Paul Schneck
Guest Editor

Design, Development, Integration: Space Shuttle Primary Flight Software System

The development of Space Shuttle software posed unique requirements above and beyond raw size (30 times larger than Saturn V software), complexity, and criticality.

WILLIAM A. MADDEN and KYLE Y. RONE

The design, development, and integration of the Shuttle on-board Primary Avionics Software System (PASS) have posed unique requirements associated with few other aerospace or commercial software systems. These challenges stem from its size and complexity, its criticality to completion of the Space Shuttle mission, and from the fact that it is only one of many components of an overwhelmingly complex state-of-the-art Space Transportation System (STS).

With respect to size and complexity, the software being readied for the first orbital flight test (STS-1) of the Shuttle is actually eight separately executable programs or memory configurations sharing a common operating system. These programs are stored on a mass memory tape device and are loaded into the on-board computers on crew request (Figure 1). Each is designed to perform the set of support functions required for the different ground and in-flight phases of Shuttle operations. In all, these eight programs, including the software operating system, comprise approximately one-half million 32-bit words of data and executable instructions. The size is at least 30 times that of the Saturn V flight software system.

SOFTWARE IS KEY TO SHUTTLE FUNCTIONS

Without software, the Space Shuttle cannot fly. There are few functions integral to the Shuttle operation for which the software does not perform computational services. It is responsible for the guidance, navigation, and flight control functions performed during all flight phases. This includes both the gathering of environment and sensor input data and the issuing of commands to the vehicle effectors (engines and aerosurfaces). It supports all vehicle/ground interface functions with the Launch Processing System at the Kennedy Space Center prior to vehicle lift-off through the launch data bus (LDB). During in-flight operation, the network signal processing (NSP) interface functions are used for processing of data and/or commands received from the Mission Control Center at the Johnson Space Center. Other software functions include the management and monitoring of on-board systems, fault detection and annunciation, and preflight and preentry checkout and safing procedures.

To obtain the required "Fail-operational/Fail-safe" reliability, the software in certain critical flight phases must execute redundantly in multiple computers. To achieve this redundancy, an intercomputer synchronization scheme has been developed to guarantee identical inputs and outputs from the redundant computers. It also provides such functions as computer synchronization at rates of up to 330 times per second and control of input data to ensure that all computers receive identical information from redundant sensors whether or not hardware failures have occurred.

Above and beyond the size, complexity, and criticality of the software, several other factors contributed to complexity of the development problem. The overall

The present tense of this article, as published in 1980–1981, has been retained in republication.

Shuttle program schedules required that the software be certified and ready to support the first orbital flight. However, the detailed definition of all requirements could not be completed in time to support a proven software design, implementation, and verification development cycle (Figure 2) due to the ongoing vehicle engineering analysis work. Additionally, the Orbiter avionic integration and certification activities performed at Houston, Downey, California, Palmdale, California, and at the Kennedy Space Center required the use of the software very early in the development cycle to accomplish certification responsibilites. To satisfy these conflicting demands and still deliver a fully verified, error-free software system consistent with Shuttle flight schedules, a development strategy was evolved that preserved the effectiveness of the proven development cycle and satisfied the customer requirements. This paper describes the major elements of the development strategy that evolved. Aspects of the succeeding verification and maintenance phases are not addressed here.

EARLY INVOLVEMENT IN CUSTOMER REQUIREMENTS

From an idealistic viewpoint, software should be developed from a concise set of requirements that are defined, documented, and established before implementation begins. The requirements on the Shuttle program, however, evolved during the software development process. The requirements were developed over a long period of time with significant change activity occurring after each baseline (Figure 3). Strong interfaces with the requirements originators were developed to gain an early understanding of the changes. Used in the development planning process, this insight enabled accurate and timely software deliveries to users.

Several factors contributed to the changes in the requirements baseline. Primary among these was the timing of the vehicle test program. Because test facility resources were being established concurrently and the vehicle was not available, critical aerodynamic and structural tests were scheduled after the initial set of detailed requirements was provided. The initial requirements were formulated with the intent of incorporating the results of these tests with data changes only; however, these goals were not completely realized and some significant software design changes resulted.

The second most significant factor affecting the requirements was on-board computer resources (core and CPU). Early in the development cycle, projections indicated that the computer capacity would be exceeded in both size (core) and load (CPU). After the initial soft-

