Porting DPPX from the IBM 8100 to the IBM ES/9370: Installation and testing

by G. E. Boehm
A. M. Palmiotti
D. P. Zingaretti

This paper describes the software tools, testing activities, and testing methods that were used to port the DPPX/SP operating system from its original implementation on the IBM 8100 Information System to its new implementation on the IBM ES/9370 Information System.

The porting of the Distributed Processing Programming Executive System Product (DPPX/SP) operating system, which was originally designed to run on the IBM 8100 Information System, to the DPPX/370 operating system which runs on IBM ES/9370 hardware, moved an operating system from one hardware architecture to another.

To accomplish the port, four basic steps had to be completed:

1. A new compiler, the PL/DS2 compiler, had to be developed to generate System/370 Assembler instructions from PL/DS (Programming Language for Distributed Systems) source code.
2. The machine-dependent components of the DPPX operating system had to be redesigned and rewritten to accommodate the new hardware.
3. The machine-independent components of DPPX had to be recompiled with the new compiler.
4. The new machine-dependent components and the recompiled machine-independent components had to be installed and tested on the ES/9370 hardware.

These four steps took approximately 30 months to complete, with the most time spent on installing and testing the new and recompiled components.

Much attention was given to the testing effort of porting DPPX because of the unique problem that testing had to address: How to take the redesigned machine-dependent components of DPPX, mix them with the recompiled components, install them and test them on ES/9370 hardware, and assure that DPPX/370 running on an ES/9370 would work with a quality rating equal to, or better than, DPPX/SP running on an 8100.

The solution was to divide the project into stages, start at the bottom, work to the top, and test it along the way.

Staging the installation and testing of DPPX/370

The layered architecture of DPPX, which is described in Reference 1, lent itself to the installation and testing of the ported code in a "staged approach." The overall strategy of the staged approach was to divide the system into its basic components, identify the dependencies that existed between them, and implement them in sequence from the bottom up in a series of seven stages. The early stages contained the low-level supervisor function that was needed to support the higher-level functions that were to come in later stages. Figure 1 illustrates this staged approach.

A *stage* was the vehicle that was used to build, plan, test, and manage the code being ported and the resources needed to do it. Each stage was assigned to a manager who was responsible for developing and executing a stage plan. The stage plan showed the major system functions to be implemented and their duration. The plan also showed, at a high level, the logical sequence in which the various system functions would have to become available so that the system could be built from the bottom up.

Each spin had its own characteristics.

Beginning with the first stage, planning sessions were held to resolve several interrelated concerns. The detailed functional content of a stage was evaluated, with consideration given to dependencies between functions of that stage. Module sizings (that is, lines of code) and functional verification testing plans were reviewed with an eye on project schedules. Current assumptions were examined for conflicts with the latest project plans, and any other outstanding questions applicable to the stage were resolved as necessary. In addition, suggestions for quality and productivity improvements were considered.

With the introduction of the critical supervisor and I/O functions, there were a lot of dependencies between functions. As a result, the development of stages 1 through 3 occurred in a fairly sequential manner. Each function being introduced into a stage had one or more dependencies on other functions preceding it in the same stage. Thus there was little parallel development. Any slippage in introducing a critical function would directly impact subsequent functions coming in.

At the end of stage 3 a very important checkpoint occurred: The system successfully IPLed from the system residence direct access storage device (SYSRES DASD) and supported a user logging on and accessing a DASD data set. With this level of function now available for use, new function was more easily introduced in stages 4 through 7. There were far fewer dependencies between the new functions coming in, which allowed more parallel development and easier testing to occur.

But even though installation and testing was viewed and tracked as occurring in discrete stages, it actually occurred as a "continuous integration" of function.

DPPX/370 spins. To introduce multiple functions in a stage, stages were divided into *spins*.

A DPPX/370 spin combined DPPX/370 software components to make up the DPPX/370 operating system. Spins, generated about every two weeks by the build department, contained different levels of function. The final product, DPPX/370, was not fully realized until spin 40; however, the birth of DPPX/370 began with spin 1, which contained the foundation for the later spins.

Figure 2 shows the diversity of software components contained in a spin. Each spin had its own characteristics. Early spins were weighted with many individual modules and scaffolding to provide primitive function. Later spins were weighted with full-function components and applications. There were also spins that contained only fixes to previous spins, with no new function being added. Figure 3 shows the relationship between spins, stages, time, and function.

Library structure. To introduce multiple modules, components, and applications in a spin, code had to be segregated into different library levels.

The Project Development Library (PDL) software development tool was used to manage the building of multiple modules into spins for testing. PDL allows multiple versions of the operating system parts to be stored and accessed by other developers.

Developers who were ready to make their code available to other developers promoted it to their depart-

