third change, or might suggest that if you were already going far enough to make the first two changes, you could go all the way, make a certain change in one of the basic design assumptions and redo a portion of the design. These side paths were particularly useful in finding simplifications to the original design. In at least some cases, a few minutes of discussion saved a few weeks of programming and unit testing. In one area, the code used to recover from machine interrupts, the side path led to a spirited technical argument extending through several flowchart reviews and ultimately resulting in a design with more capabilities than any initial proposal.

As a conclusion to this description of the process of flowchart review, it is worth reemphasizing the importance of maintaining a matter-of-fact and unemotional atmosphere. This is essential so that the programmer can accept his colleagues' aggressive questioning as just the rules of the game. Viewed in that light, a flowchart review is just a form of professional review that is part of the programmer's job as a technical professional. A group of programmers meeting for a flowchart review is then not unlike M.D.s holding a seminar to discuss a particular patient's history and the treatment that is or was being given to him. However, if a matter-of-fact atmosphere were not maintained, the aggressive questioning in a flowchart review would be an intolerable insult to the programmer's pride as a technical professional.

IV. RESULTS OF UTILIZING FLOWCHART REVIEW

About two dozen flowchart reviews, including repeats, were required for all sensor control. Although the length of the flowchart reviews varied considerably, they averaged about two hours. Since the entire group often did not attend a flowchart review, two dozen two-hour reviews amounted to slightly less than a week of the group's time. As one would expect, the number of problems uncovered at the flowchart reviews varied considerably. However, the average two-hour flowchart review led to the discovery of about a dozen problems, varying in importance from trivial to major. As a result of the reviews, several areas of the new sensor control were redesigned essentially from scratch, several areas were changed significantly, and no area was left unchanged. Perhaps the best indication of the number of changes that resulted is the number of times that it was worthwhile to repeat the review. In roughly half the cases, the first flowchart review led to sufficiently extensive changes that a second review was held after the design had been modified. Had all the errors uncovered in the flowchart reviews been found a few at a time as the code was written and tested, it would easily have required at least several more months of the entire group's time (equal to roughly one-third of the time actually required) to complete the development of the new sensor control. Thus, assuming that the programmers would write detailed flowcharts or do some other form of detailed design for their own use, there was a return of over 10:1 on the week's worth of the group's time spent in the flowchart reviews. These calculations exclude the time saved in system testing by delivering higher quality software, which probably exceeds that saved during program development. The group responsible for programming the target search and target track algorithms used in the Meck test system has also used flowchart reviews like those used by the sensor control group, with similar results.

Perhaps the most striking result of using flowchart reviews was that all the major software design errors appear to have been caught during the reviews, before the code was written. Excluding a few cases where changes in the system requirements or the discovery of errors in engineering assumptions used by the sensor control group forced some redesign, the design was very stable after the completion of the flowchart review. This illustrates both one of the successful results of flowchart reviews and one of the chief limitations found. If the functional requirements and engineering algorithms remained stable, then the software design remained stable after the flowchart review. However, the flowchart reviews were not very useful in protecting against unexpected changes in system requirements or errors in clearly articulated—but wrong—engineering assumptions made by the entire group. Fortunately, there was only one case where this problem caused a large amount of redesign, and in that case the redesign occurred before any code had been written.

The use of flowchart reviews led to the discovery of two disadvantages that seem inherent in any such highly detailed review procedure. The first is that it involves too much detail to be useful during the preliminary design stage. Sensor control was a modestly sized set of programs designed by a small group whose desks were only a few steps from each other. Thus, the lack of a formal review process during the early stages of the design was not a real problem. However, looking back, it seems that some effort might have been saved if a more formal top-down design approach, with design reviews at intermediate points, had been adopted after the basics of the data structure and radar-order-conflict-resolution algorithm had been determined.

The second disadvantage is the level of boredom that must be tolerated. Interest drops off rapidly if no serious questions have been raised for 15 to 20 minutes, and the discussion becomes very boring. During the sensor control flowchart reviews, periods of intense boredom sometimes lasted over half an hour. Also, the policy of aggressively questioning every assertion sometimes leads to three- to five-minute discussions to resolve trivial points. Despite the boredom involved in this nit-picking, such discussions should not be dropped. Discovery of many major problems resulted from unsuccessful attempts to satisfactorily resolve what seemed at first to be trivial questions.

Concerning the amount of boredom that has to be tolerated during a flowchart review, experience throughout the flowchart review has been that if the leader does not care enough to personally take part in the flowchart reviews, they will not be held. If the leader of a group lets boredom take the edge off his personal aggressiveness, then the whole group loses its aggressiveness. Although it is hoped that the leader would be a key technical contributor to the review process, his chief responsibility is to maintain the group's aggressiveness despite the inevitable boredom—and the leader's personal example is critical in carrying out this responsibility.

To be sure, it is difficult for a supervisor to allocate the several hours required to take part in a flowchart review. However, if a fair-sized piece of software is being built, then the quality of the software design is an important factor in determining the quality of the supervisory group's output. Thus, ensuring the quality of the software design—by one method or another—is an important part of the supervisor's job.

Besides the group leader's personal example, motivating the group members to participate actively requires convincing them that the reviews are productive. Because of the number of problems found during the sensor control flowchart reviews, their usefulness was obvious to the participants, although no one—especially not the group supervisor—pretended that they were fun. As mentioned above, only those group members who had the knowledge to make a meaningful contribution to a particular flowchart review or who could learn from it were specifically asked to attend the review. No one was ever asked to participate in a flowchart review just because of arbitrary group rules. Those group members who were asked to attend, especially the lead programmers who were asked to participate in most of the reviews, were told frankly that the supervisor realized that this was not one of the more enjoyable parts of their job, but that they were being invited because their participation was important. In practice, there was no problem motivating the lead programmers to participate in so many flowchart reviews. The same personality traits that made a person into a lead programmer in the first place also made that person willing to put up with some boredom to obtain the satisfaction of having had a strong personal impact on the quality of the group's work.

V. COMPARISON WITH OTHER FORMS OF PROFESSIONAL REVIEW

It is worthwhile to compare the formal group-meeting style of flowchart review used in the development of the new sensor control with other forms of professional review that have been discussed in the literature. Flowchart reviews are very similar in spirit to Weinberg's concept of "egoless programming,"[1] in which programmers are trained to encourage other members of their programming team to contribute to their work; e.g., by reading their programs. The intent of egoless programming is for each program to be—as much as is practical—the product of the collective efforts of a programming team rather than the product of an individual programmer working in isolation (hence the term "egoless"). The group members are encouraged to be technically aggressive in reviewing each other's work. Also, as with flowchart reviews, group members are encouraged to be as matter-of-fact and unemotional as possible in pointing out errors or making suggestions. As Weinberg has reported, egoless programming has worked extremely well in some programming groups. One advantage of flowchart reviews compared with egoless programming is that flowchart reviews are a formal group meeting in which the supervisor takes part. Thus, their success is less dependent upon personalities and it is considerably easier for the supervisor to ensure that the reviews maintain a uniform standard of thoroughness.

Mills[2–4] has made several very innovative proposals [e.g., chief programmer teams, programmer librarians, top-down design/structured programming, PIDGIN (roughly similar to outlines)] as alternatives to flowcharts for organizing software development. See also the papers by Donaldson,[5] Miller,[6] Baker,[7,8] and Nichols.[9] Chief programmer teams, especially when combined with top-down design, provide an opportunity for a great deal of professional review.

Another technique similar to flowchart review is a "walk-through";[10] Mauceri[11] has used group meetings to walk through the actual code in a manner similar to the way that detailed flowcharts were reviewed in the development of the new sensor control. One difference between the work reported by Mauceri and the flowchart reviews used in the development of sensor control is the handling of problems discovered during the course of a review session. Mauceri reported that the procedure his groups used was to resolve questions and problems "off-line"; i.e., they were put on a list to be settled later. During the sensor control flowchart reviews, the questions and problems could be said to have been resolved "on-line"; that is, resolved immediately as they came up during the review if this were at all possible. As mentioned above, many major problems found during the flowchart reviews were discovered as a result of repeatedly unsuccessful attempts to resolve what first seemed to be trivial problems. Another difference between the sensor control flowchart reviews and the reviews reported by Mauceri is the inclusion of unit and module test cases in the reviews reported by Mauceri. This was not done in the development of the new sensor control. Instead, professional review of unit test cases was obtained by a technique suggested by the various experiments on code reading. Based on the detailed flowcharts used in the flowchart reviews, one senior person in the group did the functional design of the unit test cases for all of sensor control. The individual programmers were still responsible for unit testing of their own code. Thus, they had to review the proposed unit test cases for completeness and possible redundancy. In this way, unit test cases were examined in detail by two people.

It is interesting to compare the group-meeting-style flowchart reviews with the widely practiced technique of "code reading," in which a programmer's code is read line by line by either his manager or a senior programmer. In code reading, ideally the reviewer and the programmer read through the code together, although sometimes the programmer merely gives the reviewer a copy of his program listing. For code written in assembly language, the flowchart review has the advantage that it can be done earlier in the development cycle. However, if the code is to be written in one of the better high-level languages, it is not obvious that the professional review procedure should be based on flowcharts. Even if one were to use a formal group-meeting style of review, it might be better to skip writing highly detailed flowcharts and to base the review on the actual code as Mauceri and his colleagues did. One disadvantage of flowchart review compared with code reading is that a flowchart review will not detect minor coding errors; e.g., misnamed variables.

The fact that a flowchart review involves a much larger number of people than a typically two-person code-reading session is both an advantage and a disadvantage. As a disadvantage, the more people involved in a given review session, the more of the group's time is consumed. As an advantage, a group review appears to be able to detect many more errors, especially errors of omission (e.g., simply forgetting a given situation or a given class of inputs) than would be found if the design were reviewed by any single person. One of the more interesting features of the flowchart reviews was the fact that no one participant noticed half the errors that were found. This illustrates the advantage of flowchart reviews by a group of a programmer's colleagues, as compared with the more traditional managerial practice in which a programmer reviews his design, probably briefly, only with his manager. The traditional managerial review procedure is probably inferior to almost any reasonable procedure that involves the detailed review of a programmer's work by a group of his colleagues.

This is not to say that a formal group-meeting-style flowchart review is always to be preferred to code reading. Flowchart reviews are not very useful for small changes to existing code corresponding to less than several dozen lines of assembly language code and to flowcharts with fewer than a half-dozen boxes. Unless it is possible to review several such changes in one session, the flowchart review will probably be finished—accompanied by much grumbling by the participants whose work was interrupted—in about as much time as the people could be brought together. In fact, some months after the original version of sensor control was delivered to the system-integration team, code reading was introduced into the sensor control group to help tighten coding and testing of the minor changes being added to the original design. How this came to pass is a story with a useful moral.

Some months after the new sensor control was delivered to the system-integration team, minor additions had to be made to the code to provide for some new capabilities. These additions went beyond the software design that had been covered in the flowchart reviews. In the time since the new sensor control had been turned over to the system-integration team, the group, or at least the supervisor (the author), had grown too cocky. The code had run well during the several months of system-integration testing, and a series of minor changes had already been introduced with few problems. Probably significantly, the design for this first series of minor changes had been included in the original flowchart reviews; the implementation of these changes had been delayed. The new changes that went beyond the original design seemed at the time to be just more minor changes; no special review seemed needed. The programmers individually tested their code and released it after they felt that the changes had been thoroughly tested. Suddenly, during one week, the system-integration team found bugs in minor changes submitted by more than half the group. As a result of this, the supervisor got a useful lesson in humility and a certain amount of cheerful harassment from the system-integration team. To deal with this minor fiasco, a referee system was set up for minor changes. Each programmer submitting a minor change was required to select a referee from among the senior members of the group. The programmer would discuss both his proposed change and the procedure to be used in testing the change with the referee. The change could not be released until the referee was satisfied with the testing as well as with the code itself. After the referee system was introduced, the problem of bugs in minor changes came to a very satisfying end.

VI. LESSONS LEARNED

One lesson that was learned from the experiments described above is the extent of the increase in quality and productivity that can be obtained from the disciplined use of professional review. The use of flowchart reviews in the development of the new sensor control:

(*i*) Improved and simplified the software design.
(*ii*) Appears to have caught all the major software design errors before code was written.
(*iii*) Reduced the software development time by at least 25 percent.
(*iv*) Improved the quality of the software delivered.

The use of a referee procedure brought an end to the errors in minor changes turned over to the system-integration team. Other forms of professional review have led to similar results.

A second significant lesson can be learned by comparing professional review with some other techniques that have also led to improvements in program quality and programmer productivity; e.g., programming teams, modular and top-down design, and structured programming. A common denominator to these techniques is the increased structure and discipline placed on the process of writing software. Although what we now know about writing software is undoubtedly much less than what remains to be learned, it is already clear that designing and writing software needs to be a much more structured process than it is today.

REFERENCES

1. G. M. Weinberg, *The Psychology of Computer Programming,* New York: Van Nostrand Reinhold, 1971, pp. 56–64.
2. H. D. Mills, "Chief Programmer Teams: Principles and Procedures," Report N. FSC 71-5108, IBM Federal Systems Division, Gaithersburg, Maryland, 1971.
3. H. D. Mills, "Top-Down Programming in Large Systems," *Debugging Techniques in Large Systems,* R. Rustin, ed., Englewood Cliffs, New Jersey: Prentice-Hall, 1971, pp. 41–55.
4. F. T. Baker and H. D. Mills, "Chief Programmer Teams," Datamation, *19,* No. 12 (December 1973), pp. 58–61.
5. J. R. Donaldson, "Structured Programming," Datamation, *19,* No. 12 (December 1973), pp. 52–54.
6. E. F. Miller, Jr. and G. E. Lindamood, "Structured Programming Top-down Approach," Datamation, *19,* No. 12 (December 1973), pp. 55–57.
7. F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, *11,* No. 1 (January 1972), pp. 56–73.

8. F. T. Baker, "System Quality Through Structured Programming," Proc. AFIPS FJCC, *41*, Part I (1972), pp. 339–343.
9. B. C. Nichols, "SAFEGUARD Data-Processing System: Structured Programming and Programming Production Librarians," B.S.T.J., this issue, pp. S211–S219.
10. A. L. Scherr, "Developing and Testing a Large Programming System, OS/360 Time-Sharing Option," *Program Test Methods*, Englewood Cliffs: Prentice-Hall, 1973, pp. 165–180.
11. R. Mauceri, "Increasing Quality and Productivity Through the Development Process," speech given at Bell Laboratories, Madison, N.J., July 14, 1973.

