Center for Devices and Radiological Health

GUIDANCE FOR INDUSTRY

GENERAL PRINCIPLES OF SOFTWARE VALIDATION

DRAFT GUIDANCE
Version 1.1

This guidance is being distributed for comment purposes only.

Draft released for comment on: June 9, 1997

Comments and suggestions regarding this draft document should be submitted by October 1, 1997 to the Dockets Management Branch (HFA-305), Food and Drug Administration, Room 146, 12420 Parklawn Drive, Rockville, Maryland 20857. For questions regarding this draft document, contact Stewart Crumpler, Office of Compliance, CDRH, 2098 Gaither Road, Rockville, Maryland 20850, Phone: (301) 594-4659, FAX: (301) 594-4672, E-mail: ESC@CDRH.FDA.GOV

U.S. Department of Health and Human Services
Food and Drug Administration
Center for Devices and RadiologicalHealth

June 1, 1997

PREFACE

As of June 1, 1997, the medical device quality system regulation is effective, including requirements for implementing design controls. Those design controls include requirements for validation of software used in medical devices. Further, the quality system regulation continues the requirement that when software is used to automate a production process or a part of the quality system, that software must be validated for its intended use. Both medical device manufacturers and FDA staff have requested further guidance regarding the meaning of these requirements, and what is needed to comply with them. The attached draft guidance on "General Principles of Software Validation" was developed in response to these requests, and is provided for your review and comment.

The subject of software validation can be very complex, and no single document is likely to provide all the information needed for every manufacturer in every circumstance. Manufacturers have wide latitude in choosing which particular techniques will be combined into their software validation program. This draft guidance is intended as a general overview of software validation principles, and as a beginning point for use in evaluating software validation programs.

The official comment period for this draft guidance will be July 1, 1997 through October 1, 1997. We request that your comments be as specific as possible, with references to specific sections and paragraphs in the document, and with specific proposals for recommended changes. Instructions for submission of your comments appear on the title page, along with the Office of Compliance contact for any questions you may have about the document.

Thank you for your participation in the comment and review process. I am certain it will make the final guidance document more useful for all of us when it is issued.

Lillian J. Gill
Director
Office of Compliance
Center for Devices and Radiological Health

Table of Contents

I. PURPOSE

This draft guidance outlines general validation principles that the Food and Drug Administration (FDA) considers to be applicable to the validation of medical device software or the validation of software used to design, develop or manufacture medical devices.

II. SCOPE

A. Applicability

This draft guidance is applicable to all medical device software, including blood establishment software, and to software used to design, develop or manufacture medical devices. This draft guidance document represents the agency's current thinking on validation of software related to medical devices. It does not create or confer any rights for or on any person and does not operate to bind FDA or the public. An alternative approach may be used if such approach satisfies the requirements of the applicable statute, regulations or both.

This document discusses how the general provisions of the Quality System Regulation apply to software and the agency's current approach to evaluating a software validation system. For example, this document lists validation elements which are acceptable to the FDA for the validation of software; however, it does not list all of the activities and tasks that must, in all instances, be used to comply with the law.

This document does not recommend any specific life cycle model or any specific validation technique or method, but does recommend that software validation activities be conducted throughout the entire software life cycle. For each software project, the responsible party should determine and justify the level of validation effort to be applied, and the specific combination of validation techniques to be used.

This document is based on generally recognized software validation principles and could therefore be applicable to any software. For FDA purposes, this draft guidance is applicable to any software related to a regulated medical device as defined by Section 201(h) of the Federal Food, Drug, and Cosmetic Act (Act) and by current FDA software and regulatory policy. It is not the intent of this document to determine or identify specifically which software is or is not regulated.

B. Audience

The FDA intends this draft guidance to provide useful information and recommendations to the following individuals:

- Persons subject to the Medical Device Quality System Regulation
- Persons responsible for the design, development or production of medical device software
- Persons responsible for the design, development, production or procurement of automated tools used for the design, development or manufacture of medical devices, or software tools used to implement the quality system itself
- FDA Scientific Reviewers
- FDA Investigators
- FDA Compliance Officers

C. Quality System Regulations for Software Validation

Software validation is a requirement of the Quality System Regulation, 21 CFR Part 820, which was published in the Federal Register (61 FR 52602) on October 7, 1996, and takes effect on June 1, 1997. Validation requirements apply to software used in medical devices, to software that is itself a medical device (e.g., software used in blood establishments), as well as to software used in production of the

device or in implementation of the device manufacturer's quality system.

Unless specifically exempted in a classification regulation, all medical device software developed after June 1, 1997, regardless of its device class, is subject to applicable design control provisions of 21 CFR 820.30. This includes completion of current development projects, all new development projects, and all changes made to existing medical device software. Specific requirements for validation of device software are found in 21 CFR 820.30(g). In addition, formal design reviews and software verification are integral parts of an overall software validation, as required by 21 CFR 820.30(e) and (f).

Any software used to automate any part of the device production process or any part of the quality system must be validated for its intended use, as required by 21 CFR 820.70(i). This requirement applies to any software used to automate device design, testing, component acceptance, manufacturing, labeling, packaging, distribution, complaint handling, or to automate any other aspect of the quality system. Such software may be developed in-house or under contract, but it is most likely to be purchased off-the-shelf for a particular intended use. This off-the-shelf software may have many capabilities, only a few of which are needed by the device manufacturer. The software must be validated for those specific uses for which it is intended. All production and/or quality system software must have documented requirements which fully define its intended use, and against which testing results and other verification evidence can be compared, to show that the production and/or quality system software is validated for its intended use.