Figure 1 The staged approach

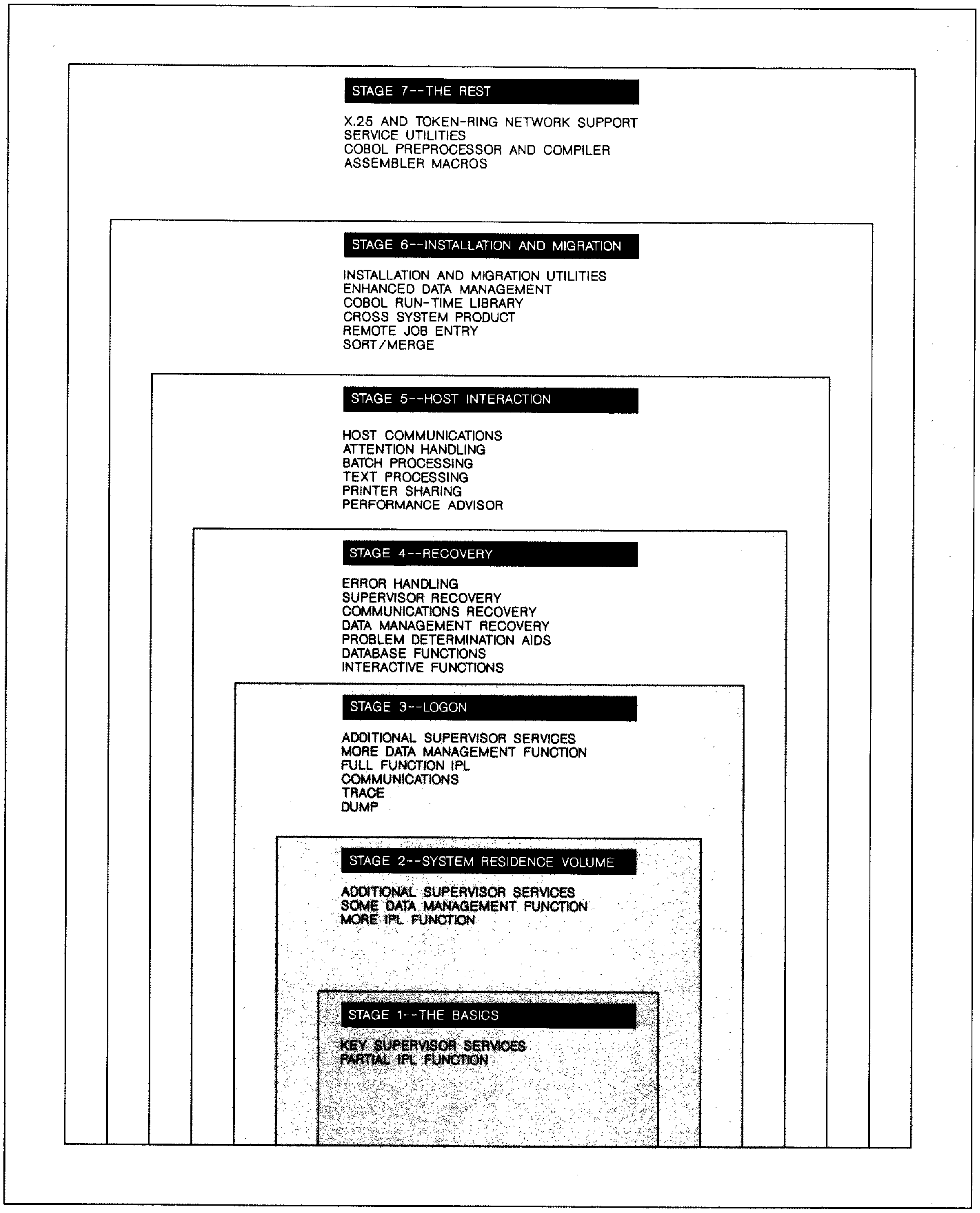
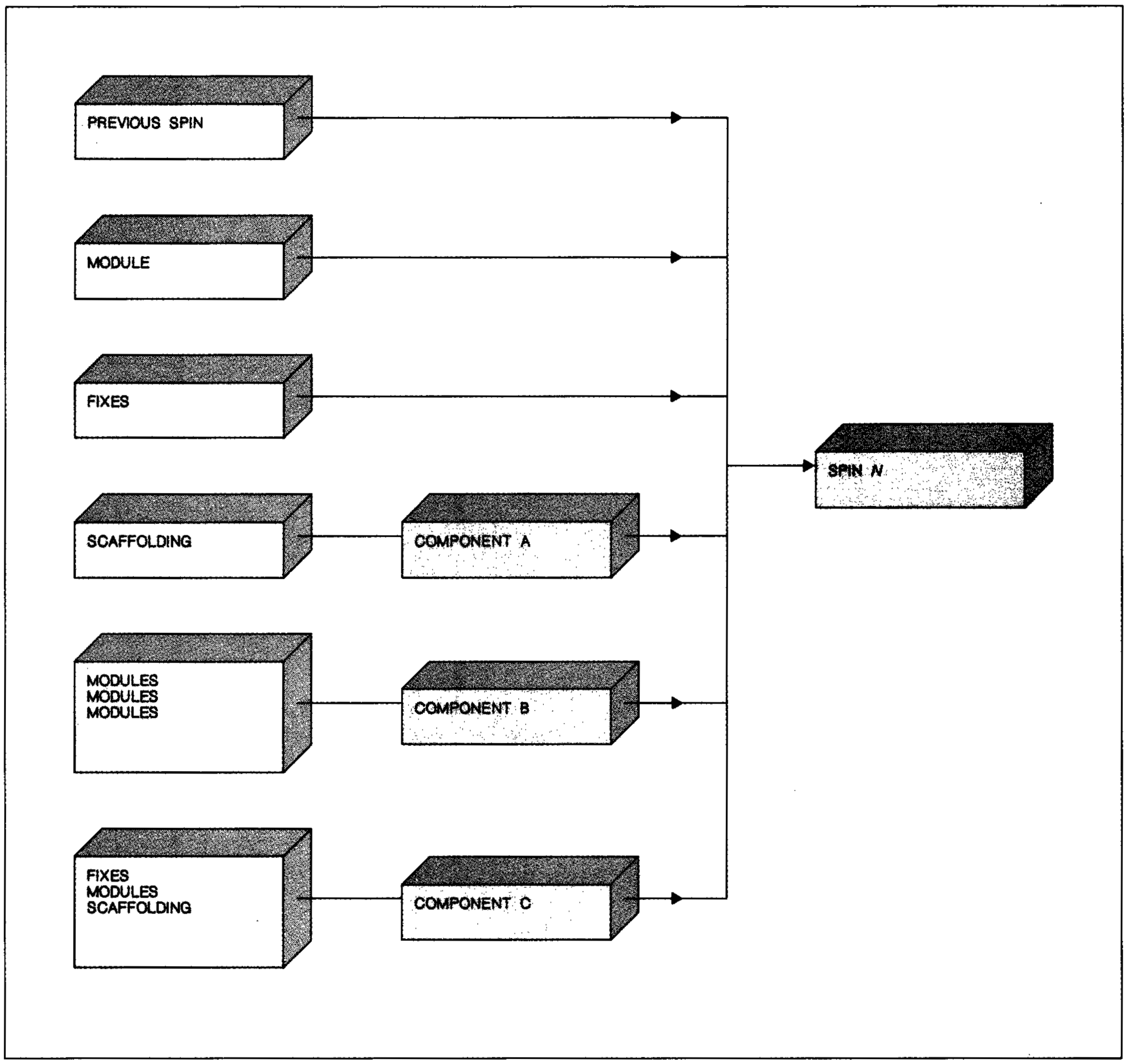