SAFEGUARD Data-Processing System:

Structured Programming and Program Production Librarians

By B. C. NICHOLS

(Manuscript received January 3, 1975)

This paper discusses the phased implementation of structured programming techniques over a period of two years. It was observed that, by standardizing programming techniques, the resulting program becomes more maintainable and programmer productivity increases. By confining the clerical work of programming to the program librarian, productivity again increases.

I. INTRODUCTION

Structured programming techniques have been widely publicized throughout the data-processing industry. In March 1970, one programming department was chosen as a pilot group to test the validity of these techniques in the SAFEGUARD environment. This paper summarizes the experience gained in the ensuing two years, as increasingly advanced structuring techniques were used by the pilot group. Phased introduction of each technique is discussed to indicate that the transition from a conventional to a structured environment can be accomplished smoothly. Effects of the phased transition on personnel are discussed, and quantitative productivity data are provided for each phase. Although the statistical validity of these data must be qualified, a definite trend toward increased productivity is indicated.

II. DEFINITION OF TERMS

Within the pilot group, the term "structured programming" was used to identify five distinct techniques. They are structured code, top-down programming, code reading, PIDGIN, and the Program Production Library (PPL).

Structured code is based on a mathematical theorem that shows that any program can be developed by the appropriate nesting of three basic logic patterns: sequence of operations, conditional branch to one of two operations, and repetition of an operation while a condition is true.[1] Elaboration of these patterns leads to the five basic logic structures used by the group to implement structured code: sequence, IFTHENELSE, DOWHILE, DOUNTIL, and CASE. Since only this statement grouping was permitted, standardization of code resulted. Also, adherence to these logic patterns results in complete control of all branching logic and therefore programs are easily readable from top to bottom.

Top-down programming requires that both design and code be developed from the control logic level down to the detail logic level. Program design has traditionally followed this approach, proceeding from system specifications to design instructions. Top-down design adds to this requirement that the control levels be coded prior to completing the detail design of lower-level paths.

Conventional program code, however, frequently does not follow the top-down approach. Detail level logic is often coded concurrently or before high-level control logic. Top-down code dictates that the next level of program code cannot be developed until all paths upon which this code depends have been coded and (preferably) tested.

Another structured technique, *code reading,* was made possible by the use of top-down programming and structured code. This is the practice of having all programmers exchange listings to ensure that each program is read by someone other than the author. Desk debugging is significantly increased and fewer, if any, preliminary clean-up computer runs become necessary.

Large programs, however, still present a problem since the ability to read them top to bottom is jeopardized by their total length. To resolve this problem, the pilot group used a segmenting technique that breaks down the program structure into functional segments. Each segment is then constructed so that ideally it occupies no more than one page of a program listing.

PIDGIN is a program design language that combines English, a programming language, and the structured conventions. This language was used to describe each functional segment. Through this design medium, system functions are visually broken into dependent segments showing the relation of each segment to the overall purpose of the program.

The last technique used in this study was the *Program Production Library* (PPL). The PPL concept stems from the observation that much of the task of computer programming is clerical. The PPL provided a standardized means for recording, cataloging, and filing all code generated, and it ensured a coherent library control system during program development and maintenance. It also provided a means for standardizing the JCL-type interface to the computer whereby processing options (Compile, Linkedit, Modify, etc.) were invoked through key words chosen by the programmers. A more detailed discussion of the PPL appears in Section VII.

III. PROGRAMMING ENVIRONMENT

Structured programming techniques, PPL, and program librarians were introduced into a programming project over a one-year period and observed for an additional year. The project comprised the development of independent functional tests for the CLC operating system.[2] The complete set of tests was developed incrementally over several years, and the end product was a set of test specifications and the programs implementing them. Data for this study were gathered from the development of the test monitor facility and the first 11 test sets. The test monitor facility provided standard result recording for all tests, such that each test set contained no reused design or code. The SAFEGUARD assembler level language was used for all coding.

The nature of the development environment is also important for interpreting the results of this study. Test sets were being developed in parallel with the operating system. The CLC was the target computer,[3] but all software development occurred on the IBM 360, testing being accomplished on the CLC or by simulation on the 360.

The activities of the pilot project group were confined solely to test design, coding, and documentation. Testing and debugging were accomplished by a separate test team. However, correction of implementation and coding errors in response to error reports made by the test group was a continuing background activity to all development efforts. This maintenance activity reached a peak during the first two months following delivery of each test set.

A programming team consisting of three to four people, each having an average of two years of programming experience, was assigned to each test set. The schedule time allowed for the development of a test set was four to five months, or an average of 16 man-months. Each development cycle had three stages: test specification (2 man-months), test design (3 man-months), coding and documentation (11 man-months).

Another equally important aspect of the development environment was the personnel skill mix. The SAFEGUARD software proved to be a great equalizer in that personnel new to the project had to learn not only the complex application area, but also a new spectrum of support software. The result was that experience with SAFEGUARD software was frequently equal in value to overall programming experience. The rotation of SAFEGUARD-experienced personnel to related critical project areas was common. In the pilot group, personnel assignments were rotated frequently throughout the two-year period studied, thus keeping the average programmer experience constant through each development cycle. Over a three-year period, a total of 21 programmers were assigned to the pilot group. Its total size ranged from 7 to 10.

The difficulty of the programming job is another important consideration. In retrospect, the tests performed in earlier sets are less complex but, at the time of their development, user documentation for the operating system was incomplete. The complexity of the later test sets was significantly higher; however, by this time documentation had improved, familiarization with the general modus operandi of the operating system had occurred, and personnel were accustomed to the test monitor interface. Hence, the relative difficulty of the programming job remained constant.

IV. IMPLEMENTATION PHASES

The test monitor and the first test set were developed using conventional programming methods. Improved programming techniques were then introduced in two distinct phases. The next three test sets were developed using structured programming and represent phase I. The next six were developed using structured programming, the PPL, and program librarians, representing phase II.

V. QUANTITATIVE RESULTS

Table I quantifies the effect of each phase on programmer productivity over the two-year period. For this study, productivity is defined as the number of delivered lines of code produced per day during the coding phase. Activities during coding include design, documentation, coding of the unit programs, and maintenance of previous test sets. Debugging was not a part of this activity, as has been previously discussed. Source statement counts include all lines coded, including comments and other descriptive lines required to meet documentation standards. The object size includes both instruction and data areas and measures the delivered product, as do the source lines. The ratio of source to object is provided to give the reader a rough indication of the number of executable instructions per line coded. Programmer-days reflects cumulative elapsed days for each programmer, and it does not account for overtime, vacations of less then one week, or illness. Also, it reflects only days spent by programmers, i.e., it does not include management or program librarians. It is the intent of this study to indicate the effect of new technologies on programmer productivity as defined above, rather than on overall product cost.

Table I — Comparison of productivity

Delivered Item	Source Lines	Object Size (32-bit words)	Ratio of Source to Object Size	Programmer-Days (Coding Phase)	Source Lines Per Programmer-Day
		Conventional			
Test Monitor	4056	1914	2.1	301	13
Set 1	6072	6540	0.9	381	16
		Phase I			
Set 2	9654	7300	1.3	240	40
Set 3	4271	2150	2.0	150	28
Set 4	6601	3500	1.9	130	51
		Phase II			
Set 5	9968	3700	2.7	165	60
Set 6	14689	7000	2.1	225	65
Set 7	16773	6500	2.6	150	111
Set 8	5588	3900	1.4	136	41
Set 9	11666	5830	2.0	160	73
Set 10	11596	6230	1.9	158	74

A comparison of raw productivity rates was made over the two-year period reported. No difference between the three- or four-person team was observed, and thus no distinction is made in Table I. The data in Table I should not be used out of the context of the background already provided in previous sections, since this can lead to rather startling conclusions. Table II summarizes the data for each phase, but must only be considered as indicating a trend rather than actual percentage gains. The productivity figures reported are dependent on many factors unique to the specific development environment of this study.

VI. PHASE I

Phase I introduced structured programming, code reading, and unit-level top-down approach into the development cycle. These techniques can probably be introduced into any existing programming project if the following prerequisites are satisfied. The programming language in

Table II — Summary of results

Implementation Phase	Total Source Lines	Total Programmer-Days	Average Lines per Day
Conventional	10128	682	14.7
Phase I	20526	520	39.8
Phase II	70280	994	70.8

use must include instructions that implement the structured programming logic patterns. This may require the development of a special set of macros to support the branching logic. In the case of the project being studied, two man-months were required to develop a macro package to provide structured statements in the language used. At the outset of phased introduction, a programmer experienced in structured programming must be available for consultation. This person need not be a member of the group itself, but should conduct an orientation seminar for those programmers asked to use the new techniques. The program areas selected for structured programming must be functionally separate from other areas. It is difficult to introduce these techniques into an existing program unless the new code represents a distinct functional unit that can be restricted to having only one entry and one exit.

The effects of phase I implementation were significant. Resistance from programmers occurred at the orientation seminar and during the early stages of implementation. However, once they began to use structuring techniques for program control, acceptance was quick. Resistance to the new techniques seemed to be directly proportional to programming experience. That is, firmly established coding habits were difficult to discard when they were to be replaced by a standardized method. There was also the matter of bruised pride, a definite psychological side effect. However, experienced programmers soon became convinced of the validity of standardization, based on their past experience and the obvious benefits. For example, because of the standardized method of coding, code reading proved to be a valuable desk debugging tool.

Toward the end of phase I, it became evident that maintenance of programs was easier. As is mentioned in Section III, the maintenance activity for each test set peaked during the first two months following delivery. Maintenance requirements generated by debugging activities generally required one programmer full time for that period. Structuring techniques made the programs easily readable and enabled them to become community property. In fact, this standardization was so effective that, immediately following delivery, maintenance of all programs in a test set could be assigned to one member of the original developing team. Maintenance responsibility included an average of 100 programs per test set. Transferability of program maintenance thus had the effect of freeing key personnel for scheduled critical design activities for the next test set, as well as lessening the impact of loss of personnel through rotation. Orientation of new personnel was also simplified, since this could be partially accomplished through code reading.

VII. PROGRAM PRODUCTION LIBRARY AND THE LIBRARIAN

The Program Production Library (PPL) facilitates the work of programmers engaged in code development; it also aids project and line management wishing to review the project's progress. The PPL depends on a computerized library system in which all types of data have a defined source and destination. It is maintained by clerical personnel, but no operations are carried out in it unless they are directly requested by the programmers.

Program librarians staff the PPL. Just as structured programming must be introduced slowly, the program librarian must be given adequate time to learn. The librarian's first job is to provide an interface with the computer center, submitting and picking up jobs. The librarian can later be taught to change source code, working from marked-up program listings. The skills required for this are the ability to interpret the sequence of source changes, to make up the appropriate change deck, to incorporate this change deck in the necessary computer input deck so the program source change will be made, and to include the proper tests so that the programmer will have a new set of outputs to analyze. This represents a high level of proficiency for a program librarian, yet it requires no programming skills.

The librarian is also responsible for maintaining current listings for all programs being developed. During the development of interdependent programs, library listings must be updated daily, since several programmers may be working on the same program or require interface to a common data area. However, on the project studied, each test in the set was designed so that all required predecessor conditions were established during the test. Each team member was assigned a specific test, and, since only the programs within a test were interdependent, it was not necessary to file final listings until they were ready for debugging.

VIII. PHASE II

Phase II involved the introduction of the programming production library and the program librarian. The overall effects observed during phase II were not immediately visible. This was due mainly to the learning curve of the program librarians. The acceptance of the librarian service and the PPL concept was not universal, and it occurred much more slowly than the acceptance of structured programming techniques. Initially, it had the effect of placing one more barrier between the programmer and successful computer output. During the project studied, the training of new librarians was a continuing activity, because of frequent turnover. One month overlaps for training were worked into the schedule, increasing the overall manpower required to support the PPL. During that training period, library performance usually suffered. This also hampered the expansion of PPL functions, since overall accuracy of PPL output varied. It took four to six months before programmers began to rely entirely on the librarian service.

The sporadic accuracy and reluctant acceptance of the PPL and librarians can be attributed almost entirely to frequent turnover of librarian personnel.

The librarian's job is not trivial and requires about two months of close supervision by trained personnel to achieve the basic skill of job setup using change decks provided by programmers. Six months after phase II began (Test Set 7, Table I), the impact of the training period had been mitigated by increased experience and improved PPL procedures that defined additional fail-safe measures for new personnel. For the remainder of the study, PPL throughput and accuracy increased despite continuing turnover.

Another effect of this turnover was the operation of the PPL on a "pool" basis. Since experienced librarians were scarce, all PPL activities were centralized into a pool of three to four librarians shared by four programming departments. This arrangement was quite effective in handling the peak activity periods that precede each delivery.

The number of librarians required for such a pool varies according to the amount of new development being done, the number of programmers involved, and their skill level. The ratio used in the environment described here was 6:1; that is, one librarian for every six programmers. These personnel were not added to the programming group. Instead, given the 6:1 ratio, in a group of seven programmers with an average experience level of two years, one programmer was replaced with one librarian. The remaining six programmers then produced the same amount of code with the aid of the librarian as the original seven programmers would have produced without the librarian.

Another observation is that personnel new to programming can gain programming experience quickly since they are not concerned with the detailed procedures required for job submission and job handling. They need only concentrate on the technical aspects of programming.

IX. CONCLUSION

Standardization of programming techniques through structured programming and its related practices leads to increased maintainability. Background maintenance activities are more easily rotated since structured programs become community property. The PPL concept extends standardization to the programmer/computer interface and as such is beneficial. The role of the program librarian removes as many clerical tasks as possible from programmers, allowing them to concentrate more directly on the technical content of development. The productivity trend indicated in Table I is presented to indicate the effect of these new technologies on programmers. Obviously, productivity should increase as programmers are freed of time-consuming clerical tasks as indicated by phase II. However, it can also be seen that productivity in phase I increases simply with the use of standardized programming techniques.

REFERENCES

1. C. Bohm and G. Jocopini, "Flow Diagrams, Turing Machines," Communications of the ACM, *9*, No. 4 (May 1966).
2. J. P. Haggerty, "SAFEGUARD Data-Processing System: Central Logic and Control Operating System," B.S.T.J., this issue, pp. S89–S99.
3. J. W. Olson, "SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control," B.S.T.J., this issue, pp. S41–S61.

Section VI
PROJECT CONTROL

SAFEGUARD Data-Processing System:

Management Overview

By E. J. DAVIS and H. M. JACKSON II

(Manuscript received January 3, 1975)

This paper describes the management approach developed to support the SAFEGUARD software design effort. Project organization and some techniques used for planning and control are discussed.