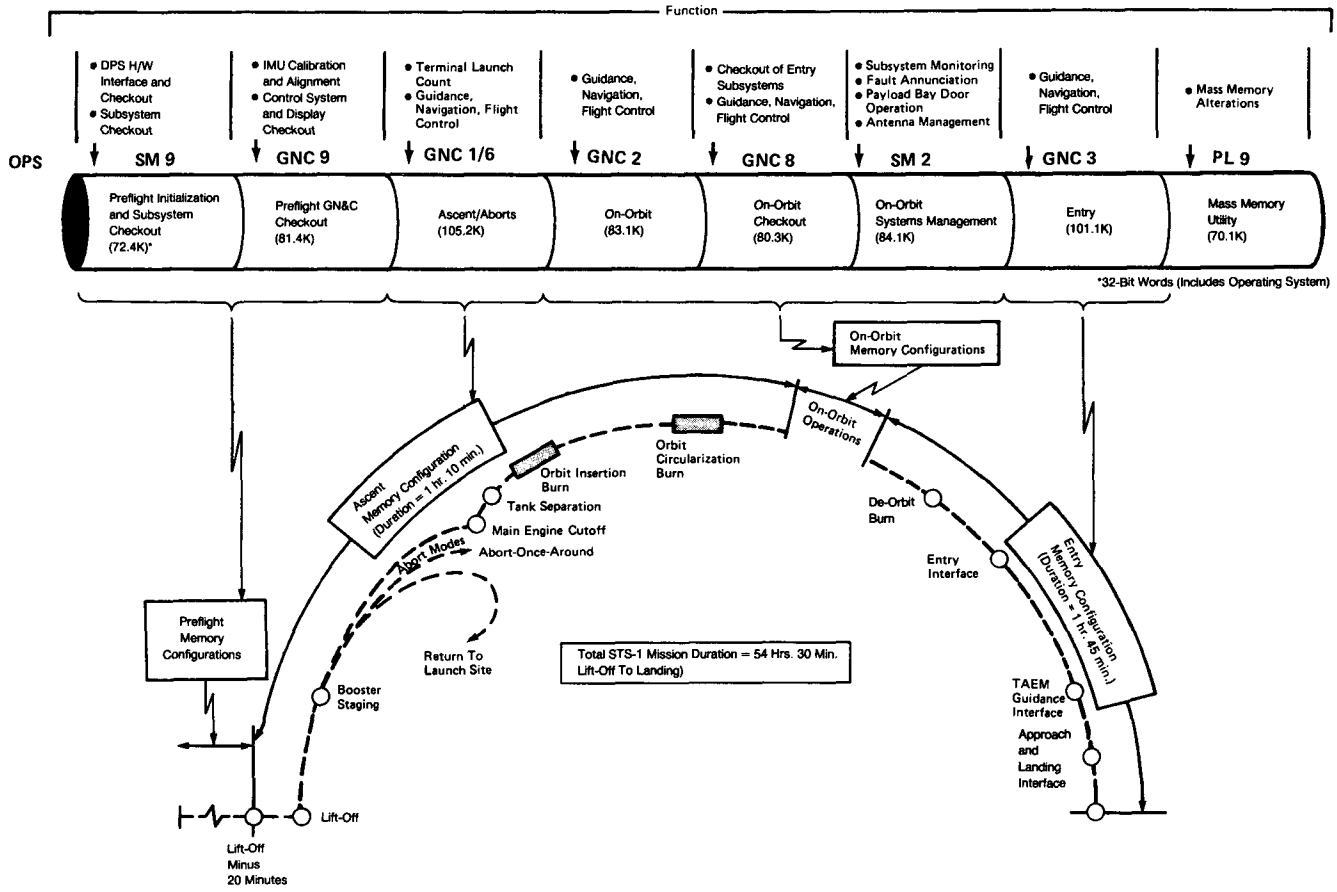


FIGURE 1. Shuttle Mission Profile and Software Memory Configurations

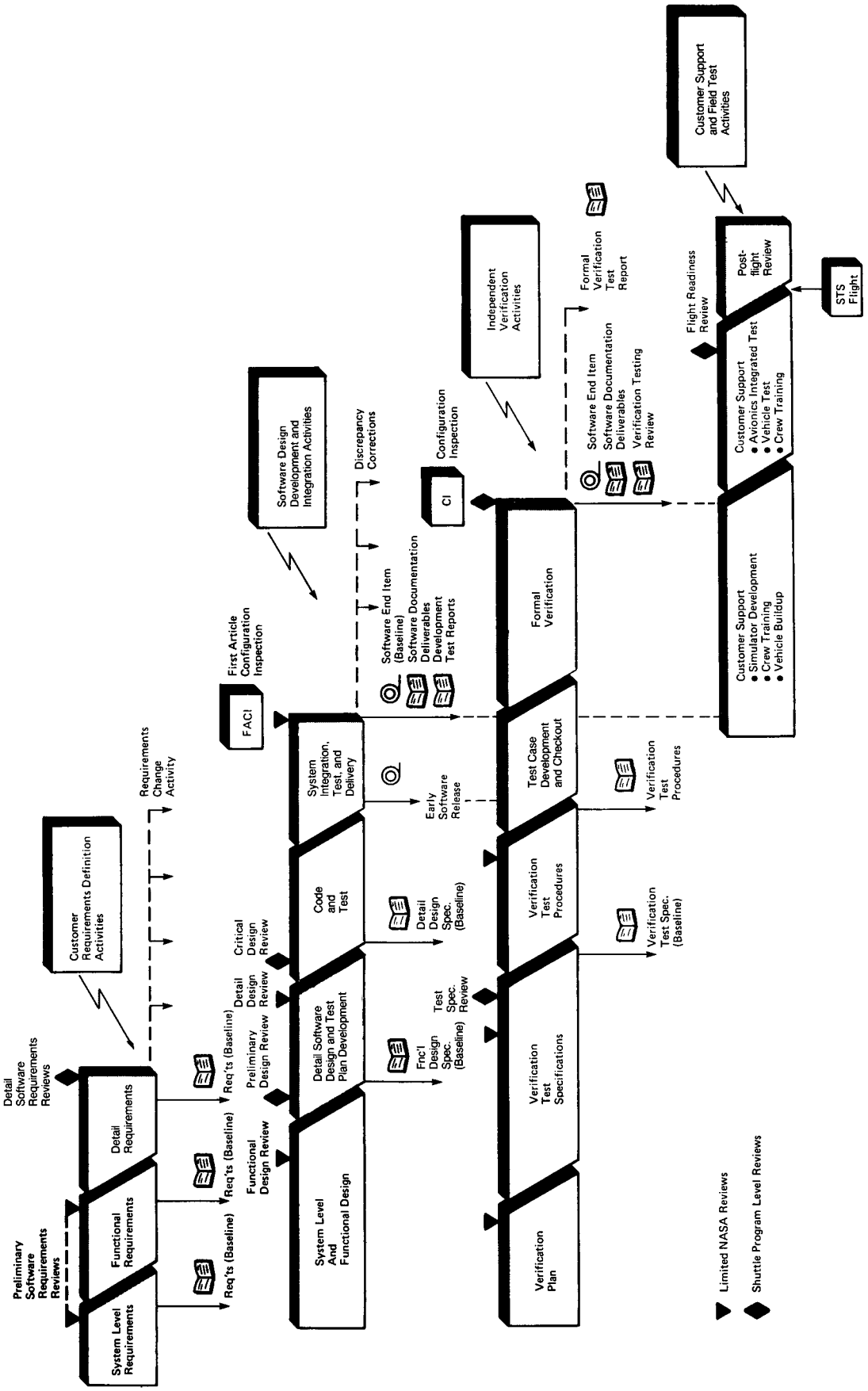


FIGURE 2. Software Development Life Cycle

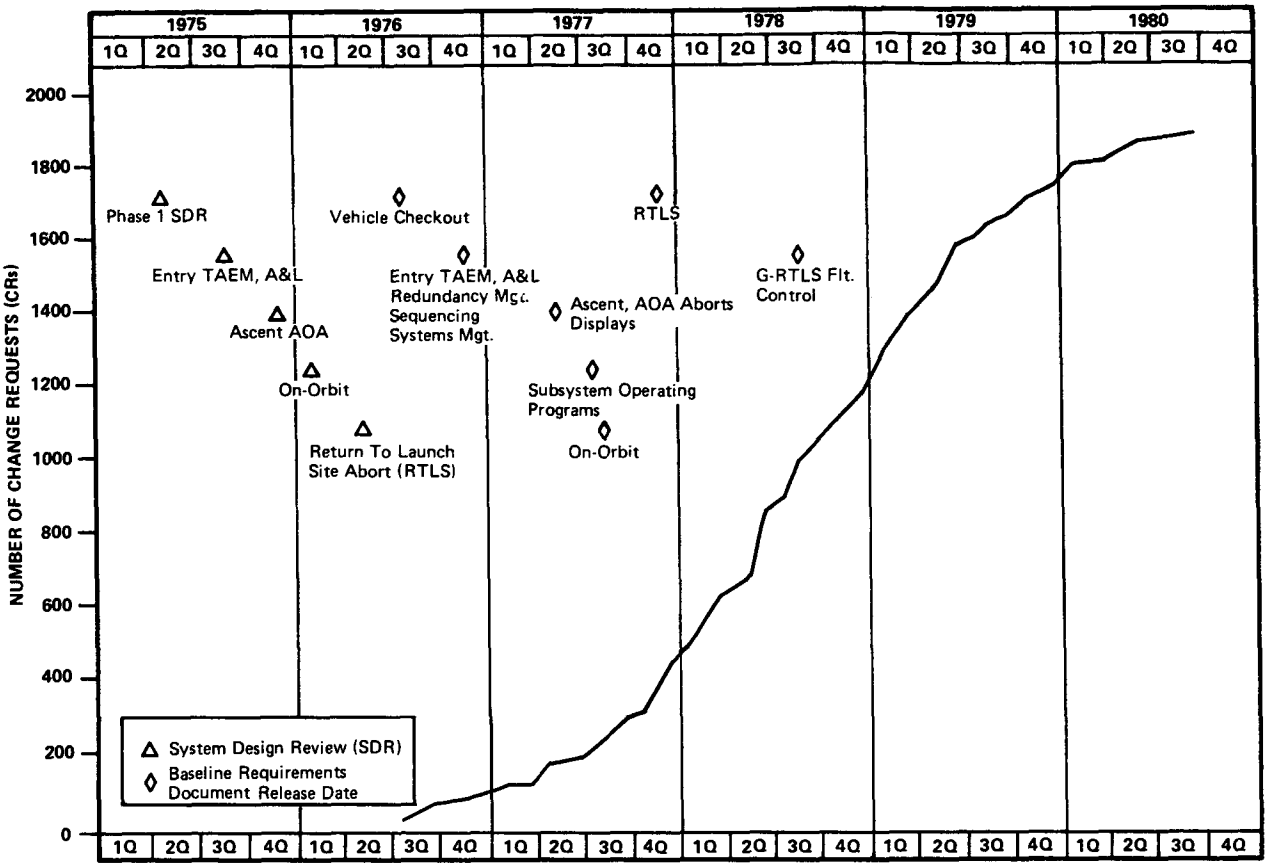


FIGURE 3. STS-1 Flight Software Requirements Change Requests (CRs)

ware design optimization, it became obvious that the only way to solve the problem was to rework the requirements. This took two forms: deletion of functions and reduction of execution rates. These items caused changes in both the software architecture and the detailed design.

Another factor that strongly influenced change activity was exposure of the software to the vehicle and laboratory test environments where real hardware was available, operational procedures were used, and flight crews were training. In many cases, it was found that the real hardware interfaces differed from those in the requirements, operational procedures were not fully supported, and additional or modified functions were required to support the crew. Again these changes fed back into both the architecture and detailed design.

Experience from the Approach and Landing Test (ALT) program had led both NASA and IBM management to anticipate these problems. A requirements analysis group was formed to provide a systems engineering interface between the requirements definition and software implementation worlds and to effect an understanding of the requirements of each. They would be the primary people to identify requirements and design trade-offs and clearly communicate the implications of the trades to both worlds. This approach proved to be effective and made it possible to accommodate the changing requirements without significant cost or schedule impacts.

REQUIREMENTS IMPLEMENTATION PLANNING

In establishing an initial implementation plan, several software development and Shuttle program objectives were considered, including the following:

1. Implement the most mature requirements first to minimize rework.
2. Release software for verification/certification as soon as possible for maximum exposure and testing.
3. Support certification of simulation/training facilities.
4. Support Orbiter fabrication, checkout, and integration at Palmdale and the Kennedy Space Center.