Figure 2 A spin

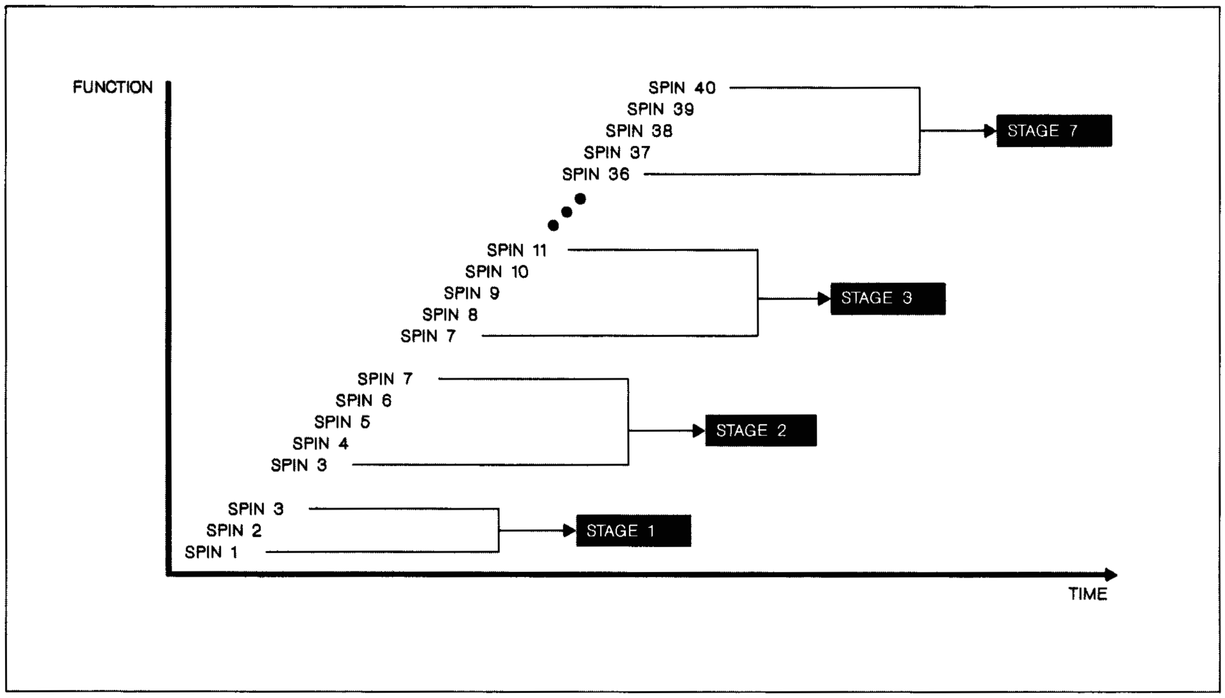


ment library. This made common subroutines, modules, and control blocks that must be linked into one load module, and messages that must be built into message data sets, accessible by everyone in the department.

The highest level of the library contained the oldest and most stable version of the code which formed the base of the operating system. Lower levels contained changes to the modules in the base version and new code that was added for later spins. To create a new spin, the old levels were promoted to the next higher library level, with the highest library level containing a collection of well-tested spins.

By keeping several library levels with different versions of code, developers chose the appropriate level for their use. Thus, developers who needed a stable version picked a higher library level to begin their access. A developer who needed the latest version of

Figure 3 Spins and stages



common code picked a lower level that was less stable but contained the latest changes.

Figure 4 shows the structure of the development library. The arrows show the direction code was promoted. As code was successfully tested, it was promoted to the more stable library levels.

Parallel libraries. To develop and implement in parallel, parallel library structures were needed to segregate the function.

In the early stages, because the low-level hardware-related components were being developed, all the testing was done by simulating the System/370 hardware on a VM system. However, separate VM systems were used for developing the code and testing the code. This ensured that testing would not affect the development environment and provided more flexibility in test system configuration.

To provide an efficient means to test the various spins built within a stage, executable object code from the development libraries was automatically copied to the PDL libraries which were set up on the test system.

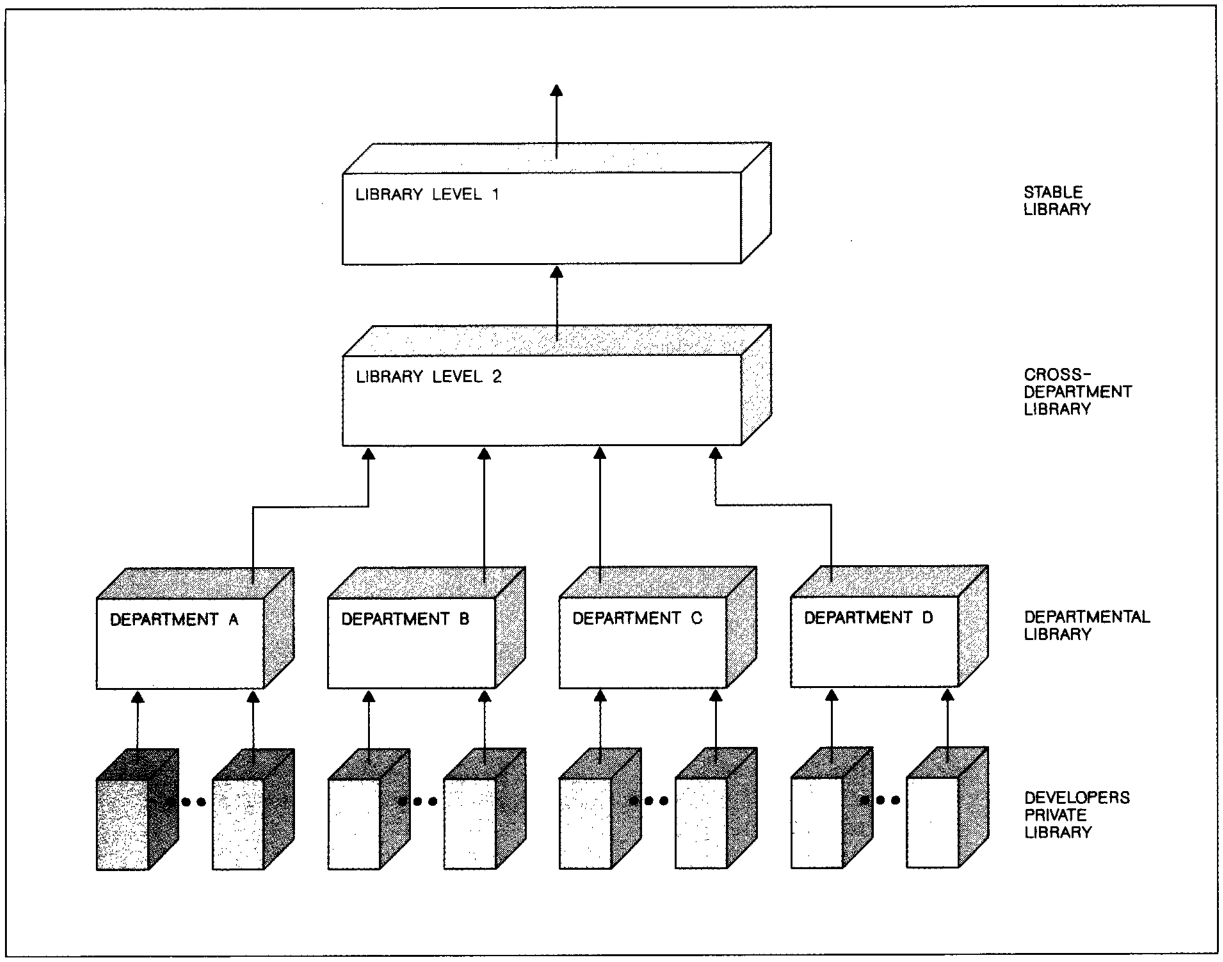
The availability of parallel libraries ensured that testing could continue on a stable base at the same time integration testing was being done on new functions. VM-based testing tools allowed the developers to select the desired level of operating system executable code to use, and performed a simulated IPL. The testers chose the appropriate level based on the function they were dependent on and the level's stability.