I. INTRODUCTION

The magnitude and scope of the SAFEGUARD system software-design effort presented unique management challenges across a broad front. Solutions to problems involving organizing, planning, activating, and controlling had to be tailored to the specific needs of the project. Successfully achieving the objectives of perhaps the most ambitious software development effort undertaken to date was no easy task. Although no dramatically new techniques or remarkable insights into the management process emerged, several useful lessons were learned. While there was not a wealth of tradition and folklore to draw on with regard to similar software development efforts, we found that the fundamental management approaches and disciplines developed over the years in hardware and systems design and other software development activities at Bell Laboratories were in most cases directly applicable.

II. ORGANIZATION

The organization structure that emerged for managing the SAFEGUARD software project is a case in point. We established an organization designed along the general lines of major deliverable generic systems. This organization is shown in Fig. 1. Note that there were four centers reporting to the project director. One center was charged with total SAFEGUARD systems design responsibility. This meant that this center concerned itself with high-level requirements, with evaluation of the design, and with customer interaction. This center undertook software design in the form of simulation programs and other

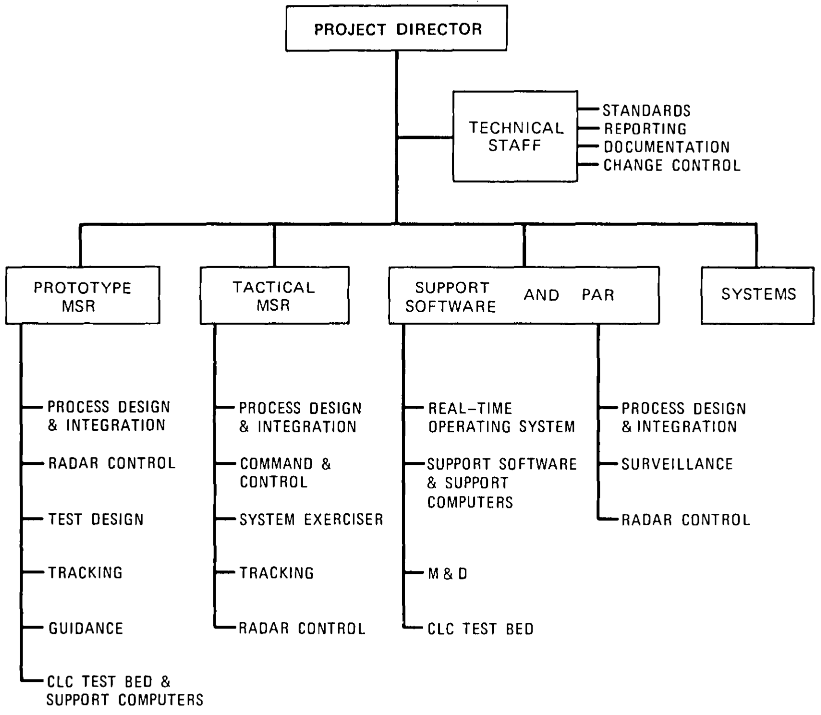


Fig. 1—Organization structure.

analytical tools which were necessary to support evaluation or the development of requirements, but designed no software system deliverable to the customer. Each of the other three centers was charged with design, test, documentation, and delivery of software associated with specific radars, i.e., the prototype Missile Site Radar (MSR) at Meck Island, the tactical MSR at Grand Forks, and the Perimeter Acquisition Radar (PAR) at Grand Forks. The PAR center was also charged with the responsibility for designing support software for the tactical radars.

The departments within these centers were given specific functional design tasks as indicated by their abbreviated titles. The identification of a number of subprojects, derived from the total project work breakdown, permitted a second organizational structure to be superimposed on the line organization structure of Fig. 1. Figure 2 shows one of these subproject organization structures for the MSR weapons subsystem. A project manager was designated for this subproject; in this case, he was the department head (second-level manager) of the Process Design and Integration Department. His responsibilities as project manager included high-level planning for the subproject, detailed design and its implementation, integration and testing at all levels, and monitoring and control of all subproject critical activities. He generally was the person who scheduled and conducted design reviews and periodic project meetings where key engineers, programmers, first-level managers, and support personnel worked together to identify problems and initiate action to solve the problems. The subproject meetings also were used to disseminate information of interest to all those working on that particular subproject. Because the organization remained intact throughout the life cycle of the project, the project manager frequently was called on to preside simultaneously over control of a released system, a system in the planning and design stage, and one in the integration and test phase. The project manager was given a great deal of latitude as to how he managed his subproject. As is evident from Ref. 1, a variety of management approaches were used concurrently, and many contributed to the overall project success. Emphasis was on results rather than technique.

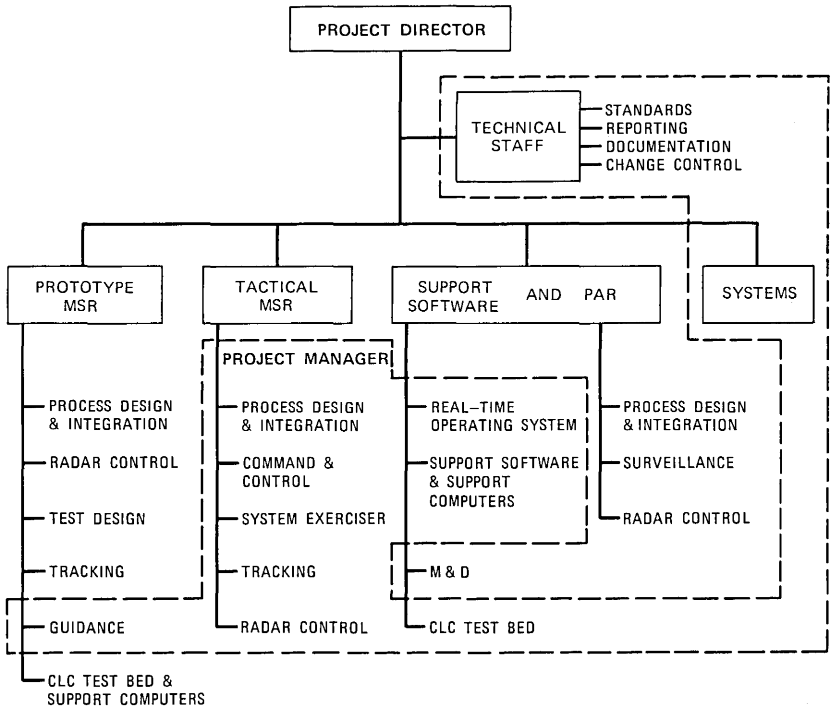


Fig. 2—Subproject organization structure.

Figure 2 shows that the MSR weapons subsystem manager considered people in other centers—for instance, the systems engineers, the CLC test bed operation, the guidance designers, the real-time operating-system designers, and the support-software and support-computers people—as part of his subproject. Note the horizontal spread of this project as it reaches across center boundaries for the people to provide its component parts. Conceptually, it illustrates the coordinated system of relationships among essential functions typical of a matrix type of organization.

All together, there were 17 subprojects—some of them nested within major subprojects like the one mentioned above—with project managers at the second level of management. Experience proved that there was a great deal of commitment to subproject goals on the part of all personnel involved. Clearly, this structure had the potential for conflict—particularly relative to critical resources like the CLC test bed, where goals for two or more subprojects were in competition. However, overall project goals were pretty well understood at all management levels so that conflicts rarely had to be referred up the line-management chain for solution. While the potential conflict situation was recognized, the benefits of cross-fertilization were also a consideration. Good ideas and design approaches were frequently passed rapidly from one subproject to another because of subproject ties that spanned the line organization.

In Fig. 1, note that there was a technical staff organization that had the charter to attack certain projectwide problems, such as training, project standards, documentation, change control, and management reporting. In some areas it provided services to the various project managers, such as training new people. In other areas, it acted as a catalyst to cause project standards to be created. It was not an enforcement agency. For instance, this group sponsored studies and development of structured programming and promoted the development of critically needed macros, but it did not have the authority to impose structured programming as a standard on any subproject. That type of decision was in the province of the project managers.

The project management approach as implemented on the SAFEGUARD software project proved to be a stable organization capable of eliciting strong project commitment at the working level and close technical control in the appropriate line organizations.

III. DETAILED PLANNING

Once overall project and subproject goals were defined and an organization was designed to accomplish them, a detailed development plan was constructed. This development plan, which was prepared in parallel with the Data Processing System Performance Requirements (DPSPRs),[2] forecast the needs of the entire project and spelled out the development approach.

Estimating algorithms, derived in part from a study of previous Bell Laboratories work in electronic switching systems and software development for earlier military systems, were used to help plan the allocation of resources. These algorithms were applied to the estimation of resource demands for each major activity. Schedules were then built up within the constraints of budget, time, and manpower. Trade-offs among these primary resources allowed the coordinated scheduling of critical activities. This anticipation of requisite predecessor/successor relationships between various parts of the job was designed to minimize delays, bottlenecks, and interruptions. Obviously, the initial plan was changed many times during the course of the project. However, it eventually led to very detailed plans which were extensively used throughout the project.

The planned addition of large numbers of people to the project, coupled with an increasing reliance on subcontractor performance, presented a significant management challenge. For example, the accomplishment of in-house training required establishment of a corps of instructors and preparation of text materials. The overall plan had to provide for this substantial investment in student and instructor time. In some cases, where traditional mechanisms were not feasible, novel techniques for evaluating and controlling subcontractor performance were adopted. One such method, the Cost-Plus-Award-Fee contract,[3] was considered one of the major project successes.

In order that forecasts of manpower buildup and total project cost be realistic, it was important that the development plan be implemented and kept current. To this end, a management reporting structure was set up by the technical staff organization to update the development plan and schedules and to provide monitoring information to project managers.

The significance of planning was that it existed across the entire project and that it used reasonably consistent definitions. The subproject managers were not required to use the algorithms that had been put together in the original development plan in working out their more detailed plans.

A conscious effort was made throughout the planning process to require the active involvement of those people who were to be charged with the responsibility of implementation. Participation in the formulation of goals, plans, and schedules conduced to a personal commitment to carry them out. In addition, the unconstrained format of the plan encouraged teamwork and emphasized the use of creativity.

IV. STANDARDS AND CONTROL

Development of appropriate standard operating procedures for designing, testing, documenting, and delivering software was a difficult and tortuous process. Since comprehensive standards did not exist at the start of the project, to a certain extent it was necessary to develop them in parallel with initial development of the software itself.

Rather than create a large, specialized bureaucracy, a small group was organized to act as a catalyst for generation of necessary standards. This group identified the need for specific standards either independently or through requests from design or test groups. A sponsor, usually from one of the design groups, was appointed for each required standard. The sponsor, in concert with designers from other subprojects, prepared a draft that was circulated to the management of affected organizations. Eventually, through a process of iterative feedback, each standard was approved at the highest level for projectwide implementation. In practice, this procedure proved very time-consuming, frequently requiring reliance on preliminary drafts when no approved standard existed. As might be expected, one of the first standards that was provided consisted of a procedure for changing standards.

The standards were divided into a number of different areas, the major ones being change management, documentation, and management reporting. In the area of change management,[4] for example, standards for "freezing" a software unit were developed. As a minimum, to be considered for freezing, a software unit must have been properly documented, successfully assembled or compiled, and successfully unit-tested. While freezing did not stop changes to software units, it did require the application of configuration control procedures, which made all proposed changes clearly visible to interested managers.

Also included in change management were standards and procedures for reporting program malfunctions. The primary mechanism was a standardized trouble report/correction report form that kept all information about a problem and its solution on a single sheet of paper. This report was eventually adapted for describing any discrepancy between observed status and requirements and, as such, became very widely used to track current program status.

Documentation standards attempted to identify and describe every type of document that was needed. Since documenting any large system is a costly and time-consuming process, each requirement was subject to the criteria of reasonableness, usefulness, and timeliness. First, it is not reasonable to expend a great deal of effort to produce a formal document when the information it contains can be made available less expensively in other ways. Second, there is no point in preparing a document that is not going to serve a useful purpose. Finally, a document's utility is greatly diminished if it is not available when, where, and in the form that it is needed. Certainly, schedule constraints did not always allow the criteria to be met, and quite a bit of learning as to just what was useful took place only after the documents were put to the test of use.

Management reporting standards were keyed to a computerized management reporting system that was developed for use on the project. The system incorporated data bases for schedule, manpower, and computer usage information, and was designed to produce a wide variety of special-purpose reports.

V. DISCUSSION

Although, as stated before, no major new management techniques emerged during SAFEGUARD development, the project's success can be attributed at least in part to the close attention that was paid to the content and control of requirements documents and to the early and detailed planning of testing. Most important, highly skilled technical people were selected for key management positions. They were relieved of most tasks peripheral to their jobs, and, subject only to the constraints of *necessary* standards and control, they were allowed to use their own style.

The papers that follow deal with some lessons learned in establishing software change control systems and subcontract administration systems. A critical appraisal of SAFEGUARD project management—as seen by the managers—is also included.

REFERENCES

1. J. D. Musa and F. N. Woomer, Jr., "SAFEGUARD Data-Processing System: Software Project Management," B.S.T.J., this issue, pp. S245–S259.
2. D. W. Meseke, "SAFEGUARD Data-Processing System: The Data-Processing System Performance Requirements in Retrospect," B.S.T.J., this issue, pp. S29–S37.
3. W. H. Mac Williams and J. E. Petersen, "SAFEGUARD Data-Processing System: A Cost-Plus-Award-Fee Contract for a Large Software Development Program," B.S.T.J., this issue, pp. S237–S244.
4. D. Van Haften, "SAFEGUARD Data-Processing System: Software Change Control," B.S.T.J., this issue, pp. S231–S236.

SAFEGUARD Data-Processing System:

Software Change Control

By D. VAN HAFTEN

(Manuscript received January 3, 1975)

Large software projects require control procedures to ensure that changes to code can be made systematically. Programmers, however, wish to be able to make changes to their programs without being bothered by administrative considerations. This paper explores the attitudes of people toward change control and the problems associated with establishing a workable system.

I. INTRODUCTION

Software change control—formalizing the identification and resolution of program errors and improvements—has been critical to SAFEGUARD for three reasons. First, software change control promotes systematic communication. Anyone on the project can formally record a problem. A formal resolution is then ensured; the suggested change is either accepted, rejected, or consciously deferred, but not ignored. Second, software change control helps in estimating the maintenance activity required and in scheduling new software releases. Third, software change control provides visibility. It allows one to see what errors have been found and what action is being taken about them. Based on the requests for changes, one can determine which capabilities of the software are being used. Change control can ensure that the design intent of the software is maintained by consistently identifying all changes made.