Due to the size, complexity, and evolutionary nature of the program, it was recognized early that the ideal software development cycle (Figure 3) could not be strictly applied and still satisfy the objectives. However, an implementation approach was devised for STS-1, which met the objectives by applying the ideal cycle to small elements of the overall software package on an iterative basis (Figure 4).

This approach was based on incremental releases. The releases were first separated into flight phases or

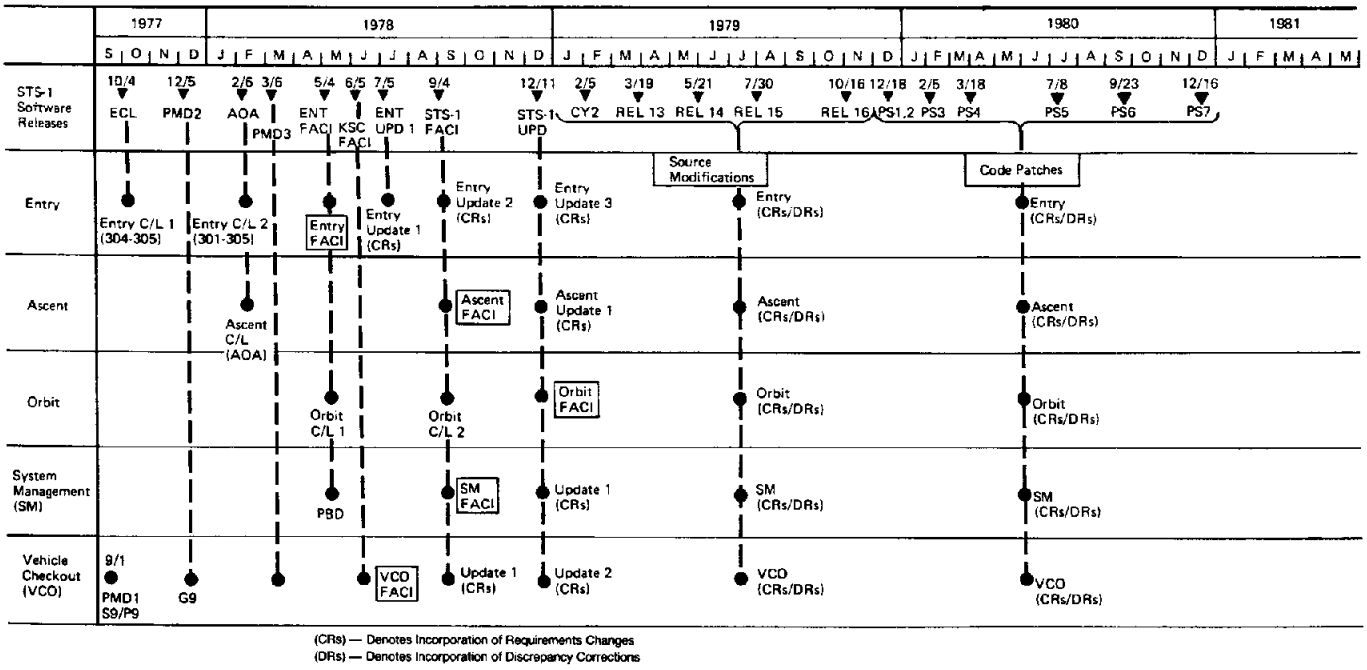


FIGURE 4. Interim Flight Software Releases

memory configuration, i.e., entry, ascent, and vehicle checkout. The first drop for each release represented a basic set of operational capabilities and provided a structure for adding other capabilities on later drops. The development of the full set of baseline capabilities for each release culminated at a first-article configuration inspection (FACI) point, which marked the beginning of the verification effort for that release.

The STS-1 software development program has had 17 interim release drops in a 31-month period starting in October 1977 (Figure 4). Although full software capability was provided after the ninth release in December 1978, an additional eight releases of the software have been necessary to accommodate the continued requirements changes and discrepancy correction activity inherent in large, complex, first-of-a-kind software systems.

This incremental release approach satisfied the original objectives. The mature portions of the Orbital Flight Test (OFT) software were those parts most common to the ALT program such as "entry through landing" and "vehicle checkout." These were developed first. Other parts were built and integrated into the system incrementally until the final FACI release was reached.

The second and third objectives were uniquely satisfied by the development approach. The software was exposed in small increments to both verification and field users. This allowed early identification of software discrepancies and eased problem resolution. The software was incrementally exposed to the simulators. Thus, the simulator checkout was completed in an environment where the number of variables could be controlled, thus easing problem isolation.

The last objective of supporting vehicle test was accomplished by phasing the vehicle-checkout function development on the same schedule as that of the vehicle fabrication and integration. Initial releases supported the vehicle fabrication test at Palmdale. Later releases incorporated additional capabilities required to support total system integration and test at Kennedy Space Center.