D. Quality System Regulation vs Pre-Market Submissions

This document addresses Quality System Regulation issues; specifically, the implementation of software validation. It provides guidance for the management and control of the software validation process. Although software validation is required for automated processes, requirements for process validation are not addressed in this document.

Manufacturers may use the same procedures and records for compliance with quality system and design control requirements, as well as for pre-market submissions to FDA. It is not the intent of this document to cover any specific safety or efficacy issues related to software validation. Design issues and documentation requirements for pre-market submissions of regulated software are not addressed by this document. Specific design issues related to safety and efficacy, and the documentation required in pre-market submissions, are addressed by the Office of Device Evaluation (ODE), Center for Devices and Radiological Health (CDRH) or by the Office of Blood Research and Review, Center for Biologics Evaluation and Research (CBER). See the bibliography for reference to applicable guidance documents for premarket submissions.

Back to Table of Contents

III. DISCUSSION

Many people have asked for specific guidance on what FDA expects them to do to assure compliance with the Quality System Regulation with regard to software validation. Information on software validation presented in this document is not new. Validation of software using the principles and tasks listed in Sections IV and V has been conducted industry-wide for well over 20 years.

FDA recognizes that, because of the great variety of medical devices, processes, and manufacturing facilities, it is not possible to state in one document all of the specific validation elements that are applicable. Several broad concepts, however, have general applicability which persons can use successfully as a guide to software validation. These broad concepts provide an acceptable framework for building a comprehensive approach to software validation. Additional specific information is available from many of the references listed in the bibliography.

A. Software is Different from Hardware

While software shares many of the same engineering tasks as hardware, it is different. The quality of a hardware product is highly dependent on design, development and manufacture, whereas the quality of a software product is highly dependent on design and development with a minimum

concern for software manufacture. Software manufacturing consists of reproduction which can be easily verified. For software, the hardest part is not manufacturing thousands of programs that work alike; it is getting just one program that works to design specifications. The vast majority of software problems are traceable to errors made during the design and development process.

Unlike hardware, software components are rarely standardized. While object oriented development holds promise for reuse of software, it requires a significant up front investment of resources to define and develop reusable software code. Many of the available object oriented development tools and techniques have not yet been adopted for medical device applications.

One of the most significant features of software is branching -- its ability to execute alternative series of commands, based on differing inputs. This feature is a major contributing factor for another characteristic of software -- its complexity. Even short programs can be very complex and difficult to fully understand.

Typically, testing alone cannot fully verify that software is complete and correct. In addition to testing, other verification techniques and a structured and documented development process should be combined to assure a comprehensive validation approach.

Unlike hardware, software is not a physical entity and does not wear out. In fact, software may improve with age, as latent defects are discovered and removed. Again, unlike hardware failures, software failures occur without advanced warning. The software's branching, which allows it to follow differing paths during execution, can result in additional latent defects being discovered long after a software product has been introduced into the marketplace.

Another related characteristic of software is the speed and ease with which it can be changed. This factor can lead both software and non-software professionals to the false impression that software problems can be easily corrected. Combined with a lack of understanding of software, it can lead engineering managers to believe that tightly controlled engineering is not as much needed for software as for hardware. In fact, the opposite is true. Because of its complexity, the development process for software should be even more tightly controlled than for hardware, in order to prevent problems which cannot be easily detected later in the development process.

Repairs made to correct software defects, in fact, establish a new design. Seemingly insignificant changes in software code can create unexpected and very significant problems elsewhere in the software program.

B. Benefits of Software Validation

Software validation is a critical tool in assuring product quality for device software and for software automated operations. Software validation can increase the usability and reliability of the device, resulting in decreased failure rates, fewer recalls and corrective actions, less risk to patients and users, and reduced liability to manufacturers. Software validation can also reduce long term costs by making it easier and less costly to reliably modify software and revalidate software changes. Software maintenance represents as much as 50% of the total cost of software. An established comprehensive software validation process helps to reduce the long term software cost by reducing the cost of each subsequent software validation.

C. Definitions and Terminology

Some commonly used terminology in the software industry can be confusing when compared to definitions of validation and verification found in the medical device Quality System Regulation. For example, many software engineering journal articles and textbooks use the terms "verification" and "validation" interchangeably, or in some cases refer to software "verification, validation and testing (VV&T)" as if it is a single concept, with no distinction among the three terms. For purposes of this guidance and for compliance with the Quality System Regulation, FDA considers "testing" to be one of several "verification" techniques, and considers "validation" to be the umbrella term that encompasses both verification and testing. Unless otherwise specified herein, all other terms are as defined in the current edition of the FDA "Glossary of Computerized System and Software Development

Terminology."

Verification is defined in 21 CFR 820.3(aa) as "confirmation by examination and provision of objective evidence that specified requirements have been fulfilled." In a software development environment, software verification is confirmation that the output of a particular phase of development meets all of the input requirements for that phase. Software testing is one of several verification activities, intended to confirm that software development output meets its input requirements. Other verification activities include walkthroughs, various static and dynamic analyses, code and document inspections, both informal and formal (design) reviews and other techniques.

For purposes of this guidance, a working definition of software validation is "establishing by objective evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements" [Ref: NIST 500-234]. Software validation is essentially a design verification function as defined in 21 CFR 820.3(aa) and 820.30(f), and includes all of the verification and testing activities conducted throughout the software life cycle. Design validation encompasses software validation, but goes further to check for proper operation of the software in its intended use environment. For example, both alpha and beta testing of device software in a simulated or real use environment, and acceptance testing of an automated testing tool by a device manufacturer, may be included as components of an overall design validation program for a software automated device.