II. THE THREE PHASES OF SOFTWARE CHANGE CONTROL

Change control on SAFEGUARD has passed through several stages, progressively becoming more formalized. The first stage was essentially *no control at all.* At the beginning of the project, the only evidence of what the final product would be was a requirements document and a high-level design specification. Developers responsible for building the final product as specified by such documents had a very special attitude toward the software they were creating. Typically, a programmer felt he understood the design considerations and implementation details of the program he was coding, and he felt his knowledge of the code was such that he could remain fully aware of all the ramifications of any changes to it. At this time, the programmer was not bothered with any type of change procedures, for he did not yet have a stable product, and possibly not even well-defined requirements. Any constraining procedures would merely have hindered him from doing his job. This phase of no-control continued as long as the programmer did not have to deliver his program to anyone else.

The second phase might be called *informal change control.* The phase began when several programs had to be integrated, and people other than the original programmer became involved. It was now desirable to have problems documented on forms called trouble reports and solutions—though not coding details—on correction reports. The trouble report and the correction report should be on one sheet of paper, to keep all the information about a problem and its solution together. This procedure met a fair amount of resistance, yet it is only consistent with the standard practice of the scientific and business world, where people write down their ideas, agreements, and problems without feeling they are needlessly harassed by paperwork. Experience has taught them the necessity of doing so. The software world is no different, since programmers, like people, cannot remember every problem and situation they encounter.

For two reasons, it is important that trouble-reporting procedures be set up before they are required. First, if they are not defined beforehand, a vacuum will exist when they do become necessary, and each part of the project will be forced to establish its own. Of course, the primary responsibility of groups that are integrating and testing software is to get a working product, not to define procedures. This responsibility will take precedence, and thus the resulting procedures may not be as good as one might expect. A second and more compelling reason for having change-control procedures defined in advance is that, at a later time when all the software on the project is brought together, it is desirable to have a consistent procedure projectwide.

The final phase of change control for the SAFEGUARD project is called *formal change control.* Since the software is being sent to a remote site for testing and eventually will be sent to the customer, stringent control is essential. In this phase, a central control organization having the following objectives is involved. First, the organization provides consistent, complete, and adequately documented deliveries to remote sites or to customers. Second, it accepts trouble reports, noting problems in the software, and keeps a record of what is being done about these problems. Third, the central control organization checks that certain minimal standards are followed in documenting the problems fixed in each software release, and it checks that all programs to be included in a release are properly identified. Finally, it provides historical backup of all SAFEGUARD software, including source code, object code, assembly listings, and load modules.

Whether a central organization is designated to perform change control or whether this responsibility is scattered throughout the design, test, and integration groups, someone will ultimately do it. On SAFEGUARD, a very small central organization was designated, but it did not insert the actual changes into the code. Therefore, at least one group in each process design department evolved into a control group for that department. Each of these groups defined their own procedures, in some ways making the central organization superfluous. Each department felt it had unique change-control problems that could only be solved by a change-control group that reported to that department's management. It was also felt that only under such an arrangement would the change-control group have the requisite interest in meeting the schedules and objectives of the department. These attitudes made transition to formal change control under one central organization difficult.

The transition from informal to formal change control can be smooth only if there is adequate management backing for such a move. This backing is necessary because of the interjection of a central control organization that is in a position to police certain activities of the development groups. Programmers and even managers are reluctant to let this control organization become involved in their activities, and will probably question both its necessity and its competence. The degree of success this central organization has will depend first on management backing and second on the similarity between the existing informal change-control procedures and the desired formal ones.

III. ESTABLISHING A CHANGE-CONTROL SYSTEM

Thus far, we have considered primarily the human aspects of software change control. The problems associated with this part of the subject are difficult to define and the solutions nebulous. Problems of standards and mechanisms for an effective change-control system are easier to solve. As a rule, these standards and mechanisms are necessary but not sufficient for maintaining control of software.

For three reasons, SAFEGUARD follows a standard procedure for identifying program statements that are changed to fix a given problem. First, the programmers responsible for maintaining the code in the future will be better able to determine the intent of previous changes by being able to relate source statements via their "change level" to a specific correction report. The change level of the program is incremented by one for each correction report written against the program, and the change level is placed on each altered statement.* Second, both the programmer and anyone else examining the code can double-check that the intended change was in fact put in. The third reason, which applies only when the object code is being patched, is that a programmer at a remote site who has solved a problem with a patch may want to check the source code change to make certain it does the same thing his patch did. This checking is especially important when the source code is written in a compiler-level language.

Both source and object code are maintained using a projectwide storage and retrieval system. This system allows the automatic insertion of new change levels into the source code as new statements are added to a program or existing statements are changed. These change levels are then carried through to the assembly listings. In addition, this system provides a convenient mechanism for transmitting changed programs from development groups to testing groups and, ultimately, to the central control organization. The library system was available to programmers early in the project.

When a set of programs is first placed under formal change control, a configuration listing is created. This configuration listing specifies, at the very minimum, a list of all the individual programs with their change levels and a precise identification of all support software (compilers, assemblers, linkage editors, etc.) used in creating this release. With each new release, this configuration listing is updated.

SAFEGUARD programmers write a large number of trouble reports, and an automated mechanism is used to keep track of them. This system was designed and built early enough in the project so that it could have been used to record trouble reports during the informal change-control phase. Although the software Status Accounting System (SAS) was available, developers all seemed inclined to invent their own automated systems because SAS was operated by the central organization at a time when each group wanted to maintain its own data base. These groups should have been allowed to maintain individual data bases using SAS.

SAS can create reports by retrieving and sorting on any data parameter stored for each trouble report and correction report, or on any combination of data parameters. The following information is stored for each trouble report: trouble report number, the program in which

* Since source statements have change levels, so also do object decks and assembly listings. The concept is also applied to load modules, patch decks, user and maintenance documentation, etc.

the problem was detected, the date the problem was detected, a functional description of the problem, the originator of the trouble report, the person to whom the problem was referred for resolution, a status indicator showing the current status of the trouble report, the date of last status change, comments about the trouble report, and the date the correction report is due. The following information is stored for each correction report: correction report number, the program in which the problem was corrected (including its change level as discussed previously), the date the correction report was written, the originator of the correction report, an indicator showing the current status of the correction report, the date of the last status change, and the identification of the load module in which the updated program was first released.

The timing of the definition of forms and procedures was important because programmers became accustomed to the forms and procedures used during informal change control and did not want to convert to others. Thus, the official trouble report/correction report form was defined early in the life of the project, avoiding the proliferation of unofficial versions. The procedures followed during informal change control were a subset of those followed during formal change control. The primary difference, of course, is the presence of the central control organization during the formal period.

The major steps of the SAFEGUARD formal change control process are now described. When someone discovers a problem, he writes a trouble report and submits it to the central organization, which logs it in and forwards it to the people responsible for the program, who accept, reject, or defer it when it is received. They tell the change control organization the date by which a correction for the problem will be submitted. After the people responsible for the program have updated their code, they write a correction report describing the change. They test the new release and update the configuration listing to include the new change levels of programs that have changed. They now send the source code, the object code, assembly listings, a load module, and all correction reports relating to this release to the central control organization. The control organization checks that all changes have been documented; that the source code, assembly listing, and the object code of each program in the load module are consistent; that all program change levels are specified; and that the configuration listing is accurate. This organization then prepares copies of the software for shipment to users or remote test sites.

Although it is not being done on SAFEGUARD, the central control organization should be responsible, upon direction from the development areas, for actually making the changes in all released software.

This requires a substantial commitment of manpower to the organization, but it is one way of ensuring that the changes indicated by correction reports are indeed made, and that no others are.

IV. CONCLUSION

Two aspects of software change control that were relatively successful on SAFEGUARD were a projectwide library maintenance system to control source and object code and a standard trouble report form. These two were not developed over a long period of time, but appeared very early in the project. Because of this stability, software developers grew accustomed to using them. The library maintenance system was available during the first phase (no change control), and the trouble report form was available at the beginning of informal change control. It was recognized that early introduction and acceptance would be beneficial, because transition to the later phases would be simplified. Two additional features of the system, change control procedures and software status accounting, proved to be more troublesome to define and implement. Since, early in the period of informal change control, each process area independently developed its own procedures, a certain amount of reexamination and redefinition was required during the transition to formal change control.

Any software change control system is destined to meet with some resistance. Programmers as a rule have very definite ideas about what should be done to their software. This factor combined with the dynamic nature of software makes change control a difficult problem, not so much in establishing the mechanisms and procedures, as more in dealing with human factors and ensuring adherence to procedures. The first step is to recognize that change control is a problem that should be addressed early and, in fact, *will* be addressed either early in a systematic manner or later in a less organized but more costly manner. Only the developers' admitting this and conscientiously addressing the problem will guarantee successful change control. The mechanisms and procedures suggested in this paper are tools, nothing less, but certainly nothing more. The human factors are the more important considerations in successful software change control.

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

A Cost-Plus-Award-Fee Contract for a Large Software Development Program

By W. H. MAC WILLIAMS and J. E. PETERSEN

(Manuscript received January 3, 1975)

This paper describes the Cost-Plus-Award-Fee (CPAF) contract that has been used to control a major software development effort, amounting to approximately $30 million annually. The amount of the award fee is determined periodically, based on a unilateral judgment of supplier performance. The lessons learned in handling a contract of this type and magnitude are summarized. The CPAF contract has proven to be a good means of ensuring the continued attention of supplier management that is necessary for obtaining high performance on time.

I. INTRODUCTION

Put very simply, the Cost-Plus-*Award*-Fee (CPAF) contract is a cost-reimbursable level-of-effort arrangement in which the fee to be paid for each (predetermined) period is based on the customer's unilateral, subjective judgment of the supplier's performance during that period, measured against previously-agreed-upon performance criteria. The fee awarded is not subject to change. The award-fee contract differs from other types of cost-reimbursable contracts such as (*i*) the Cost-Plus-*Fixed*-Fee (CPFF) contract where the fee is fixed at the outset of work, and (*ii*) the Cost-Plus-*Incentive*-Fee (CPIF) contract, in which the fee is determined by applying a previously-agreed-upon formula to objective measurements of cost and/or performance and schedule events upon completion of the work.

The key words in award-fee are "unilateral" and "subjective." This type of contract is a complete departure from convention and one not eagerly sought by suppliers unless they have enough self-confidence to take some very real monetary risks. The motivating factor for the supplier is to maximize the profit—the all-important "bottom line"—by high performance, and the award-fee contract is a vehicle for doing so if the supplier is willing to take the risk of realizing a very small profit or none at all if he does a poor job.

In the past, CPFF contracts had often been chosen for both hardware and software development programs. The principal technical difficulties lay in communications and motivation: in getting requirements changes implemented, getting feedback on current progress and problems, and getting appropriate attention by supplier management.

Software development is characterized by many requirements changes and many complex interfaces, and one must ensure close and continued communication if a software development contract is to be successful. Furthermore, software development is a process of evolution, and it is very difficult to set up predetermined performance goals against which the final product could be measured; hence, a software objective-incentive contract is frequently not desirable.

To get the good communications and motivation that are essential in the development of software, we decided to use the Cost-Plus-*Award*-Fee method. At the time of this decision, the CPAF form of contract was relatively new, and had not even been recognized in the Armed Services Procurement Regulations (ASPR). It was being used principally by NASA and the Navy, for various kinds of work including software development, and had been well regarded by them. The concept appeared to be suitable for our major software development extending over several years, since it provided a financial incentive for good performance, and this periodic pressure of profit determination offered the best promise of continued attention by the contractor management.

Accordingly, a specific award-fee approach was devised, and proposals based on this approach were invited. A contract format was devised specifying a periodic award of fee money based on a quantitative scoring of supplier performance, using stipulated subjective criteria. Its provisions included developing a curve that would give profit in terms of score and establishing an effective procedure that would ensure prompt and continuous feedback. A selection procedure was devised, and the supplier was chosen with the knowledge that this was to be an award-fee contract.

The contract was signed on January 14, 1969 and with some modifications has been used steadily since then.

II. DETAILS

The contract has covered up to about 800 people. At the time, the work was divided for control purposes into 16 mission orders (missions) covering broad areas such as MSR tactical data processing and computing facilities. In turn, the missions were divided into some 70 tasks, with titles like "Software Quality Improvement" and "PAR Installation and Test System Development." Each mission can be viewed as an individual contract since it contains a scope of work, designates a representative for evaluating performance, sets forth the planned hours and dollars estimated to do the work, assigns a base fixed fee, and establishes an award-fee pool that can be earned in whole or in part depending on the evaluated performance. Thus, the entire contract is essentially a collection of mission orders handled within a common procedural framework.

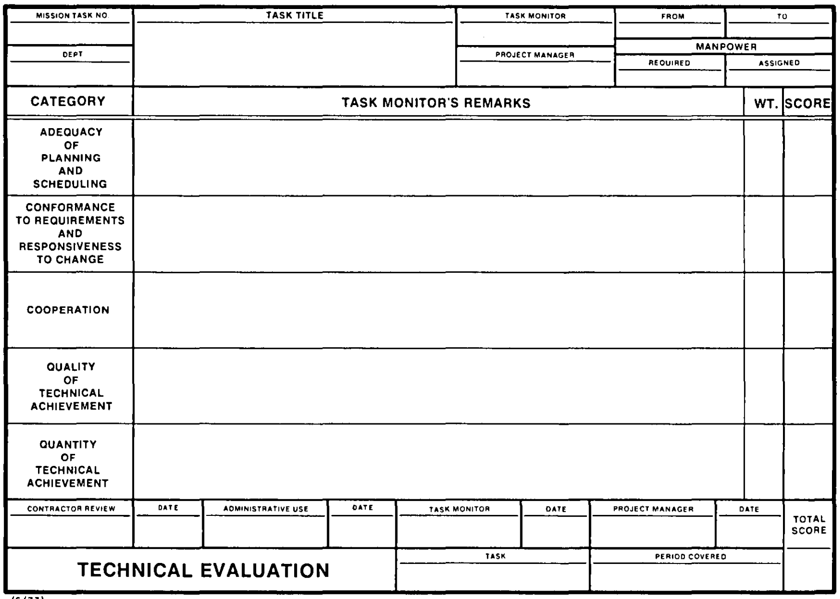
2.1 Evaluation

Evaluation of the tasks and missions is a key part of the administration of the contract and, accordingly, it has been structured carefully. For convenience, the several steps in the evaluation procedure are summarized in Table I.