FORMULATION OF DEVELOPMENT STANDARDS

The early formulation of development standards covered both design and implementation. Following an across-the-board review, the standards were baselined. Deviation from the baseline required management and change control board approval. Compliance with all development standards was checked during each design/code inspection and a postdevelopment audit with deviations documented by discrepancy reports (DRs). Here are seven subjects that are addressed by the development standards:

- redundant computer operation/synchronization;
- data homogeneity;
- processor and I/O rates, priorities, and phasing;
- interprocess data protection;
- program structuring and language utilization;
- module/data naming conventions;
- design documentation and code commentary.

THE DESIGN AND IMPLEMENTATION PROCESS

The architectural foundation for the OFT flight software (Figure 5) was the ALT system with six primary features:

Flight computer operating system (FCOS) to support redundant computer operations/synchronization and the basic functions of process management, I/O management, and DPS configuration management. Also included is the set of service macros (SVCs) for the software interface to the FCOS and external hardware.

System control (SC) functions to support system initialization, memory overlay/loading, and DPS configuration initialization.

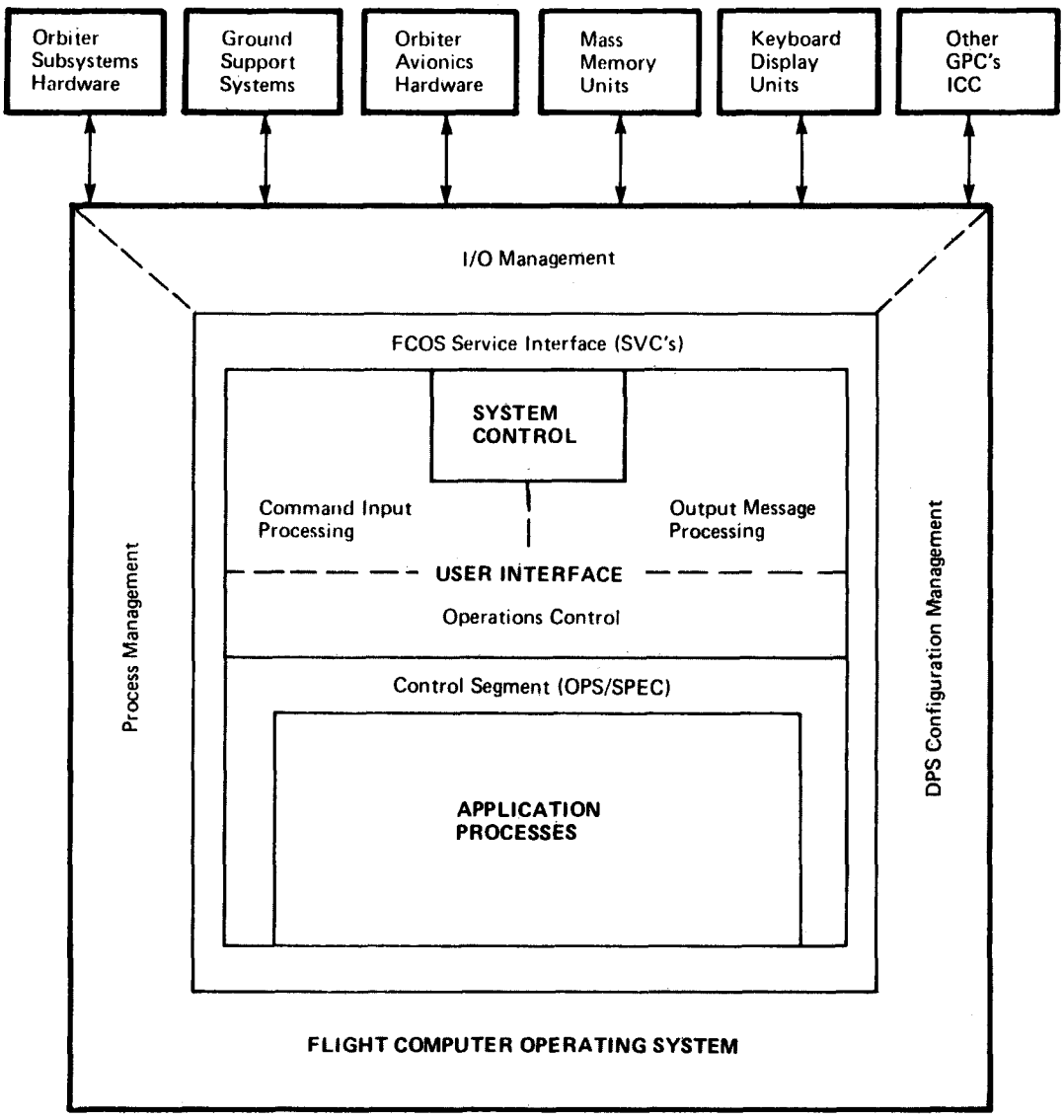
User interface (UI) functions to support user input processing, output display/message generation, and applications process controls. A set of macros called the control segment grammar provides the capability to develop standard application control logic and display/keyboard interface structures.

Flight software system generation and maintenance facilities, including the HAL/S compiler, IBM AP-101 assembler, linkage editor, program library management, and mass memory build facilities.

Software Development Laboratory (SDL), including the flight software system generation and maintenance facilities and the facilities to simulate environment and vehicle subsystem operations with which the flight software could be tested and debugged.

Basic GN&C entry, system management, and vehicle checkout applications software.

In addition to the system foundation used from ALT, the management and technical experience gained in ALT also was beneficial. To successfully implement the



□-On-board software

FIGURE 5. Software Architecture

much more complex OFT flight software, a more disciplined and structured development approach was followed (Figure 6). Increased emphasis was given in the "front end" aspects of the development cycle, including requirements definition, system design, standards definition, top-down development, and identification of development tools requirements and the resultant tools (Figures 7 and 8). Similarly, during implementation, added emphasis in design/code reviews and testing helped achieve the required software reliability within customer schedules.

In addition to steps taken to participate in the requirements definition and the development of an incremental release strategy, a significant degree of planning was accomplished relative to the design implementation process. This addressed both the implementation and maintenance of the existing ALT system, and the development of the top-level OFT design structure. Procedures for development of the design structure, modification enhancements of the base ALT system, and implementation of new OFT functional and detail requirements were put into place.

Very early in the development cycle, while the requirements definition was in process, a small group of the more experienced programmers (system design team) designed the control segment structures for the different memory configurations required for OFT. Simultaneously, the existing ALT system software (FCOS/SC/UI) was installed in new OFT program libraries. It was checked out to ensure its use as a base to implement modifications needed for OFT to reduce size, improve reliability and performance, and extend capabilities.