Once the system became stable and could be IPLed, the parallel libraries were no longer needed. Build tools were used to load the system onto test disks, the developers added their code to be tested, and IPL was executed through VM. The development libraries, however, continued to provide controlled sharing of the operating system code.

Functional verification

Functional verification testing is part of the software development process used by the DPPX organization to provide quality code to its customers. Traditionally, functional verification (FV) is a test, or group of tests, whose purpose is to verify that the functional operation of a module, from an internal perspective,

Figure 4 Development library structure



is correct. An FV test is performed by exercising every functional variation of a module with valid and invalid data to ensure that the data are processed correctly.

FV testing as performed by DPPX development includes test activities that are frequently thought of as unit test, FV test, and component test. Thus, FV testing is applied not only to individual modules but to groups of modules that are combined to provide services or functions (such as command processors) and whose external characteristics must be verified. Functional verification is performed by the programmers who own the modules being ported during a stage.

During the early stages, FV testing helped verify the correct operation of the new development compiler. The testing ensured that recompiled modules performed the same function on an ES/9370 that they had on an 8100. In general, however, modules that were only to be recompiled were not scheduled for extensive FV testing. The bulk of FV testing was performed on new modules that provided new function on the ES/9370, and on rewritten modules which were changed to accommodate the new hardware.

Functional verification of DPPX/370 provided for the detection of several classes of software errors (bugs) which follow.

Compiler bugs and user error bugs. FV testing of DPPX/370 provided the first "live" test of the new compiler, which was developed in parallel with DPPX/370. It would come as no surprise, then, to see some compiler errors surface. But compiler errors were by far the hardest bugs to detect, analyze, and correct. This difficulty was due, in part, to the lack of experience the DPPX organization had with the differences between 8100 and ES/9370 architectures, with System/370 assembly instructions, and with recognizing compiler bugs.

Compiler bugs were very important because of the far-reaching effect they could have. Once it was recognized that compiler bugs were being encountered, the first burst of debugging energy was put on determining the "type" of bug encountered. If it appeared to be a compiler bug, the bug was transferred to a "compiler debug team," that would either identify the bug as a compiler bug or a "user error bug." A user error bug is a bug that was generated because of the architectural differences between the 8100 and the ES/9370. They were usually fixed by either declaring some variable with different attributes or recoding some logic to accomplish the same task with different instructions. When true compiler bugs were discovered, the entire library of tested code had to be recompiled with the fixed compiler and retested.

During the early stages of testing DPPX/370, tool bugs were almost as difficult as compiler bugs to detect and analyze.

Compiler bugs were most prevalent during the first three stages of the porting project; after stage 3, they were rarely encountered.

Software tool bugs. During the early stages of testing DPPX/370, tool bugs were almost as difficult as compiler bugs to detect and analyze. The reason for the difficulty was because of the close coupling between the tools to build, load, and execute the DPPX/370 code and the code itself. Also, a mix of skills was required to analyze tool problems. Those familiar with the tools had minimum knowledge of the internals of DPPX/370 and those familiar with DPPX/370 had minimum knowledge of the internals of the tools. As the programmers became more familiar with the tools and the test bed, tool bugs were more easily detected and fixed.

Scaffolding bugs and DPPX/370 code bugs. Scaffolding and DPPX/370 code bugs, in contrast to compiler and tool bugs, were more readily identifiable with a specific function or area of DPPX/370 code, making them the easiest to detect, analyze, and fix. The majority of bugs fell into this class.

FV test bed. To support functional verification, simulated System/370 hardware was provided by the virtual machine (VM) operating system, which executed DPPX/370 code as a guest (that is, second-level) operating system. Special considerations had to be made to accommodate this testing environment since DPPX/370 is not designed to function as a guest operating system.

VM was used as the FV test bed for the following reasons:

VM was available before the real hardware. The porting effort began eight months before the ES/9370 hardware was available for testing DPPX/370 code. Delaying testing until ES/9370 hardware was available would have delayed the availability of the operating system.

VM capacity was available in larger quantities than the real hardware. For the duration of the project, only seven ES/9370 computers were available to 85 development programmers to port over a million lines of code.

In the early stages, before sufficient DPPX/370 function was available to provide a test bed on the real hardware, 15 multisession terminals were available to the programmers and testers of a stage to test their code. Multisession support (one terminal supporting four different sessions at the same time) was required because of the software tools used to test the ported code on VM.

In the later stages, multisession support was provided via RLSS and VM/Virtual Telecommunications Access Method (VTAM), which allowed the DPPX programmers to test their code from their office terminals.

VM provided the capability to do parallel testing of DPPX/370 function. The main aspect of a stage was that it identified the primitive or low-level DPPX/370 function that had to be available to support the higher-level functions and applications. But although a dependency existed between the low- and high-level function, both levels could be tested at the same time, in parallel. Parallel testing was accomplished by using scaffolded code to simulate function, shared VM minidisks to allow multiple system versions to exist simultaneously, and special software tools, developed by the DPPX organization.

Several software tools played an important role in the porting of the DPPX operating system. Build tools were used to: build, load, and execute primitive versions of DPPX for testing before sufficient support was available for a normal IPL; create DPPX/370 system resident (SYSRES) volumes from specification files; and place SYSRES images onto IPLable fixed block architecture disks for testing.

Aside from the build tools, there was a software debugging tool, described in the next section, that became critical to the timely completion of FV testing.

DPPX/370 running on VM remained the primary FV test bed for most of the porting effort. However, all code was eventually tested on the ES/9370 hardware during independent component and system tests.

Common verification tool. When reviewing the debugging requirements for DPPX/370, several issues had to be considered. Foremost was the nature of the operating system itself. DPPX/370 is designed to run on a machine with the architecture of a System/370, and a debugging tool was needed that would make accessible all the features of a System/370-style operating system: general-purpose registers, control registers, real and virtual addressing, condition codes, and program status words (PSWs). Since DPPX/370 is an operating system, the debugging tool would require different characteristics from typical single-program debuggers.

Several approaches to debugging were considered, including the use of debugging facilities on the target hardware itself. This was not satisfactory because of the lack of test hardware as explained earlier. Another consideration was to build a test version of the DPPX/370 system so it could be IPLed as a guest operating system running on VM/SP. With this approach, any debugging facilities of CP (VM/SP's control program) would be available. But another drawback emerged: CP's debugging facilities are too low level to meet our development productivity objectives.