The Quality System Regulation [21 CFR 820.3(k)] defines "establish" to mean "define, document and implement." Where it appears in this guidance, the word "establish" should be interpreted to have this same meaning.

Back to Table of Contents

IV. PRINCIPLES OF SOFTWARE VALIDATION

This section lists the general validation principles that FDA considers to be applicable to both the validation of medical device software and the validation of software used to design, develop or manufacture medical devices.

A. Timing

Proper software validation is not a one time event. Software validation should begin when design planning and design input begin. The software validation process should continue for the entire software life cycle. The software validation effort may pause with the release of each new version, but the software validation process does not end until the software product is no longer used.

B. Management

Software cannot be properly validated without an established software life cycle. Proper validation of software includes the planning, execution, analysis, and documentation of appropriate validation activities and tasks (including testing or other verification) throughout the entire software life cycle.

C. Plans

Established design and development plans should include a specific plan for how the software validation process will be controlled and executed.

D. Procedures

Validation tasks should be conducted in accordance with established procedures.

E. Requirements

To validate software, there must be predetermined and documented software requirements specifications [Ref: 21 CFR 820.3(z) and (aa) and 820.30(f) and (g)].

F. Testing

Software verification includes both static (paper review) and dynamic techniques. Dynamic analysis (i.e., testing) is concerned with demonstrating the software's run-time behavior in response to selected inputs and conditions. Due to the complexity of software, dynamic analysis alone may be insufficient to show that the software is correct, fully functional and free of avoidable defects. Therefore, static approaches and methods are used to offset this crucial limitation of dynamic analysis. Dynamic analysis is a necessary part of software verification, but static evaluation techniques such as inspections, analyses, walkthroughs, design reviews, etc., may be more effective in finding, correcting and preventing problems at an earlier stage of the development process. These static techniques can and should be used to focus or augment dynamic analysis.

Software test plans, test procedures and test cases should be developed as early in the software life cycle as possible. Such discipline helps to assure testability of requirements specifications and design specifications, and provides useful feedback regarding those specifications at a time when changes can be implemented most easily and cost effectively.

G. Partial Validation

Software cannot be partially validated. When a change is made to a software system (even a small change), the validation status of the entire software system should be addressed, and not just the validation of the individual change. Retrospective validation and reverse engineering of existing software is very difficult, but may be necessary in order to properly document and validate changes made to previously unvalidated software.

H. Amount of Effort

The magnitude of the software validation effort should be commensurate with the risk associated with the device, the device's dependence on software for potentially hazardous or critical functions, and the role of specific software modules in higher risk device functions. For example, while all software modules should be validated, those modules which are safety critical should be subject to more thorough and detailed inspections of software requirements specifications, software design specifications and test cases.

Likewise, size and complexity of the software project is an important factor in establishing the appropriate level of effort and associated documentation for the software. The larger the project and staff involved, the greater the need for formal communication, more extensive written procedures and management control of the process. However, small firm size is not a proper justification for inadequately staffing, controlling, or documenting the development and validation effort for a complex software project.

I. Independence

Validation activities should be conducted using the basic quality assurance precept of "independence of review." Self-validation is extremely difficult; an independent review is always better. Smaller firms may need to be creative in how tasks are organized and assigned, or may need to contract for some validation activities, in order to establish independence of review.

J. Real World

It is fully recognized that software is designed, developed, validated and regulated in a real world environment. It is understood that environments and risks cover a wide spectrum, so that each time a software validation principle is used, the implementation may be different.

Environments can include the application of software for the control of manufacturing, medical devices, design and development systems as well as quality systems. In each environment, software components from many sources may be used to create the application (e.g., in-house developed software, off-the-shelf software, contract software, shareware). In addition, software components come

in many different forms (e.g., application software, operating systems, compilers, debuggers, configuration management tools and many more).

The use of software in these environments is a complex task; therefore, it is appropriate that all of the software principles be considered when designing the software validation process. The resultant software validation process should be commensurate with the risk associated with the system, device or process.

Software validation activities and tasks may be dispersed, occurring at different locations and being conducted by different organizations. However, regardless of the distribution of tasks, contractual relations, source of components, or the development environment, the device manufacturer or specification developer retains ultimate responsibility for assuring that the software is validated.

Back to Table of Contents

V. SOFTWARE VALIDATION

The primary goal of software validation is to demonstrate that the completed software end product complies with established software and system requirements. The correctness and completeness of the system requirements should be addressed as part of the design validation process for the device. Software validation is the confirmation that all software requirements have been met and that all software requirements are traceable to the system requirements. Software validation is a required component of the design validation of a medical device. Whereas the software validation confirms that all software requirements have been met, the design validation goes further to confirm that the medical device itself meets user needs and intended uses.

A. Initial Validation Considerations

The concept phase of a project may begin when someone (e.g., marketing, field engineering, or quality assurance staff) suggests that a new automated function is needed for a medical device or a piece of manufacturing or quality system equipment, or that a software change is needed to correct a problem or enhance a software function. This suggestion is explored sufficiently to allow management to decide whether to authorize its full development or procurement. One or more staff may be assigned to consider and document the project, its purpose, anticipated users, intended use environments, system needs, and the anticipated role of software. The concept may include basic system elements, sequence of operations, constraints and risks associated with development of the software, and performance requirements for the system.

Once the concept is completed and documented, the results of the concept phase becomes an important design input for the software requirements specification. While not required by the Quality System Regulation, it is recommended that sources of input information developed during the concept phase be documented, for later reference when developing specific requirements. The concept may or may not be completed before proceeding to the requirements or preliminary design activities. However, by carefully and thoughtfully initiating the software project, it will be less likely that major requirements will be missed in later phases. For example, early decisions, such as whether to contract for software development, to use an off-the-shelf software product, or to develop software in-house, will have significant implications for later validation of that software.