Each task is defined by a specific task plan, and is monitored by a designated Bell Laboratories member of technical staff, usually a first-line supervisor. Once each month, this task monitor evaluates the performance of the supplier on his task by means of a formal set of scores, supplemented by a written commentary that notes prominent strengths and weaknesses observed during the month. The monitor, based on his subjective judgment, assigns a score between a minimum of 59 (a failure) and a maximum of 100, about the interval of a typical school report card. A score of 80 will return to the supplier a fee commensurate with what would be expected for a good-quality job on a CPFF basis. The technical evaluation form is shown in Fig. 1, and the definitions of the categories in Fig. 2. Each technical evaluation is reviewed and approved (possibly with changes based on mutual discussion) by the project manager, who is the task monitor's supervisor.

Table I — Summary of evaluation procedure for CPAF contract

Frequency	By Whom	Functions
Monthly	Bell Laboratories Task Monitors	Technical evaluation of tasks.
Monthly	Bell Laboratories Project Managers	Management evaluation of tasks in a mission. Due the 5th of the month.
Monthly	DPS Control Department	Calculates scores for all tasks. Sends preliminary evaluations to supplier as of the 12th of the month.
Quarterly	Performance Evaluation Board	Reviews evaluations. May make score adjustments. Recommends scores to fee-determining officer.
Quarterly	DPS Control Department	Adjusts scores as recommended. Makes errata sheets.
Quarterly	Fee-Determining Officer	Reviews recommended scores. Makes final decision on scores. Determines fee. Sends official evaluation to supplier.
Quarterly	Management Review Board	Discusses evaluations (and reviews contract work generally).



MISSION TASK NO	TASK TITLE	TASK MONITOR	FROM	TO
DEPT		PROJECT MANAGER	MANPOWER REQUIRED	MANPOWER ASSIGNED

CATEGORY	TASK MONITOR'S REMARKS	WT.	SCORE
ADEQUACY OF PLANNING AND SCHEDULING			
CONFORMANCE TO REQUIREMENTS AND RESPONSIVENESS TO CHANGE			
COOPERATION			
QUALITY OF TECHNICAL ACHIEVEMENT			
QUANTITY OF TECHNICAL ACHIEVEMENT			

CONTRACTOR REVIEW	DATE	ADMINISTRATIVE USE	DATE	TASK MONITOR	DATE	PROJECT MANAGER	DATE	TOTAL SCORE
TECHNICAL EVALUATION				TASK		PERIOD COVERED		

(6/73)

Fig. 1—Technical evaluation form.

THE BASIC REFERENCE IS THE TASK PLAN.

PLANNING AND SCHEDULING – QUALITY OF PLANNING AND REPLANNING, MEASURING AND PROJECTING PROGRESS, SCHEDULING, AND ALLOCATING RESOURCES DURING THE REPORT PERIOD (NOT NECESSARILY HOW WELL THEY ADHERED TO PREVIOUSLY ESTABLISHED SCHEDULES).

CONFORMANCE TO REQUIREMENTS AND RESPONSIVENESS TO CHANGE – DEMONSTRATED ABILITY TO MEET DESIGN REQUIREMENTS AND KEEP WORK IN LINE WITH PROJECT GOALS, EVEN IF CHANGING.

COOPERATION – IN ADDITION TO THE USUAL MEANING, PROMPT FURNISHING OF ALL DATA ON ANY PROBLEM AREAS THAT COULD IMPAIR PERFORMANCE OR OTHERWISE AFFECT TASK PERFORMANCE.

QUALITY OF TECHNICAL ACHIEVEMENT – IMAGINATIVENESS, ACCURACY, COMPLETENESS, RELIABILITY, AND APPROPRIATE OPTIMIZATION OF DESIGN AND IMPLEMENTATION. FOR EXAMPLE, EFFICIENCY OF CODE IN REGARD TO THE USE OF TIME AND SPACE, COMPLETENESS AND TECHNICAL AND EDITORIAL QUALITY OF REQUIRED DOCUMENTATION, INITIATIVE, IDEA GENERATION, AND GENERAL APPROACH TO THE JOB.

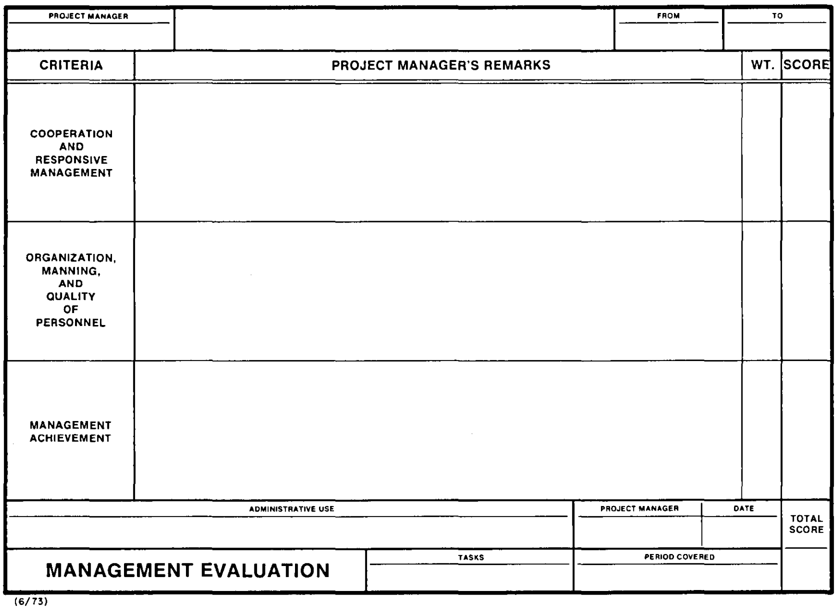
QUANTITY OF TECHNICAL ACHIEVEMENT – PRODUCTIVITY IN DESIGN AND IMPLEMENTATION OF PROGRAMS AND PRODUCTION OF DOCUMENTS. OVERALL AMOUNT OF USEFUL WORK ACCOMPLISHED DURING THE PERIOD.

MANPOWER:

REQUIRED – THIS NUMBER IS TAKEN DIRECTLY FROM THE TASK PLAN AND REPRESENTS THE MANPOWER PLANNED FOR THE MONTH SPECIFIED, IN EQUIVALENT FULL–TIME PEOPLE.

ASSIGNED – THIS NUMBER IS DERIVED FROM TOTAL MAN–HOURS (INCLUDING OVERTIME) REPORTED DIVIDED BY THE NUMBER OF HOURS IN THE ACCOUNTING MONTH. HENCE, THIS NUMBER REPRESENTS EQUIVALENT FULL–TIME PEOPLE.

Fig. 2—Definition of technical evaluation categories.



PROJECT MANAGER		FROM	TO

CRITERIA	PROJECT MANAGER'S REMARKS	WT.	SCORE
COOPERATION AND RESPONSIVE MANAGEMENT			
ORGANIZATION, MANNING, AND QUALITY OF PERSONNEL			
MANAGEMENT ACHIEVEMENT			

ADMINISTRATIVE USE	PROJECT MANAGER	DATE	TOTAL SCORE
MANAGEMENT EVALUATION	TASKS	PERIOD COVERED	

(6/73)

Fig. 3—Management evaluation form.

Separately, each project manager evaluates the tasks for which he is responsible within a given mission by means of a management evaluation (see Figs. 3 and 4). Technical scores are calculated, based on a weighting of the categories that varies with the individual tasks. Management scores are also calculated, but with a uniform weighting that is the same for all missions. The entire body of monthly evaluations is sent to the supplier soon after the start of the succeeding month, and face-to-face discussions ensue shortly thereafter.

COOPERATION AND RESPONSIVE MANAGEMENT – QUALITY OF ACCURATE AND OBJECTIVE EVALUATION OF THE IMPACT OF REQUIREMENTS AND CHANGES. FURNISHING DATA, INFORMATION, AND ADVICE ON KEY PROBLEMS, AND MAKING TECHNICAL AND ADMINISTRATIVE CHANGES AS REQUIRED.

ORGANIZATION, MANNING, AND QUALITY OF PERSONNEL – ESTABLISHING AND MAINTAINING HIGH QUALITY PERSONNEL AND A USEFUL ORGANIZATION WHICH INTERFACES CONVENIENTLY WITH THE LABORATORIES, AND MEETING CONTRACT MANPOWER REQUIREMENTS.

MANAGEMENT ACHIEVEMENT – QUALITY AND QUANTITY OF USEFUL OUTPUT. MAKING EFFECTIVE USE OF PERSONNEL, CONTROLLING THE USE OF RESOURCES, FILTERING OUT INESSENTIAL WORK, AND PROPERLY USING AND CARING FOR FACILITIES AND EQUIPMENT.

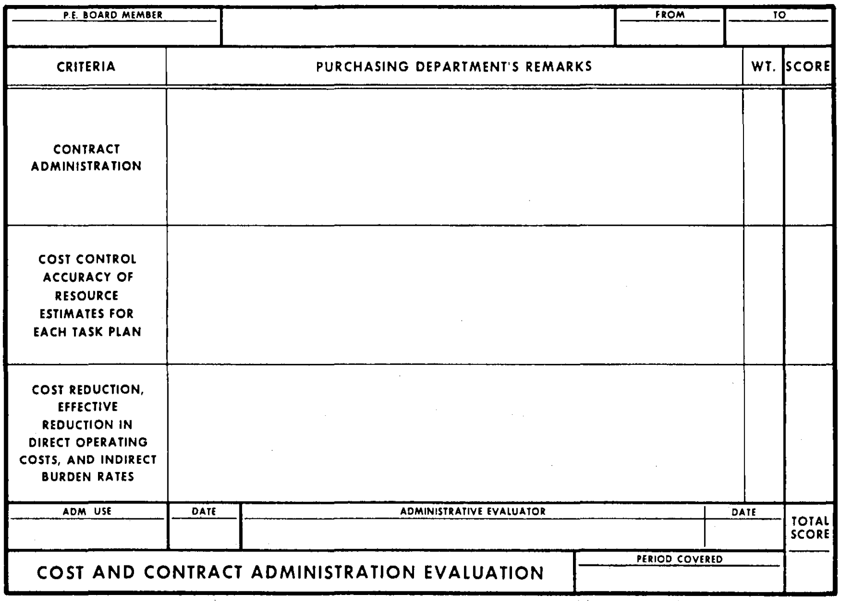
Fig. 4—Definition of management evaluation categories.

2.2 Review

Once a quarter, the evaluations for the entire three months are reviewed by the performance evaluation board, which consists of the project managers, the senior management of the division, and the head of the local contracting (purchasing) department [who enters into the scoring equation his own evaluation of cost and contract administration (see Fig. 5)]. All evaluations are scrutinized and reviewed for fairness and appropriateness of category. The project managers are permitted to change the evaluation scores if they consider it necessary, based on subsequent information of events during the quarter that had not been available at the time of the evaluation, provided they can justify the changes to the board's satisfaction. The board is permitted to change scores to reflect a broader view, and comments are frequently made in the minutes of the review meeting that draw attention to a strength or weakness or emphasize a particular problem. Normally, the changes in scores are few, and are made only for a specifically explained reason. The board then recommends, to the fee-determining officer, a set of scores by mission for the quarter.

2.3 Fee determination

The mission scores are converted to mission fees according to an essentially linear algorithm, with 59 corresponding to the base fixed fee (if any) or 0 percent award and 100 corresponding to the maximum



P.E. BOARD MEMBER		FROM	TO

CRITERIA	PURCHASING DEPARTMENT'S REMARKS	WT.	SCORE
CONTRACT ADMINISTRATION			
COST CONTROL ACCURACY OF RESOURCE ESTIMATES FOR EACH TASK PLAN			
COST REDUCTION, EFFECTIVE REDUCTION IN DIRECT OPERATING COSTS, AND INDIRECT BURDEN RATES			

ADM USE	DATE	ADMINISTRATIVE EVALUATOR	DATE	TOTAL SCORE

COST AND CONTRACT ADMINISTRATION EVALUATION	PERIOD COVERED

Fig. 5—Cost and contract administration evaluation.

fee of 15 percent which would include the base fixed fee. In military work, the base fixed fee cannot exceed 3 percent nor can the maximum fee exceed 15 percent for R&D work (including the base fixed fee, if any). The fee-determining officer (the local director of purchasing) then reviews the scores from the purchasing point of view; he is empowered to change the fees if in his sole judgment it is appropriate. He then forwards the official copy of the quarterly evaluations to the supplier, together with any score changes and performance evaluation board minutes, and with the fees for the quarter.

2.4 Supplier review

A contractor/supplier review is normally held quarterly, at the supplier's request, by the management review board, which consists of officials of the supplier, technical contractor personnel, and the fee officer, to discuss the evaluations and pertinent technical and management questions. This review may be waived by the supplier.

III. DISCUSSION

Experience has shown that firm customer management support must be given to the process of evaluating the work and reviewing the evaluations. This involves many people; for example, in July 1971, when the job stood at 14 missions and 54 tasks, the customer monitoring involved (part-time) 44 task monitors and 17 project managers. These numbers may suggest an inordinate amount of monitoring; however, this is not the case, since in a program of this magnitude one would expect to have roughly this number of customer technical people involved to ensure a good product. The distinctive feature is the coordinated evaluation effort of these people. There is a tendency for the evaluation process to become routine and thus to lose its incisiveness. This must be guarded against continuously, by vigorous top-management interest, principally at the quarterly performance evaluation board meetings. Not only must the evaluations be incisive, they must also be timely. In any busy organization, there is a tendency for paperwork such as these evaluations to lag. This must be prevented, since prompt feedback with the supplier is essential.

The evaluations must be carefully and thoughtfully done. In the course of reviewing a great many evaluations, some DOs and DON'Ts have been formulated. Since these have come from hard experience, it is appropriate to include them here.

(*i*) Make the task plans clear and concise.
(*ii*) Encourage initiative.
(*iii*) Ensure that the score represents the exact evaluation of the supplier for the period.

(*iv*) Say *why* you thought the work was good or bad. If the score is very high or very low, always include an explanation. Make the remarks constructive, so that they may be used to maximize supplier performance.

(*v*) Task monitors should discuss evaluations face-to-face with supplier counterparts.

(*vi*) Jot down comments as the month proceeds.

(*vii*) Get completed forms in on time, to permit quick feedback to the supplier.

(*viii*) Don't use an unsupported adjective:
NOT "Good"
BUT "Good replanning to accomodate a peak work load."