To address productivity, another debugging tool, Source Level Debug (SLD), was considered. It allowed one to debug programs at the source, or program code, level. However, there were more drawbacks since SLD fell short in three significant areas:

- It supported the debugging of individual CMS programs, but DPPX/370 is an operating system that does not run under CMS.
- It did not support the programming language that was being used for DPPX/370.
- It provided little support to debug at an assembler code level.

These shortcomings were unfortunate since the other features provided by SLD would increase debugging productivity considerably.

In the end, it was decided to combine some functions of SLD with new debugging functions written by our own tools department. This would provide, among other things, the necessary support for debugging non-CMS programs (our most critical requirement) and the ability to debug assembler code. It would also provide the ability to test from one's own office terminal, eliminating any dependence on actual ES/9370 hardware in the early stages of testing.

This special implementation of SLD was the tool called Common Verification Tool (CVT), an in-house tool not shipped with the DPPX/370 licensed program.

CVT provided debugging capabilities which were rich in function and easy to use. The primary functions provided by CVT include:

- Pausing at specific program locations through the use of breakpoints
- Interrupting program execution or wait states
- Displaying or altering the contents of storage, registers, and the PSW
- Stepping through machine instructions
- Logging the debug session and scrolling the session listing
- Repeating command sequences

In addition to these functions, a certain amount of additional debug capability is available by using any of several CP commands. These CP commands can be used simultaneously with CVT, thus giving the tester greater capability for addressing a given problem.

The structure of CVT requires an interface to Virtual Machine Communication Facility (VMCF), a component of the VM/SP operating system which permits information exchange between VM userIDs. In addition, code had to be added to three areas in the DPPX/370 supervisor to accommodate the needs of CVT for initialization, program and external interrupt handling, and the program contents of system storage.

Independent component tests

Philosophy. Independent component test (ICT) tests all the components in a system from an external perspective. ICT looks at each component in the system and tries to use it or break it as a customer would. *Independent* means that the people planning and executing the tests are totally unrelated to the component being tested. This independence gives the benefit of simulating a customer environment. The tester really becomes a customer and must use customer-like documentation to learn and use the functions provided by a component.

ICT usually has two parts which begin after development has completed its FV tests: a regression test which verifies that the system has not regressed since the last release and a new function test which verifies that the new function added to the system works according to the documentation that will be provided to the customer.

Risks and concerns. ICT addressed the primary concerns, from a testing perspective, with porting code from the 8100 to the ES/9370:

1. Would the code generated by the new compiler perform the same function on the ES/9370 that it did on the 8100?
2. Would the performance of the system increase when going from 8100 architecture to ES/9370 architecture?
3. Could customers migrate their applications easily from 8100s to ES/9370s?
4. Would timing and stress-related problems show up because of the difference between the ES/9370 and 8100 architectures?

Types of tests. The ICT effort, like the development effort, was accomplished in seven stages and the function of the ICT group actually went beyond a typical independent component test. The ICT group was responsible for performing the following tests.

During stages 1 through 3, there was no terminal support in the system and ICT could not test the system like an external user. To avoid wasting time, and to achieve as much test coverage as possible, ICT carried out *stage validation* (SV) *tests*. The emphasis of the SV test was to test system support for customer applications as early as possible.

To avoid wasting time, and to achieve as much test coverage as possible, ICT carried out stage validation tests.

SV tests were performed by putting hooks and stops in system code and simulating multiple-user environments. System dumps were taken to verify correct operation.

Within each stage were multiple spins. *Spin validation tests* were simply a "bucket" or subset of test cases, that had successfully completed on previous spins. When a new spin was made available, this bucket was executed to ensure that the system had not regressed. After new tests were completed successfully, they were added to the test bucket.

As development progressed through the stages, compiler problems were discovered and fixed. The only way to really ensure that these fixes did not cause problems with previously compiled code was to totally recompile the code in the system to that point. Once the system was recompiled on the "fixed" compiler, the system was handed over to the ICT team to run *compiler regression* (CR) *tests.*

The CR test bucket was created by doing an analysis of the available components and determining the kind of coding techniques that were used to develop them. The most "compiler stressful" components were selected and a subset of the ICT test cases was re-executed with the "fixed" compiler. Once these tests completed successfully, the "fixed" compiler became the "only" compiler and the porting effort continued.

As the stages progressed, more and more components became available to ICT. After a component was "ICTed," it became part of a *multiple components in stress* (MCS) *test*. Buckets of automated tests were created by using many components together in the most stressful situation that could be created. These buckets were like a stressful regression test and were used extensively in stage 7 after all the code was in the system. MCS buckets were used as availability tests and many times ran over the weekends to ensure that the system was stable and could remain operational for extended periods of time. MCS tests will continue to be valuable regression tests for future releases.

Planning for ICT. The "test group" of three senior-level DPPX programmers was responsible for developing the basic, high-level ICT plan. Other experienced programmers were frequently consulted to review and discuss the preliminary high-level plans.

The first step in determining how to test DPPX/370 was to divide the entire system into logical areas of test. These areas were called *environments* and 15 emerged. It is not necessary to discuss all 15, but some examples follow:

- *COBOL* to verify the COBOL instruction set
- *Program prep* to verify the components that a customer would need to prepare and execute a program, such as the editor, interactive map definition (IMD), format management, various compilers, and the linkage editor
- *Problem determination* to verify trace facilities, dump facilities, error reporting, and summarization
- *Connectivity* to verify all device support that was announced, such as displays, printers, controllers, modems, and pass-through operating systems
- *Communications* to verify the ability of DPPX/370 to communicate with peer systems and host applications
- *Migration* to verify the commands, tools, and procedures that customers need to migrate their applications from DPPX/SP to DPPX/370
- *Performance* to verify that the performance of the system was improved over the 8100

The components to be tested in each environment were mapped against the stage when the component would be available, and then a schedule of ICT start times for each environment was created. Based on when the environment test could start and an estimate of how large the environment would be, a planning phase was projected to precede each environment test. During the planning phase, detailed test plans and test cases were written for execution in the test phase.

Because of tight schedules, as many redundancies as possible had to be removed from the test plan. Tests needed to be prioritized to distinguish those tests that needed to be executed from those that should be executed if time and resources allowed. A test approach review (TAR) meeting was held for each component in the system. Each TAR meeting was attended by at least one member of the test group, the lead developer for the component, and others who were familiar with the component. Many times the others were from the National Service Division (NSD) or management; they might be people who had previously left the area or anyone who might be considered an expert.