The design team developed top-level control segment structures, which were implemented and tested with ALT base system software. As the requirements definition process evolved, modifications were made to implement more detail or lower level aspects of the structures. Where anticipated but undefined requirements were known, "stubs" were implemented. Stubs enable software linkage to proceed without execution, and thus, testing can be continued. Throughout this evolutionary process, the implemented structure was tested on a continuing basis to ensure the overall system sta-

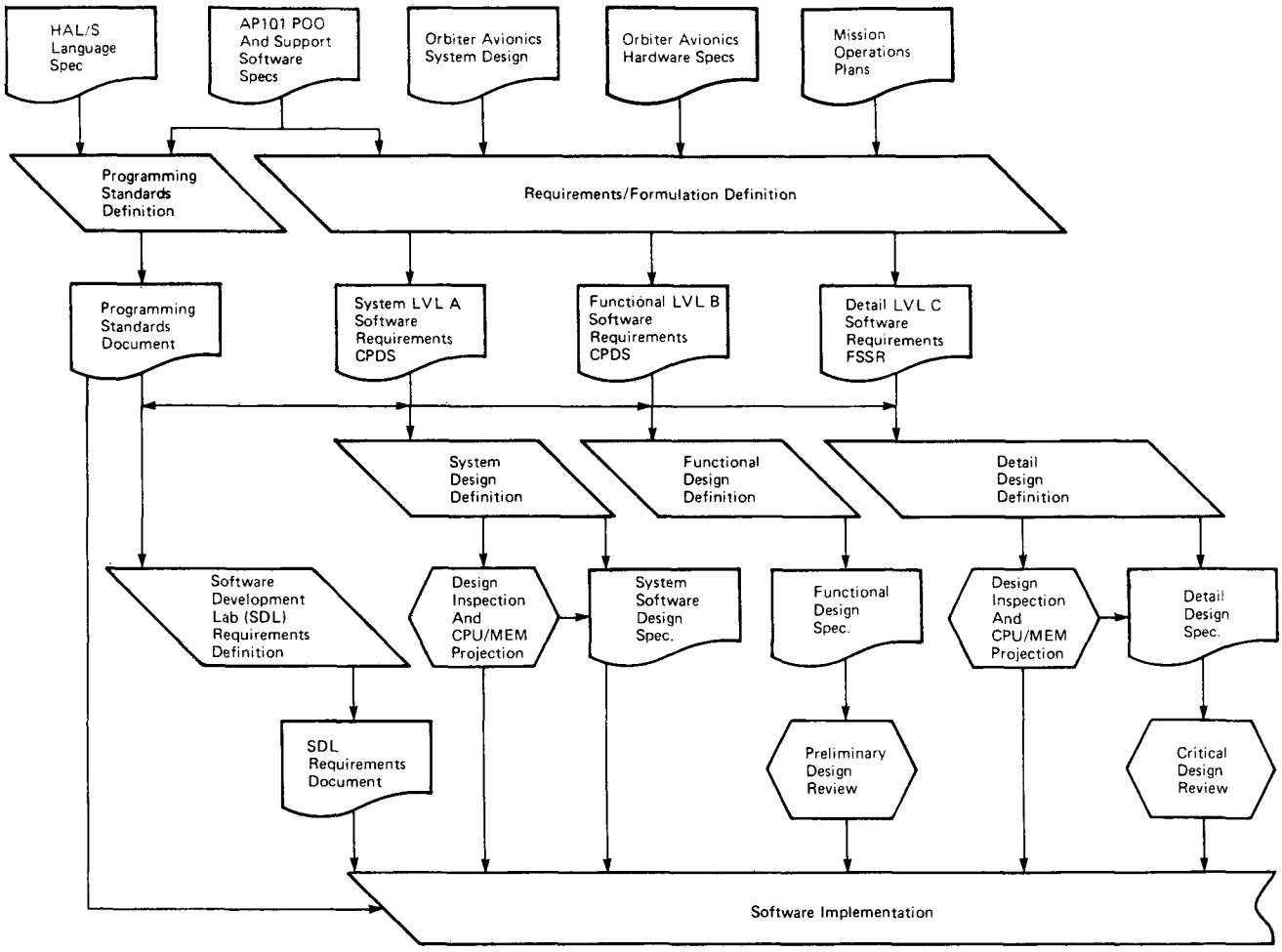


FIGURE 6. Fflight Software Requirements and Design

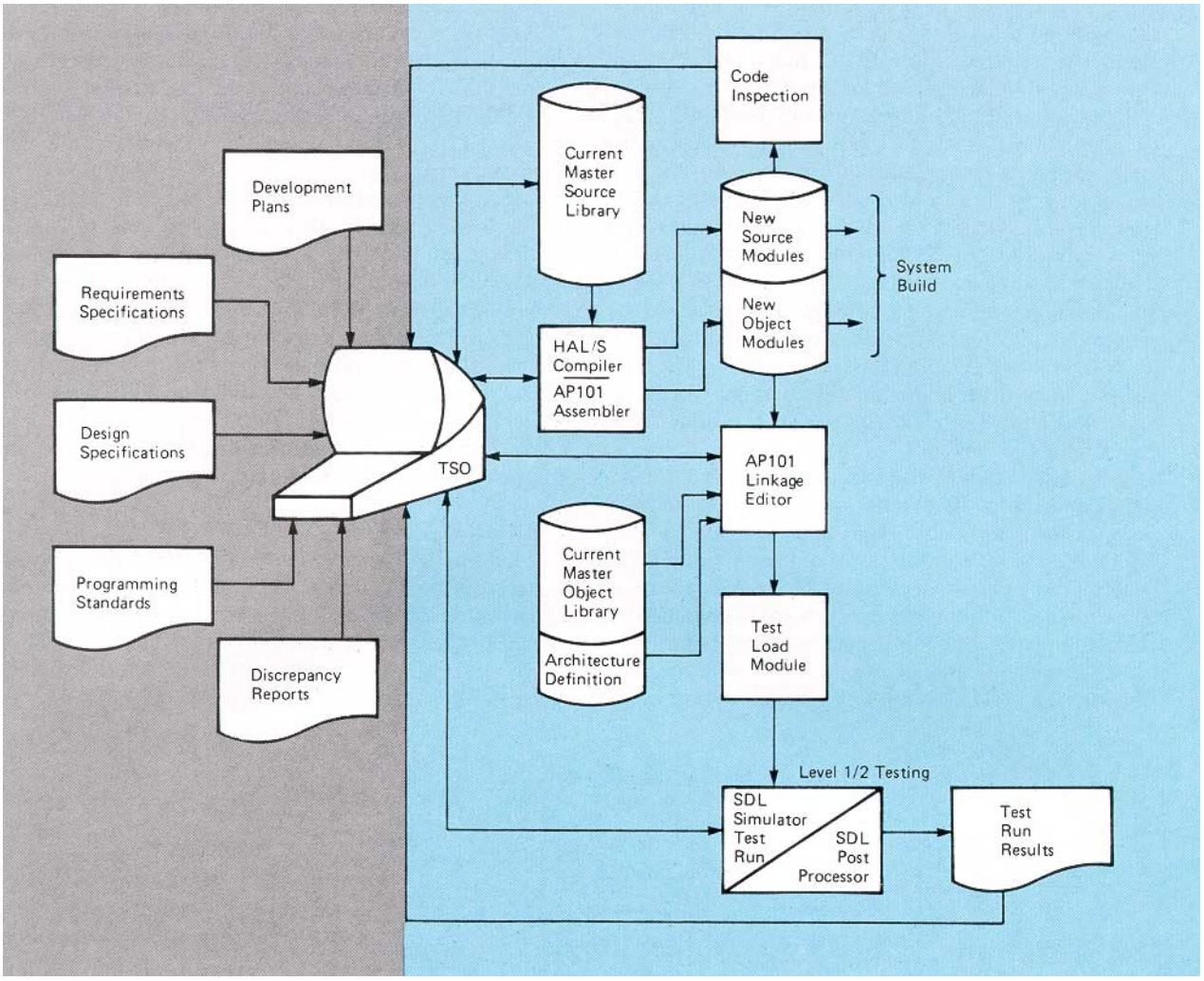


FIGURE 7. Flight Code Generation

bility. Continued testing established a sound building-block approach and also provided training valuable to programmers for interfacing with the system and for learning software implementation and configuration control procedures.