The following is a list of preliminary questions and considerations:

- What software quality factors (e.g., reliability, maintainability, usability, etc.) are important to the validation process and how will those factors be evaluated?
- Are there enough staff and facilities resources to conduct the validation?
- What part will the hazard management function play in the software validation process?
- How will off-the-shelf (OTS) software be validated? What are the specific requirements for use

of the OTS software? What are the risks and benefits of OTS versus contracted or in-house developed software? What information (description of the software quality assurance program, validation techniques, documentation of validation, "bug lists", etc.) is available from the OTS vendor to help in validating use of the software in the device, or to produce the device? Will the OTS vendor allow an audit of their validation activities? Is the OTS software suitable for its intended use, given the availability of necessary validation information? What level of black-box testing is required to demonstrate that the OTS software is suitable for its intended use? What impact will these factors have on contract negotiations and vendor selection?

- How will contracted software be validated? In addition to the issues above for OTS software, who will control the source code and documentation, and what role will the contractor play in validation of the contracted software.

- For an OTS software automated process or quality system function, will the output of the process or function be fully verified in every case against specifications? If so, the process or function is not dependent upon proper operation of the software and "verification by output" may be sufficient. However, if the output will not be fully verified against the specification, then the software must be validated for its intended use [Ref: 21 CFR 820.70(i)].

- What human factors need to be addressed? One of the most persistent and critical problems encountered by FDA is user error induced by overly complex or counterintuitive design. Frequently, the design of the software is a factor in such user errors. Human factors engineering should be woven into the entire design and development process, including the device design concept, requirements, analyses, and tests. Safety and usability issues should be considered when developing flowcharts, state diagrams, prototyping tools, and test plans. Also task and function analyses, hazard analyses, prototype tests and reviews, and full usability tests should be performed. Participants from the user population should be included when applying these methodologies.

B. Software Life Cycle Activities

This guidance does not recommend the use of any specific life cycle model. Software developers should establish a software life cycle model that is appropriate for their product and organization. The life cycle model that is selected should cover the software from concept to retirement. Activities in a typical software life cycle include:

- Management
- Requirements
- Design
- Implementation (Coding)
- Integration and Test
- Installation
- Operation and Support
- Maintenance

C. Design Reviews

Formal design review is a primary validation tool. Formal design reviews allow management to confirm that all validation goals have been achieved. Software validation should be included in design reviews. The Quality System Regulation requires that at least one formal design review be conducted during the device design process. However, it is recommended that multiple design reviews be conducted (e.g., at the end of each software life cycle activity, in preparation for proceeding to the next activity). An especially important place for a design review is at the end of the requirements activity, before significant resources have been committed to a specific design. Problems found at this point can be more easily resolved, saving time and money, and reducing the likelihood of missing a critical issue. Some of the key questions to be answered during formal design reviews include:

- Have the appropriate validation tasks and expected results, outputs or products been established for each life cycle activity?

- Do the validation tasks and expected results, outputs or products of each life cycle activity comply with the requirements of other life cycle activities in terms of correctness, completeness, consistency, and accuracy?

- Do the validation tasks and expected results, outputs or products for each life cycle activity satisfy the standards, practices, and conventions of that activity?

- Do the validation tasks and expected results, outputs or products of each life cycle activity establish a proper basis for initiating tasks for the next life cycle activity?

D. Typical Validation Tasks

For each of the software life cycle activities, certain validation tasks are performed:

1. Management

During design and development planning, a software validation plan is created to identify necessary validation tasks, procedures for anomaly reporting and resolution, validation resources needed, and management review requirements including formal design reviews. A software life cycle model and associated activities should be identified, as well as validation tasks necessary for each software life cycle activity. The validation plan should include:

- The specific validation tasks for each life cycle activity.
- Methods and procedures for each validation task.
- Criteria for initiation and completion (acceptance) of each validation task.
- Inputs for each validation task.
- Outputs from each validation task.
- Criteria for defining and documenting outputs in terms that will allow evaluation of their conformance to input requirements.
- Roles, resources and responsibilities for each validation task.
- Risks and assumptions.

Management must identify and provide the appropriate validation environment and resources [Ref: 21 CFR 820.20(b)(1) and (2)]. Typically each validation task requires personnel as well as physical resources. The validation plan should identify the personnel, facility and equipment resources for each validation task. A configuration management plan should be developed that will guide and control multiple parallel development activities and assure proper communications and documentation. Controls should be implemented to assure positive and correct correspondence between all approved versions of the specifications documents, source code, object code and test suites which comprise a software system, accurate identification of the current approved versions, and appropriate access to the current approved versions during development activities.

Procedures should be created for reporting and resolving all software and validation anomalies. Management should identify the validation reports, and specify the contents, format, and responsible organizational elements for each report. Procedures should also be created for the review and approval of validation results, including the responsible organizational elements for such reviews and approvals.

Typical Validation Tasks - Management

2. Requirements

A software requirements specification document should be created with a written definition of the software functions to be performed. It is not possible to validate software without predetermined and documented software requirements. Typical software requirements specify the following:

- All inputs that the software system will receive.
- All outputs that the software system will produce..
- All functions that the software system will perform.
- All performance requirements that the software will meet, e.g., data throughput, reliability, timing, etc.
- The definition of all internal, external and user interfaces.
- What constitutes an error and how errors should be handled.
- The intended operating environment for the software, e.g., hardware platform, operating system, etc., (if this is a design constraint).
- All safety requirements, features or functions that will be implemented in software.
- All ranges, limits, defaults and specific values that the software will accept.

