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			√	
Unit debugging aids on CLC	DEBUG	√	√	
Program patching		√	√	√
Time suspension			√	
Real-time simulation			√	
Low-perturbation aids	DARTS		√	
Manual input simulation			√	
Error-interrupt simulation			√	
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.