Preparation for the TAR meeting involved someone dividing the component into its functions, subfunctions, associated commands, command operands, possible error conditions, and PD tools. At the TAR meeting, the preparation was reviewed and discussed. The discussion involved:

- What are the risks with porting this component?
- How much of the code is "new" versus "recompiled?"
- Where were the problems in the past?
- What items were "implicitly" tested just by normal system execution?
- What items would be sufficiently tested by functional verification?
- What items needed explicit ICT tests?
- Who (what group) would write the explicit tests?
- What items would get no test at all because it was not deemed necessary?
- Of the items that needed explicit tests, were there any existing test cases, and, if so, where are they?
- What should ICT do to stress the component?
- Are there any available "regression" tests for the component?
- In what ICT environment should the various items be tested?
- How large is the ICT effort for this component?
- When will the component be available to ICT and how much of it will be available at that time?

From the TAR meeting came an understanding of what work was needed to be done, who would be responsible for the work, and when the work needed to be completed. The basic ICT test plan was created from these meetings.

Executing and controlling the test effort

As the TAR meetings progressed, a new library structure called Test Library (TL) was developed that could hold test cases, test plans, test programs, and the results of the TAR meetings. The TL also included automated tools, which could be used to create "test packages" for each of the 15 environments.

A test team and test team leader were assigned to each environment. During the planning phase, the team leaders had several responsibilities: find and merge existing test cases into the environment's test package; automate and modify existing test cases where necessary; create test cases that were needed but did not exist; coordinate the planning and testing of an environment.

The team leader also worked with the test coordinator to set up a tracking mechanism that kept management informed of the progress of the test effort.

When the execution phase began, the team leader coordinated the test effort by ensuring that any corrections made to the test cases or programs during execution were promoted back into the test package on the TL. The team leader also worked with the test coordinator to set up a tracking mechanism that kept management informed of the progress of the test effort. The test team's responsibility was to run all the test cases associated with an environment and ensure they completed successfully.

In addition to the test teams and test team leaders, the ICT group had one ICT coordinator. The test coordinator's responsibility was to understand each of the 15 test environments and their unique requirements. Some environments needed special hardware, others needed special skills. Some needed special tools and special configurations. It was the ICT coordinator who was the liaison among the different departments in the organization, ensuring that the special items were available when a test environment was to begin.

While being actively involved with the planning and execution of each of the environments, the test coordinator was also responsible for tracking and reporting the progress of the test effort. During the planning phase, the test coordinator would meet with the team leader of an environment to determine what areas are to be tested, who was to perform the tests, and how long it would take. This information was tracked graphically by stage and environment.

Measuring and evaluating the results. Problem tracking and analysis report (PTAR) status meetings were held frequently with management, and the PTARs were assessed in terms of their severity and impact on the testing effort. PTARs that were impeding test progress were given highest priority for being fixed. These meetings were very successful for informing management where the emphasis was needed to allow the test to continue smoothly.

After ICT was complete, an assessment of each component was made based on the number of problems found and the amount of code in the component. When assessment revealed a weak component, recommendations for extended testing were made to the system test group.

System test

The previous approach to DPPX system test. In the days of DPPX/SP, the goal of system test was to test the operating system by running in an environment similar to a customer environment. In effect, system test was the first customer. The objectives used to meet this goal were as follows:

1. Combine and test all components as a total system.
2. Test the system the way a customer would use it.
3. Put the entire system under stress.
4. Determine and approach system limits.

To meet these objectives customer systems were obtained that included object code, databases, and customer-developed command lists (CLISTs). These systems were used quite extensively by the system test group in conjunction with "in-house" applications that testers wrote to test areas not covered by the customer applications, as well as new functions. As new DPPX releases were developed, the customer applications were less effective for testing, since they only addressed old functions.

Since the system test group did not have the source code for these customer applications, they could not be modified to take advantage of the new functions in the system. Also if an error occurred in the application, it was virtually impossible to locate and fix the problem. This also caused problems with DPPX problem determination, since the testers were never sure exactly what the application was doing when the system error occurred. Many system test environments (STEs) had to be written to test specific functions and areas of the system. By the fourth release of DPPX/SP, system test had become very component-oriented rather than system-oriented.

When a failure was introduced, the system error log, operator log, and host NetView facility were checked to verify that accurate messages were logged.

The total systems approach. Today's method, the total systems approach, is broken up into three parts: problem determination (PD), contracted efforts, and end-user systems (EUS).

The problem determination part of system test had basically remained the same from 8100 testing to ES/9370. Tests were designed to introduce permanent and intermittent hardware and software errors to determine the capability of DPPX/370 to recognize, properly diagnose, and report the error. Hardware bug points were obtained from the Endicott ES/9370 engineers. These bug points were the physical address of pins on cards within the ES/9370 processor which, when grounded, would simulate an actual card failure. These failures included memory errors, adapter errors, and processor errors. When a failure was introduced, the system error log, operator log, and host NetView™ facility were checked to verify that accurate messages were logged. Effectiveness testing was included as part of the PD testing. In these test cases a person who was computer literate, but not DPPX literate, was asked to perform basic tasks on the computer. Certain errors would be introduced into the system as the subject was executing these tasks. The subject was timed to see how long it took to identify the exact cause of the problem. Only the messages in the operator and error logs, NetView, and the DPPX support manuals could be used for problem determination. A test was marked successful when a problem was identified, and in some cases resolved, within 30 minutes.

Certain isolated test efforts were contracted outside of IBM. Some communications tests were also run at IBM locations in Germany and Japan. The test plans were written by the actual testers and were reviewed and approved by system test members before the start of testing. A system test department member monitored all of these tests and reported at the weekly staff meetings. The test locations were selected based on their knowledge of the function, their interest (which stemmed from customer requirements), and their test bed facilities. For example, X.25 testing was performed in Germany, since our German customers have a strong requirement for X.25 communications support.

The third part was the key to the success of this system test effort. Current customer system environments were used to simulate, in our lab, the daily activities of the customer. Using these systems allowed the testing of migration, usability, and equivalence.

The EUS test was broken up into six phases: (1) obtain a customer system, (2) learn, combine, and promote, (3) migrate, (4) execute, (5) expand, and (6) test system under stress and perform unstructured tests.

The first two represented the preparation for system test, whereas the remaining four were actual testing phases.

Preparation. System test's goal and objectives were not changed with this new approach, but the methods for meeting the objectives were. In order to meet the objectives more effectively, it was necessary to find the best customer systems available to use in the test.

The general requirements of each customer were determined before approaching them for testing. The department members and management identified the following requirements.

DPPX application source code. It was crucial that the group receive the application source code for all parts of the application that were not written in Cross Systems Product. All COBOL had to be run through the preprocessor and compiler, and any other applications would have to be rewritten (for example, PL/DS and assembler applications).