Safety requirements should be commensurate with the hazards that can result from a system failure. The identification of the system safety requirements should include a software hazard analysis (e.g., software failure mode, effects and criticality analysis (SFMECA) and/or software fault tree analysis (SFTA)). The failure modes of the software should be identified, as well as their consequences. This analysis should identify all possible system failures that could result from software failures. From this analysis, it should be possible to identify what measures need to be taken to prevent catastrophic and other failures.

The Quality System Regulation [21 CFR 820.30(c)] requires a mechanism for addressing incomplete, ambiguous, or conflicting requirements. Each software requirement documented in the software requirements specification should be evaluated for accuracy, completeness, consistency, testability, correctness, and clarity. For example, software requirements should be analyzed to verify that:

- There are no internal inconsistencies among requirements.
- All of the performance requirements for the system have been spelled out.
- Fault tolerance and security requirements are complete and correct.
- Allocation of software functions is accurate and complete.
- Software requirements are appropriate for the system hazards.
- All requirements are expressed in terms that are measurable.

Assertions are executable statements incorporated into the software as fault tolerance protection for system safety and computer security objectives. All assertions (e.g., checking algorithms, logic states, system integrity checks) and all responses to unfavorable results of the assertions should be checked to verify that the operation of the assertions will not adversely impact system performance or safety.

A software requirements traceability analysis should be conducted to trace software requirements to system requirements (and vice versa). In addition, a software requirements interface analysis should

be conducted, comparing the software requirements to hardware, user, operator and software interface requirements for accuracy, completeness, consistency, correctness, and clarity, and to assure that there are no external inconsistencies. In addition to any other analyses and documentation used to verify software requirements, one or more Formal Design Reviews (a.k.a. Formal Technical Reviews) should be conducted to confirm that requirements are fully specified and appropriate, before extensive software design efforts begin. Requirements can be approved and released incrementally, but care should be taken that interactions and interfaces among software (and hardware) requirements are properly reviewed, analyzed and controlled.

Typical Validation Tasks - Requirements

Preliminary Hazard Analysis
Traceability Analysis - System Requirements to Software Requirements (and vice versa)

Software Requirements Evaluation
Software Requirements Interface Analysis
System Test Plan Generation
Acceptance Test Plan Generation

3. Design

In the design phase, software requirements are translated into a logical and physical representation of the software to be implemented. To enable persons with varying levels of technical responsibilities to clearly understand design information, it may need to be presented both as a high level summary of the design, as well as a detailed design specification. The completed software design specification should constrain the programmer/coder to stay within the intent of the agreed upon requirements and design. The software design specification should be complete enough that the programmer is not required to make ad hoc design decisions.

The software design specification describes the software's logical structure, parameters to be measured or recorded, information flow, logical processing steps, control logic, data structures, error and alarm messages, security measures, and predetermined criteria for acceptance. It also describes any supporting software (e.g., operating systems, drivers, other applications), special hardware that will be needed, communication links among internal modules of the software, links with the supporting software, links with the hardware, and any other constraints not previously identified.

The software design specification may include:

data flow diagrams
program structure diagrams
control flow diagrams
pseudo code of the modules
context diagrams
interface/program diagrams
data and control element definitions
module definitions
module interaction diagrams

The validation activities that occur during this phase have several purposes. Software design evaluations are conducted to determine if the design is complete, correct, consistent, unambiguous, feasible and maintainable. Appropriate consideration of software architecture (e.g., modular structure) at the design phase can reduce the magnitude of future validation efforts when software changes are needed. Software design evaluations may include analyses of control flow, data flow, complexity, timing, sizing, memory allocation, and many other aspects of the design. A traceability analysis should be conducted to verify that the software design implements all of the software

requirements, and that all aspects of the design are traceable to those requirements. A design interface analysis should be conducted to evaluate the proposed design with respect to hardware, user, and related software requirements. The hazard analysis should be re-examined to determine whether any additional hazards have been identified and whether any new hazards have been introduced by the design.

At the end of the design activity, a Formal Design Review should be conducted to verify that the design is correct, consistent, complete, accurate, and testable, before moving to implement the design. Elements of the design can be approved and released incrementally, but care should be taken that interactions and interfaces among various elements are properly reviewed, analyzed and controlled.

Typical Validation Tasks - Design

Test Design Generation (module, integration, system and acceptance)
Updated Hazard Analysis
Traceability Analysis - Design Specification to Software Requirements (and vice versa)
Software Design Evaluation
Design Interface Analysis
Module Test Plan Generation
Integration Test Plan Generation

4. Implementation (Coding)

Implementation is the software activity where detailed design specifications are implemented as source code. This is referred to as programming or coding. Implementation is the lowest level of abstraction for the software development process and represents the last stage of the decomposition of the software requirements where module specifications are translated into a programming language. Implementation also represents the first stage of composition where the construction of the actual software code begins.

Programming usually involves the use of a high-level programming language and may also entail the use of assembly language or microcode for time-critical operations. The source code may be either compiled or interpreted for use on a target hardware platform. Decisions on selection of programming languages and software build tools (assemblers, linkers and compilers) may occur in the requirements or in the design phase of development. Such selection decisions should include consideration of the impact on subsequent validation tasks (e.g., availability of debugging and testing tools for the chosen language). Some compilers offer optional error checking commands and levels. For software validation, if the most rigorous level of error checking is not used for translation of the source code, then justification for use of the less rigorous translation error checking should be documented. Also, there should be documentation of the compilation process and its outcome, including any warnings or other messages from the compiler and their resolution. Modules ready for integration and test should have documentation of compliance with translation quality policies and procedures.