Some problems were observed from the supplier's point of view. Some supplier managers felt that task monitors were arbitrary in their scoring, and sometimes they tended to please the task monitors rather than exerting their own judgment on how best to do their jobs. At times, requirements changes made supplier managers uncertain as to the customer's needs and made them regard evaluations as unfair. In the main, these problems were growing pains, and disappeared as higher management review was applied to the evaluations.

A frequent question is "Are you paying a proper fee for the work?" The answer is that, if you need high performance on time, then the value of a high-quality job more than compensates for a higher fee. If the proper evaluation of the supplier's performance results in a high fee, then by definition you must be receiving the kind of product you desire. And such is the case with the contract under discussion.

IV. CONCLUSIONS

The award-fee contract discussed here has been in operation for more than four years and has covered, as of October 1974, over $130 million of effort. The gaps in communication have been few, and by and large they have been spotted and corrected promptly.

In summary, the award-fee contract is a good vehicle for dealing with a large, complex, dynamic problem, where the customer needs as good a job as he can get and on time. This type of contract requires good faith between customer and supplier and a substantial monitoring and evaluation effort. The format encourages good customer-supplier communications and the active management involvement that is, in fact, necessary to successful performance. The improved visibility of problems makes it possible to address them quickly and solve them. The CPAF contract format has played a very important role in getting high-quality software on schedule in a major software development.

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Software Project Management

By J. D. MUSA and F. N. WOOMER, JR.

(Manuscript received January 3, 1975)

A broad-based study of software project management for the SAFEGUARD project is presented. SAFEGUARD posed unprecedented project management challenges because of its size and complexity, yet the project was successful in attaining its objectives. This account of what some of the challenges were and which approaches were most effective in meeting them will hopefully suggest guidelines for the management of other software projects. Subjects include planning, methods for gathering status information, control actions, requirements, and programming methodology; also, differences between managing a software and a hardware project are explored. This study is based on intensive semistructured interviews with 26 SAFEGUARD software managers at all levels concerning their experiences on the project and opinions derived from them. The views expressed are limited to those of the individual managers interviewed and do not represent a consensus of SAFEGUARD management.

I. INTRODUCTION

For the purpose of this paper, "project management" is defined as *work planning and scheduling*; *gathering and reviewing status information*; and *controlling and allocating* human, computer, financial, and time *resources* so as to meet project objectives. Control consists of a continuing series of corrective actions resulting from status reviews. Project management differs from management in general in that technical questions or individual personnel matters are not considered, except where they might affect the areas within the definition of project management.

II. METHOD OF STUDY

Most accounts of management experience on a project are relatively personal ones. In this case, it was decided to obtain a broad perspective by interviewing a cross section of 26 software managers about their experiences. The interviews included subcontractor managers and involved every level of management from first to fourth; and the participants held a wide range of jobs. There were varying degrees of background in software and management, and widely differing management philosophies.

A discussion lasting approximately two hours was held with each manager. An interview guide was sent to each participant one week before the interview. One-third of this guide contained background material; two-thirds, questions. The guide prepared the managers for the interviews, giving them time to consider the topics. Each interview was semistructured in the sense that the open-ended questions of the guide were generally followed; however, digressions into related topics were also encouraged.

The authors looked for patterns in the responses and held follow-up discussions with the managers based on the initial draft of this paper.

III. RESOURCES DEVOTED TO PROJECT MANAGEMENT

To gauge the importance placed on project management activities, each manager was asked what percentage of his organization's resources he was willing to allocate to this function. Estimates ranged from 5 to 25 percent, with the average about 12 percent. To get a job running smoothly, it was generally believed that more resources (up to 50 or even 100 percent) should be devoted to project management in the early stages of a job. On SAFEGUARD, the rapidly changing environment necessitated a high level of planning effort extending into the late phases of the job. Several people believed that the proportion of effort required for project management is larger on large projects; this was especially so on such a complex project as SAFEGUARD.

IV. PLANNING

System requirements were first defined in 1969 in a system concept paper; by the third quarter of 1969 they were specified at a detailed system engineering level in the data-processing system performance requirements.[1] During the same time period, a complete software development plan and schedule was prepared, along with the rationale on which it was based. This overall plan was followed by extensive planning in more detail at lower levels, much of it stimulated by the requirements of the Management Reporting System (MRS). The MRS is described in Section 5.2.

One of the most difficult challenges was establishing the proper time relationships between different parts of the job. When this was not done realistically, simultaneous design and coding often resulted; this was inefficient because then interfaces were not ironed out and the feasibility of algorithms was not investigated (and they finally had to be). It was found that the timing of test planning activities was fairly critical. The optimum time appears to be approximately when program design is complete but coding has not started, since a good test plan should have both requirement- and implementation-oriented components. The point in time at which users had enough knowledge about their data reduction requirements to specify them in detail and the point in time at which the requirements were needed by the data-reduction system designers were usually incompatible. It was very hard to schedule system evaluation, since the design had to be far enough along to provide definitive information but the evaluation had to be completed early enough so that problems could be identified and corrections implemented before the design was frozen.

Several managers would have placed more emphasis on centralized planning. They suggest that a group, reporting directly to the highest-level manager, should be responsible for overall planning and schedule control. A formal development plan for the entire SAFEGUARD project had been prepared; however, almost everyone felt that this development plan was useful mainly for introductory orientation.

Despite a general feeling that planning is important, a slight majority of managers did not think it necessary to prepare written, explicit schedule and resource plans for their own areas. Most of them do not enjoy such planning; however, this distaste does not extend to technical planning, such as the generation of requirements and test plans. Those who feel that a development plan is necessary generally suggest a written plan (with format left to the author) plus supplementary bar charts. Most managers believe the PERT networks are not worth the effort required.

Accurate estimation of time, manpower, and computer time required for a job was found difficult by most managers. Estimation accuracy was not significantly affected by the size of a job. Several managers commented that they found it difficult to evaluate schedule performance because the work often changed after the original plan had been created. One noted that although computer time usage was the hardest resource to estimate, it was the least critical of all the resources to manage (except when "turnaround" time deteriorated).

When errors in estimating occurred, the prime sources of error were found to be neglecting learning time and underestimating the lengths of test intervals and the amount of maintenance support required. One manager felt that from 25 percent to 50 percent of the peak manpower of a project must be "kept" for the maintenance phase. The lower figure represents the manpower required to correct "bugs" but not to implement any changes, and the higher figure represents the manpower required to deal with a fairly high change rate.

Several participants felt that a better basis for estimating a job was previous experience on similar jobs rather than the use of numerical planning algorithms. Hence, their approach was to let their subordinates use their own direct experience to do the estimating and then question them on it.

V. GATHERING STATUS INFORMATION

Gathering status information is one of the most important project management activities. There was a fair amount of diversity in establishing the criteria that were considered to be most important in evaluating the worth of a particular report or information-gathering technique; however, timeliness and accuracy stood out. Definitiveness (providing enough information to identify the accomplished work unambiguously) ranked next. Several managers indicated that, above all, a report or technique should make problem areas visible. Conciseness was considered useful, but completeness ranked fairly low. Some minor considerations were flexibility and understandability. It was felt that asking the question, "What actions can I take as a result of this report?" was a good approach to measuring its value.

A very strong pattern of preference for informal methods emerged, with primary reliance on *oral* rather than *written* communications. This preference was found to be independent of management level. One argument advanced for oral reporting was that the useful life of status information is short. Another advantage of oral communication is that it permits questioning and probing and rapid interaction with feedback. Also, many managers believe that they can evaluate the reliability and accuracy of oral communication better than that of written communication because they can observe the way in which a respondent answers questions. Lower-level managers prefer making oral reports because it takes less time and involves more personal contact with their superiors. The main weakness of oral communication is that for higher-level managers it may be either indirect (i.e., received secondhand) or time-consuming.

Some managers found conventional written progress reports moderately useful as reminders for low-priority items or for keeping up to date on activities in peripherally related organizations, but most believe that written reports have a low density of useful information. (They may be inaccurate, heavily filtered or censored, or out of date.) Written reports usually only formalize communication that has already occurred. Finally, the formality of writing does cost time and money. This cost could be such that the need for any written report should be periodically challenged.

The types of information-gathering techniques used by different levels of management were generally not very different. There was, of course, less need for detail at the higher levels. Higher-level managers were more interested in tracking the status of capabilities; lower-level managers more interested in components. Lower-level managers were very well satisfied with the information available to them; however, higher-level managers occasionally felt that some of their needs were not met. Most of the attempts made to provide better information to higher-level managers involved written reports. They apparently were only partially satisfactory in meeting the needs, since higher-level managers still primarily relied on oral communication.

The contractor's low-level managers had mixed attitudes about technical involvement of high-level managers. They appreciated the understanding that resulted, but believed that frequently there was too much concern with detail. They felt that this concern indicated a lack of trust and that it restricted their freedom to do the job in the way they thought best. It was suggested that higher managers sometimes dipped into detail simply because they enjoyed keeping technically involved.

High-level managers, in general, recognize some of the dangers that their subordinates cite and realize that there may be disadvantages in their concern with detail. They also know that keeping informed is expensive in terms of demands on the time of their subordinates. However, one of them pointed out that perhaps the disadvantages must be accepted as concomitants to the drive for technical understanding and that the advantages on the whole won out. One advantage, for example, was that technical understanding permitted rapid evaluation of situations and made for more immediate decisions.

Reporting techniques did not change much as a result of changes in the phase of a job, excluding reports obviously tailored for a particular phase, except that there was some tendency for more detailed information to be needed in the later stages. The program-design, code, and unit-test phases of a job, characterized by a large number of parallel efforts, were the most difficult to track. Attempts at numerical characterization of status were not always completely successful because available measuring units are generally nonuniform and are not sufficiently descriptive of problems. The system analysis and system design phases can be tracked by observing the status of the requirements and functional specification documents, and the system integration test phase by the completion of tests of various capabilities. There is a clear need to find a way of defining more intermediate milestones of a specific nature for the program-design, code, and unit-test phases.

It was noted that many persons receiving reports would like to have had direct control of the level of detail and format. In some cases, reports satisfied the needs of several people, but these "communities of interest" were generally small in size and on the same managerial level.

The persons making reports were sometimes unhappy about the duplication of data between reports, resulting from lack of standardization. This situation needs to be recognized as being generally unresolvable; good compromises may not always be possible.

Most participants felt that they needed information on a weekly basis in their area of responsibility. Information more than one week old has limited usefulness for solving problems. A monthly interval is considered sufficient for overall project status information external to one's area of responsibility and for computer usage and manpower and financial reports. (It was noted that the reason for the less frequent reporting of financial status was that dollars usually cannot be controlled very rapidly on a project.) Several participants believe that one needs daily information on a few crucial items, particularly if a problem exists, or during the final stages of a project.

5.1 Discussion and meetings

Informal individual discussions were by far the most commonly used data-gathering method. Managers found that individual discussions were generally more effective than meetings. However, there is often not enough time to talk about a problem with all the individuals concerned.

Meetings, usually held on a weekly basis, were the second most common data-gathering technique. Most managers found that meetings were more successful if they lasted no more than $1\frac{1}{2}$ to 2 hours, with an agenda prepared beforehand. A majority agreed that someone should be assigned to record and publish action items generated during a meeting. It was also agreed that specific problems should usually be "delegated" for later solution.

It was found that there are several common problems that occur in meetings that a manager must learn to deal with. Occasionally, most of a meeting may be taken up with "educating" high-level managers. Some participants may make contributions merely to make an impression. In other cases, meetings can become forums in which one manager tries to shift the responsibility for a problem to another manager. People may make unnecessary efforts to dig up problems just because they believe their managers expect them to bring them up. It was agreed that discipline should be exercised as to the frequency, length, and size of meetings; the principle of representation, rather than that of full attendance of all parties, should also be followed.

5.2 Written reports

Trouble Reports (TRs) were generally considered to be the most valuable written source of information on the project. Fairly early during the SAFEGUARD project, the concept of writing a TR for a program malfunction was generalized to permit such reports to be written on documentation and requirements as well. This idea was apparently very well received, and TRs were widely used to report and record all sorts of discrepancies. After a program had been turned over for integration, the tracking of these reports was the method most commonly used by first-line supervisors for keeping track of program status. Several managers initially preferred local TR accounting systems to a centralized system, since speed in gathering information and meeting the differing needs for detail of different groups were their most important objectives. However, later in the project a centralized system was established which met these goals. Program-review boards were set up in several areas to evaluate TRs and approve or disapprove suggested program changes; they were found to be very effective.

A computerized Management Reporting System (MRS) was developed for use on the SAFEGUARD project. The system incorporated data bases for schedule, manpower, and computer usage information. The schedule data comprised some 3000 items, with information on the scheduled and estimated start and completion dates of various significant activities and events. Status information was provided, as well as indications of dependencies between the various items. Error-checking and data-interrelating capabilities were incorporated. On a monthly basis, information was updated and a standard report published. In addition, sorting and retrieval capabilities were provided so that a wide variety of special reports could be produced, on an as-requested basis.

The success of this system was mixed. Several managers, particularly at the higher levels, felt that MRS was of significant help in structuring and planning the project, in that it forced both long-range planning and the coordination of plans between different areas. On the other hand, the three-week time lag between gathering information and publishing it was considered too great. Experience indicated, however, that gathering, publishing, and distributing this much information (approximately 850 pages every month to 70 managers), with high standards of accuracy and good coordination of dependencies and dates across interfaces, can probably not be speeded up to any significant degree. Several managers did not like the discipline forced upon them or the background information lost in a fixed, computerized reporting system. They believe that managers should be allowed to choose the reporting method best suited for describing the status of their work.

A Principal Events Reporting System was developed which identified and carefully defined a number of important milestones on the SAFEGUARD project. Many of the events listed were the completion of tests of various functional capabilities. Status reports on these events were made by TWX within 48 hours of their scheduled completion dates. Estimated dates were given for completion of late items, and follow-up TWXs were sent on the rescheduled dates. A principal events report, which described and gave the status of all items, including previously completed events, was issued at quarterly intervals. This system was primarily valuable to the customer and to higher levels of management; the TWX reports were particularly useful.

Program unit reports (reports indicating the status of the smallest program units and data sets in terms such as "design complete," "coding complete," "unit test complete," etc.) were considered to be of little value by most managers.

The weekly TWX status reports sent between the test sites and the contractor's home location were considered to be useful because of their timeliness and conciseness. The almost universal recognition given to the need for writing and handling a TWX expeditiously ensures that the information is timely. Furthermore, a TWX progress report must be concise; this ensures that only the most important items get reported. This seems to indicate that if a system of written progress reports is to be of any value, it should be severely constrained in both preparation time and length. (It might actually save money to *require* all written communication to be sent by TWX.)