CLISTs, panels, command facility extension (CFE) scripts, and user ID definitions. All parts of the system, written or modified by the customer, were necessary to run their application.

DPPX databases and transactions. Test data and the customer-defined transactions were crucial to executing the complete customer application.

Application documentation. We felt that any information the customer could supply us in the form of data-logic flow diagrams and manuals would help us learn how the application runs and how to run the application.

In order to duplicate the operating environment completely, it was important to have host source code that communicates with DPPX, host databases and transactions, and host application documentation. However, due to the complexity and hardware dependencies of the host applications, it was virtually impossible to obtain this material.

With the requirements known, it was decided that obtaining three customer systems would be sufficient for the system test effort.

Applications were selected from three different business environments (an auto parts inventory system, an insurance claims processing system, and a plant maintenance and control system) which, when combined, would use most of the components of the operating system. It was also a requirement that at least one customer application be written primarily in COBOL and at least one be written primarily in Cross System Product.

Once the three customers and IBM had come to an agreement concerning the terms and conditions of the project, contracts were written up and signed, and key people from the system test department visited the customer sites to learn the system and understand the running environment.

Once the customer application arrived it was immediately loaded on an 8100 to verify that all the parts were present. The department was divided into three teams, each consisting of one senior department member and one junior member. Each team was responsible for a customer system, and the senior members were jointly responsible for creating a fourth combined system which consisted of all three customer applications.

The team members spent approximately one month familiarizing themselves with their customer system. They documented procedures for starting and running the application. These documents, along with the documentation received from the customer, were used to create Teleprocessing Network Simulator (TPNS) scripts that would be used for multithread and stress testing. This familiarization period was also important for planning which parts of the system would be migrated to the ES/9370.

Test Library (TL) was used for the archiving of the customer application parts. The tool resided on a virtual memory (VM) system. The testers scanned the customer system and transferred the customer-specific parts to TL. All parts likely to cause a problem when merged into the combined system were documented.

An installation process was developed for each system to simplify the installation of a customer system from TL to an 8100. This installation process consisted of running CLISTs that created all of the databases, and set up environments and device definitions. Installation instructions were written to guide the tester, and any specific hardware and software requirements were identified.

The final phase of system test preparation was to preprocess and compile the COBOL applications, and perform any needed rewrite of other applications. The MVS COBOL II preprocessor and compiler were used in this phase. This proved to be the most tedious and time-consuming task. It was found that the old DPPX COBOL compiler was not as strict about adhering to the standard COBOL programming rules. In many cases programs that compiled without error with the DPPX compiler were not compiling with the MVS COBOL compiler. Work had to be done to modify the DPPX COBOL applications. Applications written in other languages such as DPPX Assembler and PL/DS also had to be rewritten in either COBOL, Cross System Product, or System/370 Assembler. In two cases the customer sent a team of their developers to aid in the rewriting of the applications. This proved mutually beneficial for obvious reasons. Once this work was completed, both the 8100 applications and the rewritten ES/9370 applications were transferred to TL for archiving. We were now ready to start system test.

System test activity. The first phase of system test was migration. Using *DPPX/370 Migration: Planning,*[2] each team determined the best method for migrating its system. Some modifications to the team's plans had to be made to verify that all three migration methods were used (Distributed Systems Executive, stand-alone DASD dump/restore, and Peer Data Transfer). The migration tool was loaded on the 8100, and the teams used *DPPX/370 Migration: Procedures*[3] to execute the migration process. Timings were taken throughout this migration phase to determine how much planning and actual migration time was needed for these large systems.

The rewritten COBOL II programs were compiled on MVS and transferred down to the ES/9370.

The rewritten COBOL II programs were compiled on MVS and transferred down to the ES/9370. System/370 Assembler code was assembled and transferred to the ES/9370 systems. A subset of the COBOL programs was also preprocessed and compiled on the DPPX/370 system.

Finally, a visual inspection of the system was performed to verify that no parts were missing, and the contents of the 8100 databases were compared with the corresponding ES/9370 databases to verify that no corruption had taken place during the migration. Three methods were used to compare the databases: (1) if the databases had a record length less than 133, they could be printed by DPPX and compared, (2) if they were longer than 132, they were transferred to VM and compared, or (3) if the customer application generated detailed reports from the database contents, the 8100 and ES/9370 reports were compared.

With the system now completely migrated, it was time to begin running the customer applications on the ES/9370. This phase was divided into three basic parts. First, the application was tested live by the test team. In some cases this uncovered run-time errors in the COBOL modules. For the most part, however, the live testing proved that the migration was extremely successful. Second, the TPNS script was run using a single TPNS user to verify the network path and perform some performance tests. The output of these tests was compared with that of the same run on the 8100 system. Third, the number of TPNS users was increased until the system was running at least 70 percent CPU utilization. At this time more performance tests were run and compared with 8100 tests.

Errors were introduced during TPNS runs to determine the effect of hardware and software (application software) errors on the DPPX/370 system while running the customer applications. These errors included disk and SDLC adapter errors, stopping TPNS before completion (to simulate a communication break outside the ES/9370), stopping and restarting environments, submitting looping applications to batch processing, and powering off and on live devices defined to the system.

Once the team was satisfied with how the application was running, the team entered the next phase of system test, to expand the system beyond its current capabilities. The team determined which functions, or components, were not currently being executed by the system. They then modified the system to incorporate these functions. Basically, these functions were those introduced in Release 4 of DPPX/SP and DPPX/370 that the customer was not implementing. All customer systems were at DPPX Release 3 level. One customer system was brought to a Release 4 level upon arrival.

The expanded system was then rerun, starting with the customer application, and starting the new components one at a time until the entire expanded system was running. The objective was to push the processor to approximately 90 percent utilization and continue under that stress for long periods of time without IPLing.

At the same time, the senior team members were combining the three customer applications on a single system to run a CPU and I/O intensive stress test for an extended length (at least 65 continuous hours). Where necessary, CLISTs and in-house developed applications were added to stress the system further. Due to the size and complexity of the new system, it was impossible to run an equivalent system on an 8100 and get performance comparisons; however, some performance tests were run. Also, time was spent tuning the combined system to maximize execution time and reduce overhead.

The final phase of system test was called the "unstructured test period." After all the test cases had been executed and all high-severity problems had either been fixed or a plan was in place to get them fixed, all department members converged on the combined system to rerun any tests they felt uncovered the most problems, or to run any new tests that they were not able to use during the earlier phases of system test. This has proved in the past to be quite beneficial, shaking out some smaller problems before the system goes out to the customer.