Source code should be evaluated to verify its compliance with specified coding standards. Such coding standards should include conventions for clarity, style, complexity management, and commenting. Code comments should provide useful and descriptive information for a module, including expected inputs and outputs, variables referenced, expected data types, and operations to be performed. Source code should also be evaluated to verify its compliance with the corresponding detailed design specifications. Source code evaluations are often implemented as code inspections and code walkthroughs. Small firms may employ desk checking with appropriate controls to assure consistency and independence. Source code evaluations should be extended to verification of internal interfaces between modules and layers [horizontal and vertical interfaces] and compliance with their design specifications. Appropriate documentation of the performance of source code evaluations should be maintained as part of the validation information.

During implementation, both static and dynamic, informal and formal, testing methods may be employed. The static methods include the code evaluations described above. When a source code module has passed the necessary static code evaluations, then dynamic analyses of the module begins.

Initially, dynamic testing may be informal as the programmer refines a module's code to conform to the specifications. However, because module testing is an essential element of software validation the firm's procedures should clearly define when module testing begins and who is to conduct module testing. If the programmer is responsible for module testing, then the procedures should clearly differentiate between the programmer's informal testing conducted to implement the module, and the dynamic testing conducted as part of validation.

A source code traceability analysis is an important tool for verifying that all code is linked to established specifications and established test procedures. A source code traceability analysis should be conducted and documented to:

- verify that each element of the software design specification has been implemented in code
- verify that modules and functions implemented in code can be traced back to an element in the software design specification
- verify that tests for modules and functions can be traced back to an element in the software design specification
- verify that tests for modules and functions can be traced to source code for the same modules and functions

Typical Validation Tasks - *Implementation*

Traceability Analyses
    Source Code to Design Specifications and Vice Versa
    Test Cases to Source Code and to Design Specifications
Source Code and Source Code Documentation
Source Code Interface Analysis
Test Procedure and Test Case Generation (module, integration, system and acceptance)

5. Integration and Test

This aspect of software validation is closely coupled with the prior validation activities. In fact, effective testing activities overlap with other software development activities. Software testing objectives include demonstration of compliance with all software specifications, and producing evidence which provides confidence that defects which may lead to unacceptable failure conditions have been identified and removed. A software testing strategy designed to find software defects will produce far different results than a strategy designed to prove that the software works correctly. A complete software testing program uses both strategies to accomplish these objectives.

As programming of the individual modules comprising a software system is completed, the modules are combined until the complete program has been assembled. This process is called integration and proceeds according to the project's integration plan. The methods which may be used range from non-incremental integration to any of those employed for incremental integration. Non-incremental integration is often used for small programs, while some form of incremental integration is typically employed for large programs. The properties of the program being assembled dictate the chosen method of integration.

PLANS: Test plans should be created during the prior software development phases. They should identify the test schedules, environments, resources [personnel, tools, etc.], methodologies, cases [inputs, procedures, outputs, expected results], documentation and reporting criteria. The test plans should be linked to each of the specification phases; e.g., requirements, design, and implementation; and correlated to the other project plans including the integration plan. Individual test cases should be definitively associated with particular specification elements and each test case should include a predetermined, explicit, and measurable expected result, derived from the specification documents, in order to identify objective success/failure criteria.

Test plans should identify the necessary levels and extent of testing, as well as clear, pre-determined acceptance criteria. The magnitude of testing should be linked to criticality, reliability, and/or safety

issues. Test plans should also include criteria for determining when testing is complete. Test completion criteria should include both functional and structural coverage requirements. Each externally visible function and each internal function should be tested at least once. Each program statement should be executed at least once and each program decision should be exercised with both true and false outcomes at least once. Test completion criteria should also include measurements or estimates of the quality and/or reliability of the released software.

PERSONNEL: Sufficient personnel should be available to provide necessary independence from the programming staff, and to provide adequate knowledge of both the software application's subject matter and software/programming concerns related to testing. Small firms may use various techniques (e.g., detailed written procedures and checklists) to facilitate consistent application of intended testing activities and to simulate independence.

METHODS: The methodologies used to identify test cases should provide for thorough and rigorous examination of the software. They should challenge the intended use or functionality of a program with test cases based on the functional and performance specifications. They should also challenge the decisions made by the program with test cases based on the structure or logic of the design and source code.

Complete structural testing exercises the program's data structures; e.g., configuration tables; and its control and procedural logic at the appropriate level, and it identifies "dead" code. It assures the program's statements and decisions are fully exercised by testing; for example, confirming that program loop constructs behave as expected at their boundaries. For configurable software, the integrity of the data in configuration tables should be evaluated for its impact on program behavior. Structural testing should be done at the module, integration, and system levels of testing.

Complete functional testing includes test cases which expose program behavior not only in response to the normal case, but also in response to exceptional, stress and/or worst case conditions. As applicable, it demonstrates program behavior at the boundaries of its input and output domains; program responses to invalid, unexpected, and special inputs are confirmed; the program's actions are revealed when given combinations of inputs, unexpected sequences of inputs, or when defined timing requirements are violated. Functional test cases should be identified for application at the module, integration, and system levels of testing.

Module (a.k.a. unit or component) level testing focuses on the early examination of sub-program functionality and ensures that functionality not visible at the system level is examined by testing. Module testing should be conducted before the integration testing phase to confirm that modules meet specifications and ensure that quality modules are furnished for integration into the finished software system. The magnitude of the testing effort should be commensurate with the criticality (risk), complexity, or error prone status associated with a module. For low hazard modules, the static analyses (testing) conducted during implementation may be sufficient. However, dynamic testing (executing the code with test drivers) is usually necessary to verify the correct behavior of critical modules. For example, critical modules may be exposed to fault injection testing or other stress testing in order to characterize the behavior of the module with out-of-range or unexpected input conditions or parameters.