Several managers used wall schedule charts but all eventually abandoned them as not being very useful, except possibly for initial planning. The majority felt that their wall charts did not shed any particular light on critical issues.

Managers primarily employed computer usage reports to spot trends or to make budget projections rather than for control. Manpower usage reports were not employed in manpower allocation decisions because so many other factors were more important.

Financial reports were used for reporting on expenditures. Several managers felt that more detailed information should have been provided as to the sources of the data and the date on which it was valid.

VI. MANAGEMENT REPORT ANALYSTS

One innovation that was introduced on the SAFEGUARD project was the assignment of a staff assistant, called a Management Report Analyst (MRA), to each second-level manager. The MRAs were, in general, college graduates with backgrounds in planning, scheduling, and budgeting. Besides acting as general "right-hand assistants," they aided the managers with budget preparation and control, planning and scheduling, interfacing with overall project management reporting systems and following up on action items. Almost all of the managers who were assigned MRAs were enthusiastic about their usefulness. A high-level manager noted that MRAs made it possible to accelerate schedule-information flow. Some of the managers felt that the MRAs saved them time by buffering them from duplicate requests for information.

There seemed to be no need to assign MRAs to first-line supervision on a full-time basis; sharing the MRA assigned to the second-level manager was satisfactory. One high-level manager said that he felt that the usefulness of MRAs was such that they also should have been assigned to the third-level managers. It might be noted that the fourth-level managers were already assigned staff assistants.

The comment was made that MRAs were most beneficial when they were assigned to report directly to a second-level manager rather than to a central group supervisor. However, some managers thought it was best to centralize the physical location of the MRAs so that there could be interchanges concerning common problems, solutions, interfaces, and action items. Also, centralized training was felt desirable.

VII. CONTROL

Control relates to the corrective actions that result from the process of comparing status to plan. The size and complexity of the SAFEGUARD project made prediction of the likely consequences of various actions difficult at times, complicating the process of selecting from among the alternatives.

Good organization, adapted continuously to changing job requirements, was found essential to the successful implementation of management actions. As a by-product, it was observed that the amount of status reporting and communication required was substantially reduced when the organization was well fitted to the tasks to be accomplished and responsibility was not divided. Interface and system-coordination departments had to be in the mainstream of activity and authority to function effectively. Some managers would have created a more clearly hierarchical organization. This might obviate the problem (common among managers with strong technical orientation) of multiple levels of management working on the same problem at the same level of detail. Others felt that *conflicting* needs in a complex project require *conflicting* organizations; consequently, informal organizations and informal channels of communication should be encouraged.

Most of the managers believe that interfaces on the project were handled well or very well, but many agreed that specific improvements could have been made. Some of the interface areas that were initially significant sources of problems were the contractor-subcontractor interface; the interface between support software users and designers; the interface between system engineering and development; and the interface between users and support service activities, such as the computation center.

Most managers found that personal contact and meetings are the best ways to coordinate interfaces; letters of agreement were considered to be relatively ineffective (partly because of reorganizations), although they were of some value when used with subcontractors. Even where organizations are geographically remote, personal contact is preferred. (For example, coordination with even our Pacific site was found to work much better by phone than by TWX or letter.) It is recommended that documentation should be used only to confirm and record agreements after the fact.

7.1 Control of time

Most managers noted that although only slight deviations from any scheduled end dates were acceptable to their superiors (the average of estimates of acceptable deviation is five percent), they had (and should have) relative freedom to change intermediate schedules.

When tasks could not be finished according to plan, the most commonly employed strategies for recovery were to work personnel overtime or use extra computer time. Increasing manpower on the job or decreasing the scope of the task were secondary choices. Slipping schedules, decreasing or deferring documentation (surprisingly), and decreasing the quality of testing, in that order, were considered to be increasingly undesirable. (One manager observed that decreasing testing is a very bad option because people have a way of forgetting they agreed to reduce testing when a program doesn't work.)

Several managers found that overtime was ineffective except in urgent situations, because people tended to get stale. This is felt to be particularly true in creative jobs.

Many of the managers concluded that adding manpower to a job is usually not a useful technique. Even if budgetary constraints do not exist, suitable people usually are not available at the critical point of the job. Adding people generally hurts the effort because of the training required at such a late stage in the project. (A few areas did add people effectively late in the project but they had special skills as trouble shooters.)

Managers generally determined that, when schedule changes are necessary, it is best to consider all the inputs at one time and do a complete revision. Complete revision permits the changes to be carefully thought out and documented. (One high-level manager said he felt that some of his principal contributions to the project were vetoes on changes.)

It was generally agreed that the MRS approach to controlling schedule dates was a good one: high-level managers controlled the schedule dates for events, while the supervisor responsible for an event provided his own estimated date. Many managers believe that a subordinate should only be held to "end" dates (i.e., the dates at which deliveries to another organization must be made). A subordinate should be required to make his intermediate dates visible, but should be permitted to modify them as he desires.

7.2 Control of human and computer resources

Allocating resources efficiently was found to be most difficult at the higher management levels. It was hard to gain detailed insight into how various activities contributed to the real goals of the project. Activities, particularly in the support area, tended to go on "forever" unless their value was questioned. Most lower-level managers considered that they had the knowledge and the freedom to allocate manpower and computer resources productively within their own areas.

There is general agreement that selecting good people was the factor of greatest importance in the success of SAFEGUARD. Selection must be considered not only as an initial choice but also as the continuing process of assigning people to jobs and problems. The generally flexible, informal management style that prevailed on SAFEGUARD aided management greatly in this process, in that there was a lot of "self-selection." Large numbers of nonmanagers exercised initiative in digging out and solving problems. The strong technical capability of project members occasionally led to excessive technical discussion, design polishing, and uneven work coverage due to a concentration on technically interesting problems (to the detriment of important but less rewarding ones). These difficulties were small, however, compared to the advantages.

Although many of the programmers were inexperienced, there was a cadre of people with three or four years experience (at the start of the project) who became the lead programmers and in some cases first-line supervisors. Previous background of the contractor, background gained with similar systems, was also valuable. Inexperience occasionally resulted in some errors of judgment. However, one of the surprises of this study was that inexperience had a relatively minor overall effect on the project.

The shortage of experienced software managers on the project posed a more serious challenge than the shortage of experienced programmers. It was found that if there was not enough supervisory attention given to programming, both efficiency in attaining objectives and quality of output sometimes suffered.

A "software mystique" can discourage managers who have no software background from applying some of their general managerial skills; hence, many skills may be lost.

Turnaround time was the key parameter that was monitored in the control of support computer resources, since it had maximum impact on schedules, overtime costs, and programmer satisfaction. Allocation of support computer time was found to be worthwhile only when turnaround time deteriorated, i.e., it did not pay on a regular basis.

VIII. REQUIREMENTS

Overall success in various areas of the project was considered to be strongly correlated with the degree to which project system requirements were clearly and completely defined, written, and stabilized. It was found to be necessary to focus on the feasibility of implementation of requirements and routine details such as interface coordination as much as on the requirements themselves. (Simulations of algorithms on a support computer were determined to be very beneficial.) It proved to be very challenging to pick the right level of specification of detail without unduly restricting the designer's freedom to apply his special competence. It was suggested that it might be useful to have a high-level requirements document for the customer and a more detailed one for the developer. One pitfall to avoid is the incomplete requirements specification, particularly with inexperienced personnel, who often will not recognize the deficiency early enough to request timely corrective action.

Some of the areas that the development managers believe should have received more attention in the system requirements are:

(*i*) Error control.
(*ii*) Interface specifications.
(*iii*) Requirements for equipment tests.
(*iv*) Maintainability, reliability, and availability.

Coordination of software requirements with hardware changes was found to be very important.

One suggested way of achieving greater focus of system engineering on implementation is to place senior designers in the systems groups during the first part of the project, so that they can make the systems engineering people more aware of their detailed needs. As a system is defined and as more detailed development starts, designers may return to their development groups.

IX. PROGRAMMING METHODOLOGY

Another surprise in this study was that the lack of well-developed methodology in the programming art did not turn out to be a serious problem. Managers did comment that lack of an established technology meant that extra time had to be spent in experimentation, which impacted on cost and schedules.

There are a number of things that managers believe that they learned in the area of programming methodology. Several managers noted the need to keep debugging and testing in mind during design. For example, one must consider recording requirements and clarity of code. The idea of holding design reviews with flowcharts was a technique that many managers tried and found very beneficial. A number of managers became strongly convinced of the virtues of structured programming, considering it to be extremely important in the maintenance phase.

Another attitude that changed during the course of development was the attitude toward patches. It was originally planned to make all changes by altering source code, so that program listing documentation could be kept under good control. It was found, however, that the amount of recompilation and relinking necessary for a large program made this approach impractical for priority changes or for fixing bugs during the testing process. It was more practical to place "alters" in the source code and recompile and relink at less frequent intervals. Consequently, it became very important to provide good patch facilities and good procedures for documenting patches and keeping them under control.

Many managers found that the best documentation for programs was a well-commented listing. This represents a change of opinion on the part of a number of managers who had first seen some value in flowcharts and narratives, but who later found that few people used them in the maintenance of programs. Flowcharts appear to be better design tools than program-maintenance tools. Narratives appear to be of value primarily to system evaluators.

Several managers believe that more attention should be given to developing good unit test tools early in the development cycle. This philosophy of independent testing (i.e., test cases generated and tests conducted by groups other than the original design organizations) was widely used on the project and in general was quite successful.

There is general agreement that using a local "duplicate" computer facility for checking out programs prior to shipment to site was not only useful, but was in fact necessary to the success of the project. Using a support computer for simulations of algorithms, system performance measurements, etc., was very effective.

There were some comments that standards for programming practices should be specified to a uniform level of detail. Several managers believe that standards should be *function*-oriented rather than *format*-oriented; i.e., different formats should be permitted provided they satisfy the objectives of the standards.

X. DIFFERENCE BETWEEN MANAGEMENT OF SOFTWARE AND HARDWARE PROJECTS

All participants were asked how managing a software project differs from managing a hardware project. In general, about half the managers believe that there is nothing essentially different, and the other half see differences ranging from minor ones to everything being completely different.

The largest number of noted differences occurred in the area of development methodology. It was observed that there are more variables in software than in hardware development. It was felt that good engineering tools plus the constraints of physics will ensure a reasonable hardware product, but good programming tools will not ensure a good software product (it appears to be difficult to set up enough worthwhile constraints). Programming is more of an art than a science at present, and software design is often influenced by a personal approach to the problem. There seems to be a much tighter coupling between software development and the entire system-integration process than between hardware development and the system-integration process. This has been caused not only because logic and control is mostly implemented in software, but also because hardware lead-time constraints force system trade-offs to be made with software to a greater extent than with hardware. Software design was considered to require more time because software is usually more complex. However, in hardware design, each element is usually more thoroughly designed because it will be mass-produced.

The management of changes was another large area of noted difference. It was considered that software is generally more volatile than hardware and is more vulnerable to external influences. Several managers believe that for software there is less understanding of the impact of change on schedules and costs on the part of customer, manager, and programmer. The lead time required for changes in hardware is well recognized, but it is not in software.

In the area of personnel, it was noted that technical competence is both much more important and harder to evaluate for software because the design technology is not well developed. There is a shortage of people with both the programming and hardware backgrounds necessary to a good system perspective.

Quality control is at present more difficult for software; the criteria for success (what is a good program?) are less well defined. The volume of a worker's output is much greater in software and, thus, a supervisor cannot examine his work very extensively. (For example, a typical hardware group might produce 12 hardware circuits in a year in contrast to 30,000 instructions from a software group.)

Other comments were that software subcontracting is more difficult than hardware subcontracting because the requirements tend to be more variable, management of software development is more of an art, and estimation of the duration and size of hardware jobs is more accurate.

XI. CONCLUSIONS

What major lessons may be drawn from the SAFEGUARD software experience by prospective managers of other software projects?

(*i*) There appears to be great virtue in maintaining a flexible, informal, participative style of management.

(*ii*) Selecting good people and matching them to the right functions and problems are probably the most important management functions.

(*iii*) Informal, oral approaches to reporting status seem to work the best; written reports should be kept to a minimum. When used, they should be strictly constrained by length, time deadlines, and very hard-headed analysis of their purposes.

(*iv*) Defining and tracking concrete events based on functional capabilities, as exemplified in the Principal Events Reporting System, was found particularly useful by higher-level managers.

(*v*) Informal status reporting should be balanced with carefully prepared and written requirements and test plans and good configuration control of both requirements and code.

(*vi*) Using project management specialists as staff assistants for managers was found to be a very productive technique.

XII. REFERENCE

1. D. W. Meseke, "SAFEGUARD Data-Processing System: The Data-Processing System Performance Requirements in Retrospect," B.S.T.J., this issue, pp. S29–S37.

Glossary

ABM	Antiballistic missile.
BMDC	*Ballistic Missile Defense Center*—SAFEGUARD operations center in Colorado.
CENTRAN	*Central Logic and Control Translator*—The SAFEGUARD project standard software language.
CLC	*Central Logic and Control*—The multiprocessor computer used to drive each SAFEGUARD data-processing system.
DPS	*Data-Processing System*—The CLC hardware, software, and peripheral devices.
DPSPRs	*Data-Processing System Performance Requirements*—Documents that specify the required performance to be provided by the SAFEGUARD system software.
ECU	*Exercise Control Unit*—Digital interface equipment between the CLC and radar analog hardware used to facilitate simulation of a threat environment.
IOC	*Input Output Controller*—Controls the transfer of data between the CLC and its peripherals.
M&DSS	*Maintenance and Diagnostic Subsystem*—Test equipment and software supporting digital equipment maintenance.
MDC	*Missile Direction Center*—The MSR site and its remote launch facilities.
MDP	*Maintenance and Diagnostic Subsystem Processor*—CDC 1700 computer supporting digital equipment maintenance.
Meck	*Meck Island*—Field test site; part of the Kwajalein Atoll.
MSR	*Missile Site Radar*—Part of the MDC site complex; the radar equipment for missile tracking and local surveillance.
PAR	*Perimeter Acquisition Radar*—Long-range surveillance and tracking radar.
PPS	*Policies, Procedures, and Standards*—Manual containing documents that state policy defining the management, documentation, design, implementation, and control of SAFEGUARD software.
SAFSCOM	*Army SAFEGUARD System Command*—The Army agency having responsibility for SAFEGUARD ABM development.
SDRS	*SAFEGUARD Data Reduction System.*
SNX	*SAFEGUARD NIKE-X*—CLC assembly language.
SPARTAN	The long-range interceptor missile employed by the SAFEGUARD system.
SPRINT	A fast-reacting, short-range interceptor missile employed by the SAFEGUARD system.
STACS	*SAFEGUARD Tactical Computer Simulator*—Used for unit/task level debugging of programs.
STAG	*SAFEGUARD Threat Action Generator*—A software facility that enables the simulation of a SAFEGUARD threat for use by the system exerciser.
TSCS	*Tactical Software Control Site*—A collection of SAFEGUARD hardware that provides a duplicate of the software environment at a deployed tactical site.
TR/CR	*Trouble Report/Correction Report*—Part of a control system in which all problems were identified by a trouble report and the solution to each problem was described by a correction report.
XPF	*Execution Preparation Facility*—Performs the linkage editor function for software to be executed on the CLC.