When the definition of system (Level A) and functional (Level B) requirements was accomplished and baselined at system design reviews, the software functional design was completed and documented. Major elements were the memory size and CPU loading projections that were developed based on the overall system structure that had been implemented and the anticipated detail requirements. A preliminary design review was held with NASA and associate contractors to critique and approve the design. This established a baseline for subsequent detailed requirements and design development.

As the detail (Level C) requirements and associated design evolved, the development environment became more production oriented with an increased number of people involved. The design team was responsible for the overview and consistency of all elements of the detailed design. Memory and CPU projections were updated on a continuing basis. This process generated a detailed design containing a "code to" level of detail, including module structure and interfaces, database definition and organization, equations and algorithms, I/O data tables and interprocess variable data protection. Upon completion of the detailed design for each module, a formal design review was held with analysts and programmers to assure compliance with requirements and standards, correctness, completeness, efficiency, and adequacy of interfaces. Design inspections were tracked during development and the results documented. When the detail design for all software was completed, a critical design review was held with the Shuttle community where the design was approved and baselined for implementation.

The implementation phase was performed with the same attitude toward understanding, completeness, consistency, and overall planned system approach as was done for the design phase. The preparation and

development testing of Orbiter flight code utilized the same ground rules of top-down, structured development. The resource of the HAL/S high-level language, which is particularly suited for top-down structured coding, was especially helpful during this phase. This resource permitted coding the functional design of the major elements of the flight software system. The higher level modules were coded while leaving the lower levels as undefined and (for the time) unneeded stubs. This orderly procedure was very useful because it allowed coding and testing of the higher level logic and algorithms in the total development process. To generate flight code, the production and test facilities of the SDL are used (Figure 7). Use of a high-level language coupled with improved development techniques and tools doubled productivity over comparable Apollo development processes.

Each coded module of software was subjected to a code inspection with an audit team to ensure that the code was consistent with requirements, design, and standards, and efficient in terms of memory and CPU. Each review process was tracked in the software development plans, and review results were documented. Upon completion of module coding, review, and unit testing (Level 1), each module was scheduled for inclusion into the baseline master system. This was a continual process since the master system was updated on a three-week cycle. Postbuild testing was performed before release of the new master system to ensure continued stability. Build results were documented and widely distributed to the project to provide visibility into the status of the integration process and the master system (Figure 8).

As a flight software release neared completion, a final programming standards audit was performed. This audit was conducted using both automated and manually generated data and emphasized multicomputer redundant set operations, interprocess variable data protection, overlap of data processing with I/O, process scheduling and termination, and restricted instructions and sequencing. The status and results of this audit were presented at a FACI, which marked the completion of the baseline requirements implementation and the start of formal verification.

AN INTEGRATED TEST APPROACH

The improved implementation methods and controls used during the development process help to produce software with fewer latent errors. However, assurance that the software is error-free can be gained only by a well-structured form of testing. Early in OFT, determination was made that an integrated test approach was

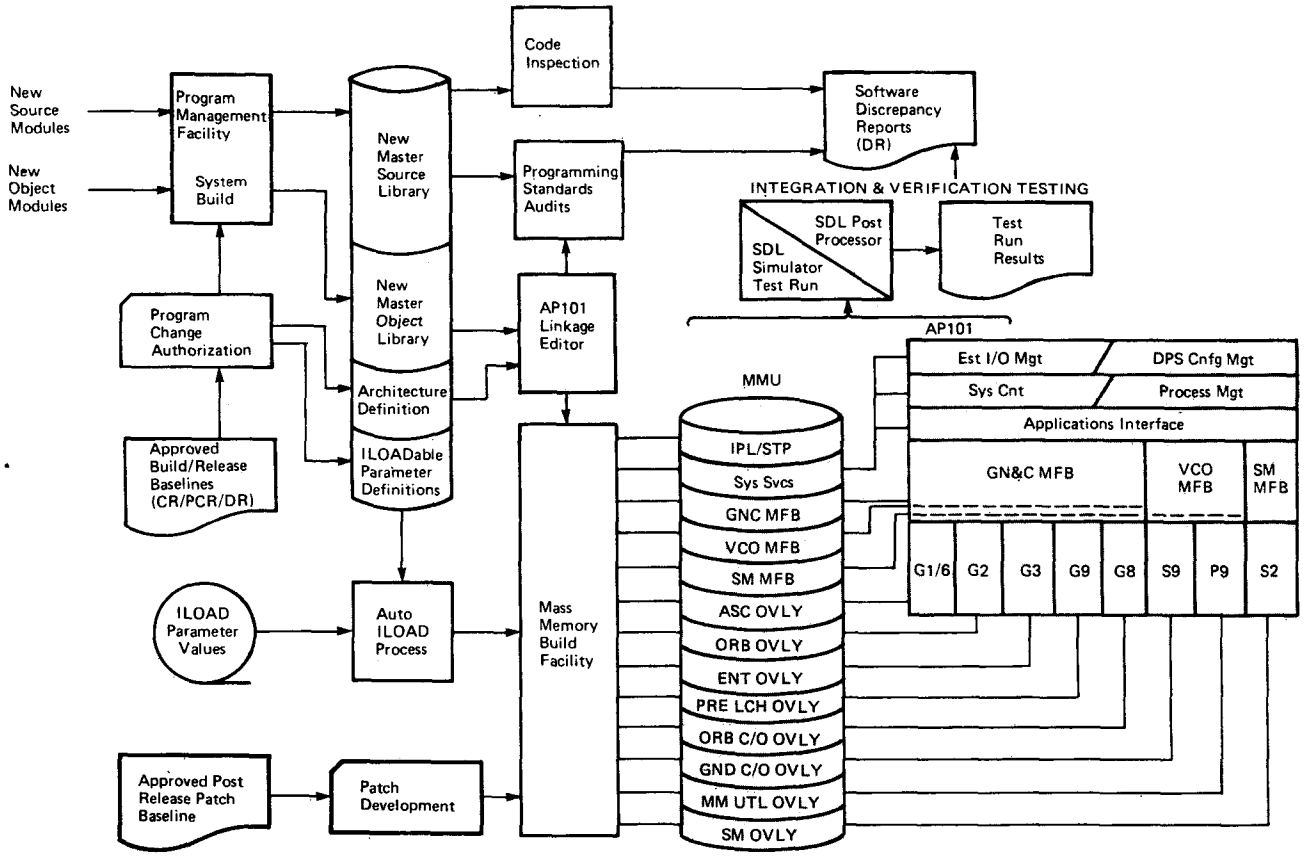


FIGURE 8. Flight Software Integration

required to control the testing process across the project. Several goals were set forth to ensure a successful test approach: establishment of documentation and control to ensure visibility into the testing process, establishment of test execution and documentation standards, and parallel test planning during the design process. The key element of this test approach, however, was development of a test management approach that emphasized a hierarchical ordering of development tests that allowed for continual integration of program parts as they were developed and a systematic sequence of evaluation tests on the flight software system (Figure 9).