Once system test was completed, an exit review meeting was held by both system test and development managers to confirm that the test exit criteria had been met and all outstanding problems had a plan in place to be resolved.

Since the application source code and documentation was available, it was much easier to diagnose a problem and separate system problems from application errors. Once it was understood in detail how the applications functioned, the time needed to obtain the proper data for development to fix a system problem was reduced significantly. If a problem needed to be reproduced, the tester was more familiar with what was going on at the time of the error and was more likely to be able to reproduce the error and take less time to do so.

The total systems approach proved to be a great improvement over the old method of system test.

Benefits of the new approach

The total systems approach has proved to be beneficial in many ways.

Working with the customer created an atmosphere of technical exchange. Because of the information passed to the customer about the migration experiences with their system, the customer was able to start planning for migration earlier. In many cases the testers had to work with the customer's technical staff to resolve problems.

The department members gained a better understanding of how the customer actually uses the DPPX system. In general, the development and test groups are isolated from the customer; they really do not understand the degree of complexity of the customer applications and how these applications interact with the DPPX system. By gaining this understanding, they can better develop and test DPPX.

Prior to this test effort the system test department members had single areas of expertise where they would spend most of their time testing. Through this new method, the testers learned more about the overall system than would otherwise have been possible in the same amount of time.

More components were tested by fewer people earlier in the test cycle than was possible using the old method. System test concentrated on a complete system throughout the test period rather than testing individual components.

Fewer systems were needed to run this test, compared with the old method, and in general there was a constant high degree of machine utilization during the test period. The system was under stress early in the test cycle.

The combined customer system functioned as an excellent vehicle for regression testing and fix package testing. Since the combined customer system exercises almost every component in the DPPX/370 system, there is no need to spend weeks sifting through old test plans to create a regression test. It also stressed the system in a more realistic manner by having a large number of simulated users execute real customer applications.

Concerns with the new approach. There were two major concerns that arose from management about this new approach. There was a serious concern that the test group may inadvertently become concerned with the customer applications rather than with the system itself. Management felt that it was possible that the testers would spend more time trying to understand what the application was doing rather than how the system was functioning under the application. To avoid this, all system test plans were inspected by peers, management, and key developers. Also, weekly status of tests completed and planned for the following week was provided to management.

There was also a concern that error testing would not be performed to the degree it was done in the past. This concern was addressed by having the testers document in their test plans all errors they planned on introducing into the system. Thus coverage could easily be evaluated. The problem determination test, which introduced a variety of hardware and software errors, was also documented and inspected.

Directions. The total systems approach proved to be a great improvement over the old method of system test. Many problems that were not found in component testing were uncovered and a number of customer-application-dependent problems were found. These problems would not have been found under the old method of testing. DPPX/370 was exposed to a customer system environment from the start of system test. This provided us with a stable system earlier in the development cycle.

We are exploring other opportunities which will involve the customer even more in the development cycle, including involving the customer in more test efforts and possibly development and design efforts.

Summary

The development approach taken by the Distributed Systems Programming (DSP) organization was key to the successful port of DPPX/SP to DPPX/370 for many reasons. Programmer productivity was optimized by staging the development activities, by having a test bed before the actual hardware was available, and by providing the developers with tools to simplify debugging. A stable DPPX/370 system was attained very early in the development cycle by testing DPPX/370 at every stage rather than waiting for all development activity to complete. The component tests were executed by testers other than the component developers. This exposed documentation and usability problems, in addition to functional errors, which otherwise may not have been found. The use of customer application environments in system test permitted greater and more realistic test coverage and required fewer resources than normal.

The DSP organization is continually investigating new ways to improve its approach to systems development.

Acknowledgments

This paper, a summary of information documented in a series of internal technical reports, is due in no small part to the contributions of a number of our colleagues. Their work has provided a wealth of information from which to draw. We wish to acknowledge the work of Bob Abraham, Lou Fitzpatrick, John Forsythe, Garth Godfrey, and Brian Goodrich.

Cited references

1. R. Abraham and B. F. Goodrich, "Porting DPPX from the IBM 8100 to the IBM ES/9370: Feasibility and Overview," *IBM Systems Journal* **29**, No. 1, 90–105 (1990, this issue).
2. *DPPX/370 Migration: Planning*, GC23-0641, IBM Corporation; available through IBM branch offices.
3. *DPPX/370 Migration: Procedures*, GC23-0669, IBM Corporation; available through IBM branch offices.

Gerald E. Boehm *IBM Kingston Programming Laboratory, Neighborhood Road, Kingston, New York 12401.* Mr. Boehm is a staff programmer in the Distributed Systems Build, Release, and Tools department in IBM's Kingston Programming Laboratory. He joined IBM in 1973 as a junior programmer in the Distributed Computing Facility project in Poughkeepsie, New York. In 1977 he transferred to Kingston and joined the DPPX development project. He has participated in design, code, test, and inspection activities for the DPPX linkage editor and stand-alone input/output processor. Mr. Boehm accepted a 19-month assignment in Sweden to help develop procedures, tools, and test cases for the DPPX/APL project. Since 1983 he has been involved with the design and implementation of automated procedures to improve various DPPX development activities. He earned his B.S. in physics from St. Francis College, Loretto, Pennsylvania, and an M.S. in computer science from Pennsylvania State University. He received training as a computer system operator while in the United States Air Force.

Arthur M. Palmiotti *IBM Kingston Programming Laboratory, Neighborhood Road, Kingston, New York 12401.* Mr. Palmiotti is a project programmer in the Distributed Systems Test department in IBM's Kingston Programming Laboratory. He joined IBM in 1984 as a junior associate programmer and concentrated his efforts on testing device attachment and support. Since 1985 he has been an active member of the DPPX I/O council which determines

what devices can be supported and identifies development and test requirements. Other responsibilities include system test planning, stress testing, and customer system testing. Mr. Palmiotti received his B.S. in mathematics and computer science at the State University of New York at New Paltz in 1983.

David P. Zingaretti *IBM Kingston Programming Laboratory, Neighborhood Road, Kingston, New York 12401.* Mr. Zingaretti is an advisory programmer in the Distributed Systems Programming Development department in IBM's Kingston Programming Laboratory. He joined IBM in 1981 to work in DPPX communications networking. From 1981 through 1986 he developed DPPX operating system enhancements with primary emphasis on the command facilities, environment management, DASD queue and X.25 components. From 1986 to 1988 he was responsible for porting various DPPX components from the 8100 Information System to the 9370 Information System. Mr. Zingaretti earned his B.S. in accounting from King's College, Wilkes-Barre, Pennsylvania, in 1975 and an M.B.A. in information systems technology from Marywood College, Scranton, Pennsylvania, in 1983.

Reprint Order No. G321-5390.