List of Contributors

Newell H. Brown, B.S.E.E., 1952, University of Maryland; Bell Laboratories, 1952—. Mr. Brown has conducted sensitivity analyses of analog computer systems used in U. S. Navy radars. He assisted in the development of a millimeter radar contributing in the areas of antennas and systems. Since 1957, he has been concerned with problems of battle management, target classification, and systems engineering for ABM systems. He currently heads a department responsible for algorithm and program design for radar and missile control, performance evaluation of a multiprocessor computer system, and systems engineering for the SAFEGUARD Missile Direction Center. He holds patents on antenna systems. Member, ORSA.

Ronald R. Conners, B.S. (Physics), 1964, St. Louis University; M.S. (Electrical Engineering), 1965, University of California at Berkeley; Bell Laboratories, 1964–1974; American Telephone and Telegraph Co., 1974—. From 1964 to 1969, Mr. Conners was involved in the programming, integration, and testing of TSPS No. 1. From 1969 to 1974 he supervised various activities within SAFEGUARD, including development of the CENTRAN compiler, the acquisition of support computers, and monitoring the operation of these computers. He is presently a data systems supervisor at AT&T.

Edward J. Davis, B.S. (Physics-Mathematics) and M.B.A. (Management), 1964, Fairleigh Dickinson University. From 1958–1968, Mr. Davis was a manager, vice-president, and director of companies concerned with data-processing. Since 1969, Mr. Davis has been on the staff of Data Communications, Inc. on loan to Bell Laboratories, where he is a supervisor in the SAFEGUARD Support System Department. In this position, he has been involved in a variety of assignments in the development of large software programs. He is also presently Adjunct Professor of Management in the Graduate School of Business, Fairleigh Dickinson University.

Bernard N. Dickman, B.A. (Mathematics), 1963, Reed College; M.S. (Mathematics), 1966, New York University; Bell Laboratories, 1966—. Mr. Dickman initially worked on the development of a high-level language for No. 1 ESS. He participated in the design and implementation of the tracing facilities for SNOBOL 4. While working on the SAFEGUARD project, he was responsible for the design and coordination of implementation of CENTRAN. Later he worked on the SAFEGUARD hardware logic fault simulator and on an IR&D study. During 1974, Mr. Dickman was involved in the design and development of a UNIX-based program-development environment and is presently responsible for the design and implementation of a network specification language for BISCOM.

Bartholomew P. Donohue III, B.S. (Mathematics), 1960, Monmouth College; M.A. (Mathematics), 1962, University of Maine; Bell Laboratories, 1962—. Mr. Donohue has worked on several ABM projects, including software development for various parts of SAFEGUARD. Currently, he is the head of the System Operations Planning Department. Member, SIAM, Phi Kappa Phi.

Wilfred S. Doyle, B.S. (Physics-Mathematics), 1949, Northeastern University; Western Electric Co., 1955–1963; Bell Laboratories, 1963—. Mr. Doyle has worked on the development of software for military systems since 1963. He is currently analyzing real-time multiprocessor timing problems for SAFEGUARD.

Michael P. Fabisch, B.A., 1958, Columbia College; B.S.E.E., 1959, and M.S.E.E., 1960, Columbia School of Engineering; Bell Laboratories, 1960—. Mr. Fabisch worked on the development of the No. 1 Electronic Switching System and assisted in the installation of two early ESS offices. Currently, he is head of the Support Software Department and is involved in the development of SAFEGUARD.

R. Don Freeman, B.S., 1962, Michigan State University; Ph.D., 1965, Massachusetts Institute of Technology; Bell Laboratories, 1965—. Mr. Freeman initially did research on computer software, during which time he developed a technique for the automated typesetting of mathematical expressions. In 1968, Mr. Freeman transferred to the SAFEGUARD project, first working on testing software used on Meck Island. He supervised the sensor control software group from 1971 to 1973. Mr. Freeman currently supervises a group responsible for software that inventories and assigns outside plant facilities as part of the BISCUS/FACS project in the Business Information Systems program.

Lawrence J. Gawron, B.S.I.E., 1968, and M.S. (Computer Science), 1969, Pennsylvania State University; Bell Laboratories, 1969—. Mr. Gawron has worked on the design, development, and system analysis of SAFEGUARD's CLC multiprocessor operating system. Member, Phi Kappa Phi, Tau Beta Pi.

Joseph R. Gibbons, B.A. (Mathematics), 1965, King's College; M.S. (Computer Science-Mathematics), 1969, Stevens Institute of Technology; Bell Laboratories, 1966—. Mr. Gibbons has worked on a tactical environment simulation system and on missile subsystem test programs. Since 1969, Mr. Gibbons has been involved in the development and process design of a system exerciser.

Joseph P. Haggerty, B.E.E.E., 1967, Manhattan College; S.M. and E.E., 1969, Massachusetts Institute of Technology; Bell Laboratories, 1969—. Mr. Haggerty initially worked on the SAFEGUARD linkage editor, XPF. He later designed verification tests for the CLC operating system and analyzed the effect of proposed changes to the operating system. Member, ACM, IEEE, Eta Kappa Nu, Tau Beta Pi, Sigma Xi.

James R. Hahn, Jr., B.S.E.E., 1961, University of Miami, Florida; M.S.E.E., 1963, North Carolina State; Bell Laboratories, 1961—. Mr. Hahn's initial assignments were in thin-film memory design and logic design. He then supervised the design of the Maintenance and Diagnostic Subsystem for SAFEGUARD. Next, he was responsible for producing fault dictionaries using the SAFEGUARD logic simulation facility. He presently supervises a group involved in final system planning and evaluation of the total on-line maintenance aspects of SAFEGUARD.

Erna S. Hoover, B.A., 1948, Wellesley College; Ph.D., 1951, Yale University; Bell Laboratories, 1954—. Mrs. Hoover has worked on the design of the No. 1 Electronic Switching System and on the design of data base management systems. Currently, she supervises a group responsible for the surveillance and track software for the Missile Direction Center.

Harold M. Jackson II, B.S.E.E., 1949, Duke University; M.S. (E.E.), 1955, Ohio State University; Western Electric Co., 1949–1952; Project Officer, U. S. Air Force, Wright Air Development Center, 1952–1954; Bell Laboratories, 1955—. At Western Electric, Mr. Jackson designed test equipment for airborne analog bombing and navigation computers and participated in systems applications studies of magnetrons. His initial assignments at Bell Laboratories involved design and test activities related to transmitters and receivers for the NIKE-HERCULES radars. He designed portions of the precision tracker for TELSTAR. His assignments in the digital computer field have included design and test of special-purpose and general-purpose machines, specification of software requirements, implementation and testing of real-time operating systems and real-time missile guidance software, and establishing projectwide software change management, quality assurance, documentation standards, and management reporting systems for use on SAFEGUARD. He currently heads a department which does the design of tests involving the SAFEGUARD R&D Missile Site Radar system at Meck Island in support of the Army's ballistic missile test program. Member, IEEE, Phi Beta Kappa, Tau Beta Pi, Sigma Xi, Eta Kappa Nu, and Pi Mu Epsilon.

Robert A. Jacoby, B.S. (Physics-Mathematics), 1961, St. Thomas College; Univac, 1969—. As a Resident Visitor at Bell Laboratories, Mr. Jacoby was initially concerned with the design of real-time executive control programs for the NIKE-ZEUS and SPRINT missile test programs. He later worked on the design and development of an ABM system simulation, with specific interest in the user interfaces and data display. Since 1970, he has been involved in the design and development of the SAFEGUARD Data Reduction System. He is a Univac supervisor.

John P. Kuoni, B.A. (Mathematics), 1965, and M.S. (Statistics), 1967, Oregon State University; Western Electric, 1967—. As a Resident Visitor at Bell Laboratories, Mr. Kuoni initially worked on developing the linkage editor software for both the Meck test system and SAFEGUARD. He later completed performance improvement and cost control activities for the project computation centers. He is now assigned to the data base management technology department at Western Electric headquarters.

W. H. Mac Williams, B.S.E.E., 1936, Dr. Eng. (Electrical Engineering), 1941, Johns Hopkins University; Bell Laboratories, 1946—. As a Navy officer in World War II, Mr. Mac Williams was active in research and development of shipboard antiaircraft fire control equipment including radar, directors, and computers. At Bell Laboratories, he has worked in air defense and communications systems research and development, making use of both analog and digital computers, and in operational studies of the Bell System. At present, he is head of the Data Processing System Control Department. His current responsibilities include studies to improve software quality and fundamental studies of software reliability.

John F. McDonald, B.S.E.E., 1961, Kansas University; M.S.E.E., 1963, New York University; Bell Laboratories, 1961—. Mr. McDonald was involved in various phases of ABM system work from 1961 to 1973. He participated in generating the initial system requirements for the Meck test program and was involved in process design activities for the Meck software. He participated in the initial process design of the Perimeter Acquisition Radar software. Later he supervised the system design and system testing of the PAR system exerciser. Currently, he supervises a group involved in the systems engineering of new Business Information Systems applications and enhancements to existing BIS products.

Donald W. Meseke, B.S.E.E., 1958, Kansas State University; M.E.E., 1960, New York University; Bell Laboratories, 1958—. Mr. Meseke initially worked on display subsystems for the NIKE-ZEUS ABM system and later participated in test planning and data analysis for the Kwajalein test program for NIKE-ZEUS. Since 1967, he has worked on system design for SAFEGUARD and its predecessor, SENTINEL; he took part in writing the SAFEGUARD Data Processing System Performance Requirements. He is presently evaluating SAFEGUARD software design.

John D. Musa, B.A., 1954, and M.S., 1955, Dartmouth College; Bell Laboratories, 1958—. Mr Musa's assignments in the area of ballistic missile defense have included systems engineering, computer program design, human factors engineering, and supervision of computer simulation facilities. He was responsible for the creation and operation of a computerized project-management reporting system for SAFEGUARD and has actively been involved in the investigation of new program development techniques. He currently supervises a group that is designing display-and-control simulation programs for use in human factors testing. Member, IEEE.

Barbara C. Nichols, B.A. (Chemistry), 1966, Duke University; IBM Corporation, 1966—. As a Resident Visitor, Ms. Nichols was the manager of an IBM group developing verification tests for the CLC operating system. In 1970, with the assistance of Dr. Harlan Mills, this group became a pilot project for the phased introduction of structured programming techniques. Ms. Nichols is currently a member of the technical staff of the IBM Federal Systems Center where she is studying new techniques of software development.

John W. Olson, B.S.E.E., 1957, Michigan Technological University; M.E.E., 1959, New York University; Bell Laboratories, 1957—. Mr. Olson worked on the design of special-purpose digital processors for the NIKE-ZEUS project. He later supervised groups responsible for developing multiprocessor computers for NIKE-X and SAFEGUARD. Currently, he supervises a group responsible for telephone loop carrier systems. Member, IEEE, Eta Kappa Nu, Phi Kappa Phi.

John E. Petersen, B.S. (Marine Engineering), U. S. Merchant Marine Academy; Western Electric, 1941–1959; Bell Laboratories, 1959—. Since joining Bell Laboratories, Mr. Petersen has been primarily engaged in R&D contracting for the major subsystems of the SAFEGUARD project. He is currently head of the Whippany contracting department.

Alexander K. Phillips, B.S. (Aeronautical Engineering and History), Massachusetts Institute of Technology, 1969; IBM, 1969—. Since 1970, Mr. Phillips has been involved with SAFEGUARD operating-system development.

Charles J. Rifenberg, B.S. (Mathematics), 1963, St. Peter's College; M.S. (Computer Science), 1969, Stevens Institute of Technology; Bell Laboratories, 1963—. Mr. Rifenberg has worked on the design and development of basic support software. Since 1971 he has supervised several groups within SAFEGUARD, being responsible for support-software testing, computer-center support, developing the SAFEGUARD digital logic simulator, and developing a simulator for offensive attacks. Currently, he is supervisor of the COSMOS order-processing design group.

Francis E. Slojkowski, B.S.E.E., 1959, St. Louis University; M.E.E., 1961, New York University; Bell Laboratories, 1959—. Mr. Slojkowski has worked on a variety of military hardware and software development projects, including circuit analysis and design for the NIKE-HERCULES system, analysis and program design for NIKE-ZEUS, and management of prototype data-processing installations for NIKE-X and SAFEGUARD. He currently supervises a group responsible for the design of SAFEGUARD maintenance software.

Daniel Van Haften, B.S. and M.S. (Mathematics), 1970, Michigan State University; Bell Laboratories, 1970—. Mr. Van Haften has worked on the development and implementation of software change control for SAFEGUARD. Currently, he is engaged in development work on the Status Accounting System, an information storage and retrieval system for tracking problems in SAFEGUARD software. Member, Phi Beta Kappa, Phi Kappa Phi, Pi Mu Epsilon.

Patricia A. Van Sciver, B.A. (Mathematics), College of William and Mary; IBM, 1969—. Before joining IBM, Mrs. Van Sciver taught mathematics at the secondary and college levels. As a Resident Visitor at Bell Laboratories from 1969 to 1974, she worked on SAFEGUARD support software development test and verification.

F. Nelson Woomer, Jr., Electronic Technology, 1952, RCA Institutes; Bell Laboratories, 1952—. Mr. Woomer has been involved in the development of the M33, NIKE-AJAX, and NIKE-HERCULES fire control systems and the Missile Site Data Processing Systems at Meck Island, Kwajelein Atoll, and the Marshall Islands. On SAFEGUARD he worked on the development and operation of the Management Reporting System and on methods to predict scheduling problems during program integration. Currently, he is working in the Missile Direction Center system integration group.

Bell System