During the development period, compilation units were added to the master system via the system build process, which was invoked cyclically. Parts of the processing associated with each cyclic master system update were tested to determine the preservation of the software's basic capabilities on that particular master system update. Also, more detailed tests were used to determine the quantitative status of the new capabilities that had completed testing. The former testing was termed "regression testing"; the latter, "new capabilities testing." All specified test plans were documented in an integrated test plan that covered all phases of testing.

Level 1 Testing (Unit)

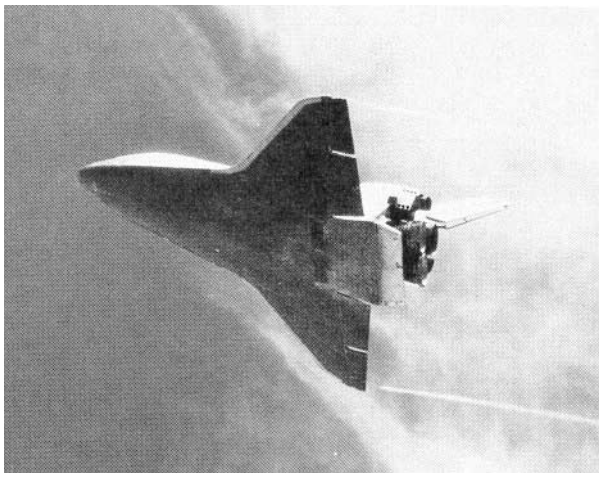
During the development activity, specific testing was done to ensure that the mathematical equations and logic paths provided the results expected. These algorithms and logic paths were checked for accuracy and, where possible, compared against results from external sources and against the system design specification (SDS). The testing activity occurred in parallel with new capability testing but was accomplished by the development programmer. The test results were documented by means of a unit test checklist.

Level 2 Testing (Functional)

The Level 2 facet of the development test activity was similar to the Level 1 testing. However, the Level 1 testing described above was expanded to test modules that interfaced with each other in the total functional environment and that are required to satisfy a specific user input command. It combined modules that by design operated in conjunction with each other and tested them as a function against the SDS and the requirements. This activity was accomplished in parallel with ongoing new capability testing. Test results were documented in development test reports.

Level 3 Testing (Subsystem)

Level 3 testing demonstrated the ability of a subsystem to execute its nominal functions in a simplex flight computer environment (e.g., fly an ascent trajectory or perform self-test of part of the vehicle hardware). These tests were the first real indicators of the software performance as an integrated system. All facets of the applications programs from the integrity of the algorithms to the interfaces with the system software were exercised. Completion of the Level 3 tests was one of the key milestones in the path to releasing a system for verification and the field usage. The test results were documented in development test reports.



Space Shuttle "Enterprise," as it flew during the Approach and Landing Tests, provided beneficial technical and management experience needed for the more disciplined and structured development approach used in developing the more complex Orbital Flight Test flight software.

Level 4 Testing (System)

Level 4 testing exercised control logic interfaces, operational sequence (OPS) transitions, mode-to-mode transitions, specialist function (SPEC) operations, and display processing in a multiple flight computer environment. Inter- and intracomputer interfaces (overlays, data transfers and timing, and process synchronization) were tested to check the hardware and software interfaces in the SDL environment. The test results were documented in the development test reports.

Level 5 Testing (Release Validation)

Prior to delivering the software to field users, the Level 4 tested end item was loaded into a hardware mass memory and a system test was executed in one of the NASA simulation/training facilities. This was to verify that the delivered software would function in the most realistic hardware environment available. The test results were presented with the delivery.

CONFIGURATION CONTROL

One of the most complicating factors that affected the development of the Orbiter avionics software was the extremely large number of people involved. This included not only the programming staff but also those involved in requirements definition, SDL development, verification, and field support. Coupled together with the previously mentioned high degree of requirements

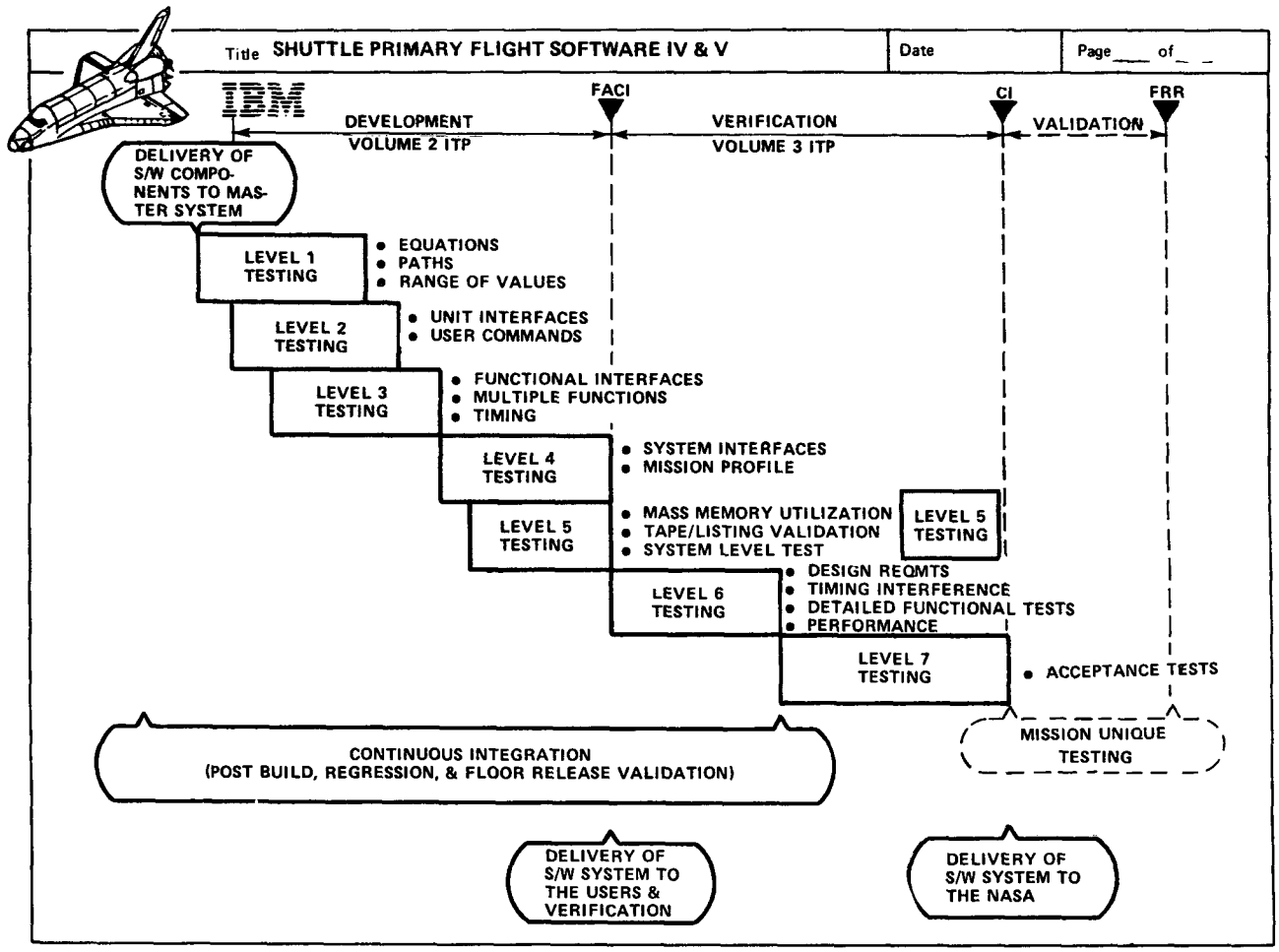


FIGURE 9. Levels of Testing

change, the incremental releases of early versions of the software, and the overall size and complexity of the software itself, a very complicated configuration management problem was created.