Integration level testing focuses on the transfer of data and control across a program's internal and external interfaces. External interfaces are those with other software including operating system software, system hardware, and the users. When a program is built using incremental integration methods, sufficient regression testing should be conducted to assure the addition of new modules has not changed the behavior of existing modules.

System level testing demonstrates that all specified functionality exists and that the software is trustworthy. This testing verifies the as-built program's functionality and performance with respect to the requirements for the software system and for the device. Test cases designed to address concerns such as robustness, stress, security, recovery, usability, etc., should be used to verify the dependability of the software. Control measures, e.g., a traceability analysis, should be used to assure that the necessary level of coverage is achieved.

System level testing also exhibits the software's behavior in the intended operating environment, and is an important aspect of design validation for the device. The location of such testing is dependent upon the software developer's ability to produce the target operating environment. Depending upon the circumstances, simulation and/or testing at (potential) customer locations may be utilized. Test plans should identify the controls to assure that the intended coverage is achieved and that proper documentation is prepared when planned system testing is conducted at sites not directly controlled by the software developer.

The quality of any software tools used during testing should be such that they do not introduce errors into the application being developed. For tools designed to find software errors, there should be evidence that they perform as intended. These tools can include supporting software built in-house to facilitate module testing and subsequent integration testing (e.g., drivers and stubs) and commercial testing tools. Appropriate documentation providing evidence of the quality of these tools should be maintained.

DOCUMENTATION: Test results [inputs, processing, outputs] should be documented in a mannerpermitting objective pass/fail decisions to be reached. They should be suitable for review and decision making subsequent to running the test. Test results should also be suitable for use in any subsequent regression testing. Errors detected during testing should be logged, classified, reviewed and resolved prior to release of the software. Test reports should comply with the requirements of the corresponding test plans.

Typical Validation Tasks - *Integration and Testing*

Traceability Analysis - Testing
- Module Tests to Detailed Design
- Integration Tests to High Level Design
- System Tests to Software Requirements

Test Evaluation
Error Evaluation/Resolution
Module Test Execution
Integration Test Execution
System Test Execution
Acceptance Test Execution
Test Results Evaluation
Final Test Report

Back to Table of Contents

VI. INSTALLATION TESTING

Installation testing is an essential part of validation for a device or the validation of an automated process used in its design, manufacture, or implementation of its quality system.

Section 820.170 of the Quality System Regulation requires installation and inspection procedures (including testing where appropriate) and documentation of inspection and testing to demonstrate proper installation. Likewise, Section 820.70(g) requires that manufacturing equipment must meet specified requirements, and Section 820.70(i) requires that automated systems be validated for their intended use.

Terminology in this testing area can be confusing. Terms such as beta test, site validation, user acceptance test, and installation verification have all been used to describe installation testing. To avoid confusion, and for the purposes of this guidance, installation testing is defined as any testing that takes place outside of the developer's controlled environment. Installation testing is any testing that takes place at a user's site with the actual hardware and software that will be part of the

installed system configuration. The testing is accomplished through either actual or simulated use of the software being tested within the environment in which it is intended to function.

Guidance contained here is general in nature and is applicable to any installation testing. However, in some areas, i.e. bloodbank systems, there are specific site validation requirements that need to be considered in the planning of installation testing. Test planners should check with individual FDA Centers to determine whether there are any additional regulatory requirements for installation testing.

Installation testing should follow a pre-defined plan with a formal summary of testing and a record of formal acceptance. There should be retention of documented evidence of all testing procedures, test input data and test results.

There should be evidence that hardware and software are installed and configured as specified. Measures should ensure that all system components are exercised during the testing and that the versions of these components are those specified. The testing instructions should encourage use through the full range of operating conditions and should continue for a sufficient time to allow the system to encounter a wide spectrum of conditions and events in an effort to detect any latent faults which are not apparent during more normal activities.

Some of the evaluations that have been performed earlier by the software developer at the developer's site should be repeated at the site of actual use. These may include tests for a high volume of data, heavy loads or stresses, security, fault testing (avoidance, detection, tolerance, and recovery), error messages, implementation of safety requirements, and serviceability. The developer may be able to furnish the user with some of the test data sets to be used for this purpose.

In addition to an evaluation of the system's ability to properly perform its intended functions, there should be an evaluation of the ability of the users of the system to understand and correctly interface with it. Operators should be able to perform the intended operations and respond in an appropriate and timely manner to all alarms, warnings, errors, etc.

Records should be maintained during installation testing of both the system's capability to properly perform and the system's failures, if any, which are encountered. The revision of the system to compensate for faults detected during this installation testing should follow the same procedures and controls as any other software change.

The developers of the software may or may not be involved in the installation testing. If the developers are involved, they may seamlessly carry over to the user's site the last portions of design-level systems testing. If the developers are not involved, it is all the more important that the user have persons knowledgeable in software engineering who understand the importance of such matters as careful test planning, the definition of expected test results, and the recording of all test outputs.

Back to Table of Contents

VII. MAINTENANCE AND SOFTWARE CHANGES

As applied to software, the term maintenance does not mean the same as when applied to hardware. The operational maintenance of hardware and software are different because their failure/error mechanisms are different. Hardware maintenance typically includes preventive maintenance actions, component replacement, and corrective changes. Software maintenance includes corrective, perfective and adaptive changes, but does not include preventive maintenance actions or component replacement.