In order to gain control of this situation, an internal control/coordination board structure was established. This started with a requirements review board (RRB), where the assessment of all requirements changes was coordinated. A baselines control board (BCB) was established to coordinate and control the system build/release schedule planning and associated content configuration definition/control. Eventually the structure was expanded to include five different review/control boards (Figure 10). Results, actions, and recommendations of these five independent boards were coordinated through a project baselines control board, which in turn interfaced with spacecraft software division configuration control board (SSD CCB) and the orbiter avionics software control board (OASCB).

Membership of the internal review boards included representatives from all affected project areas. This projectwide representation enhanced communication among functional organizations and provided a mechanism to achieve strict configuration control. Each board that was responsible for assessing and scheduling changes kept an up-to-date log of all recommendations that were brought to the BCB for approval. After review, all items that were approved were documented in a project baseline report. Changes not authorized by this report were not allowed on builds. Similarly, items scheduled, but not supported, were analyzed very thoroughly.

Changes to approved configuration baselines, which evolved from design changes, requirements change requests (CRs), or software discrepancy reports (DRs), were coordinated through the appropriate boards (internal) and ultimately approved by NASA. This structure allowed an internal coordination of a total impact and then provided one central, coordinated assessment to the NASA control boards reflecting IBM's position or changes. This applies to both application software baselines and support software.

Documentation of approved baselines was subsequently reported and monitored in project management plans, orbiter management review monthly presentations, and Shuttle avionics software schedule baseline

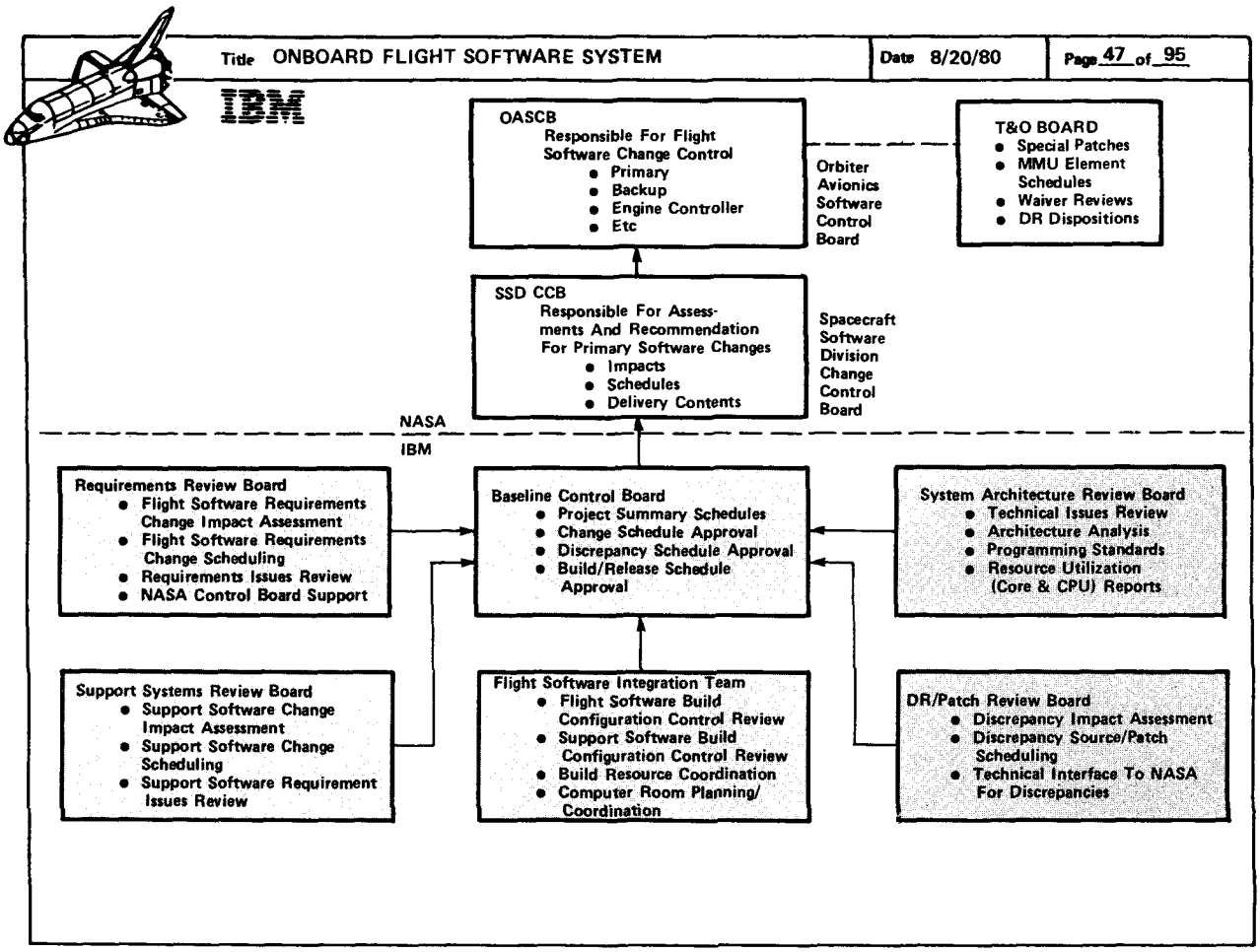


FIGURE 10. Configuration Control Boards Structure

and planning data package. Audits to verify consistency between approved baselines and reported baselines were performed weekly by the project office.

SUMMARY

Since the software package is an integral and critical part of the Shuttle systems, a development and testing approach was employed that ensured that the software met customer requirements, performed in accordance with Shuttle operational requirements, and was delivered to users with minimum errors. In order to develop and deliver a software system that met these goals, the development organization addressed the following areas:

- early involvement with software requirements generation,
- development of a reasonable requirements implementation plan,
- early identification of development standards,
- utilization of top–down, structured implementation techniques,
- establishment of design and code reviews and audits,
- establishment of an integrated test approach for the entire development process, and
- configuration control of the incremental build and integration of the evolving software system.

CR Categories and Subject Descriptors: C.4 [**Performance of Systems**]—*reliability, availability, and serviceability*; D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.9 [**Software Engineering**]: Management—*software quality assurance*; D.4.5 [**Operating Systems**]: Reliability; J.2 [**Physical Science and Engineering**]—*aerospace*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development, software maintenance*

General Terms: Design, Management, Reliability, Verification

Additional Key Words and Phrases: avionics system, PASS, space shuttle

Authors' Present Address: William A. Madden and Kyle Y. Rone, IBM, Federal Systems Division, 1322 Space Park Drive, Houston, TX 77058.