Changes made to correct errors and faults in the software are considered as corrective maintenance. Changes made to the software to improve the performance, maintainability, or other attribute of the software system is considered as perfective maintenance. Changes made to the software to make the software system usable in a changed environment is considered as adaptive maintenance.

All modifications, enhancements, or additions to existing software or its operating environment (corrective, perfective, or adaptive changes) are design changes, and are subject to design controls provisions of the Quality System Regulation. The regulation [21 CFR 820.30(i)] requires design validation, unless the manufacturer documents the justification of why design verification is sufficient. The validation requirements, activities and processes that are used for software maintenance actions are the same as those that are used for the development of new software. The validation activities associated with each software change should be documented as part of the record of that change.

When changes are made to a software system, either during initial development or during post release maintenance, sufficient regression testing should be conducted to demonstrate that portions of the software not involved in the change were not adversely impacted. This is in addition to testing which evaluates the correctness of the implemented change(s).

The specific validation effort necessary for each software change is determined by the type of change, the development products affected and the impact of those products on the operation of the software. Careful and complete documentation of the design structure and interrelationships of various modules, interfaces, etc., can limit the validation effort needed when a change is made. The level of effort needed for fully validate a change is also dependent upon the degree to which validation of the original software was documented and archived. For example, to perform regression testing, the test documentation, test cases and results of previous validation testing should have been archived. Failure to archive this information for later use can sharply increase the level of effort and expense of revalidating the software after a change is made.

In addition to validation tasks that are part of the standard software development process, some additional maintenance tasks include the following:

Software Validation Plan Revision - For software that was previously validated, the existing software validation plan should be revised to support the validation of the revised software. If no previous software validation plan exists, a software validation plan should be established to support the validation of the revised software.

Anomaly Evaluation - Software anomalies should be evaluated in terms of their severity and their effects on system operation and safety. Anomalies should also be evaluated as symptoms of deficiencies in the quality system. A root cause analysis of anomalies can identify specific quality system deficiencies. Where trends are identified, appropriate corrective and preventive actions, as required in Section 820.100 of the Quality System Regulation, must be implemented to avoid recurrence of similar quality problems.

Proposed Change Assessment - All proposed modifications, enhancements, or additions should be assessed to determine the effect each change would have on the system. This information should determine the extent to which validation tasks need to be iterated.

Validation Task Iteration - For approved software changes, all necessary validation tasks should be performed to ensure that planned changes are implemented correctly, all documentation is complete and up to date, and no unacceptable changes have occurred in software performance.

Back to Table of Contents

Bibliography

Food and Drug Administration References

Design Control Guidance for Medical Device Manufacturers, March 1997

Do It by Design, An Introduction to Human Factors in Medical Devices, March 1997

Glossary of Computerized System and Software Development Terminology, August 1995

Guideline on the General Principles of Process Validation, May 1987.

Medical Devices; Current Good Manufacturing Practice (CGMP) Final Rule; Quality System Regulation, 61 Federal Register 52602 (October 7, 1996).

Reviewer Guidance for Computer Controlled Medical Devices Undergoing 510(k) Review, August 1991 (currently under revision)

Reviewer Guidance for a Pre-Market Notification Submission for Blood Establishment Computer Software, January 1997

Software Development Activities, Reference Materials and Training Aids for Investigators, July 1987.

Other Government References

NIST Special Publication 500-165, Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards, National Institute for Science and Technology.

NIST Special Publication 500-234, Reference Information for the Software Verification and Validation Process, National Institute for Science and Technology, March 1996.

The Application of the Principles of GLP to Computerized Systems, Environmental Monograph #116, Organization for Economic Cooperation and Development (OECD), 1995.

Standards

ANSI/ANS-10.4-1987, Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry, American National Standards Institute, 1987.

ANSI/ASQC Standard D1160-1995, Formal Design Reviews, American Society for Quality Control, 1995.

IEEE Std 1012-1986, Software Verification and Validation Plans, Institute for Electrical and Electronics Engineers, 1986.

ANSI/ISO/ASQC Q9000-3-1991, Guidelines for the Application of ANSI/ISO/ASQC Q9001 to the Development, Supply and Maintenance of Software, American Society for Quality Control, 1991.

ISO/IEC 12207, Information Technology - Software Life Cycle Processes.

Other References

Technical Report No. 18, Validation of Computer Related Systems, PDA Journal of Pharmaceutical Science and Technology, Vol. 49, No 1/January-February, 1995.

Software Systems Engineering, A. P. Sage, J.D. Palmer: John Wiley & Sons, 1990.

Software Verification and Validation, Realistic Project Approaches, M. S. Deutsch, Prentice Hall, 1982.

The Art of Software Testing, Glenford J. Myers, John Wiley & Sons, Inc., 1979

Validation Compliance Annual 1995, International Validation Forum, Inc.

Back to Table of Contents

Development Team

The following team was assembled to prepare this DRAFT for public comment.

Center for Devices and Radiological Health

Office of Compliance	Stewart Crumpler, Howard Press
Office of Device Evaluation	Donna-Bea Tillman, James Cheng
Office of Science and Technology	John Murray
Office of Health and Industry Programs	Dick Sawyer, Bryan Benesch
Office of Surveillance and Biometrics	Isaac Hantman

Center for Drug Evaluation and Research

Office of Compliance	Charles Snipes

Center for Biologics Evaluation and Branch

Office of Compliance	Alice Godziemski

Office of Regulatory Affairs

Office of Regional Operations	David Bergeson, Joan Loreng

Back to Table of Contents

(Date June 9, 1997)