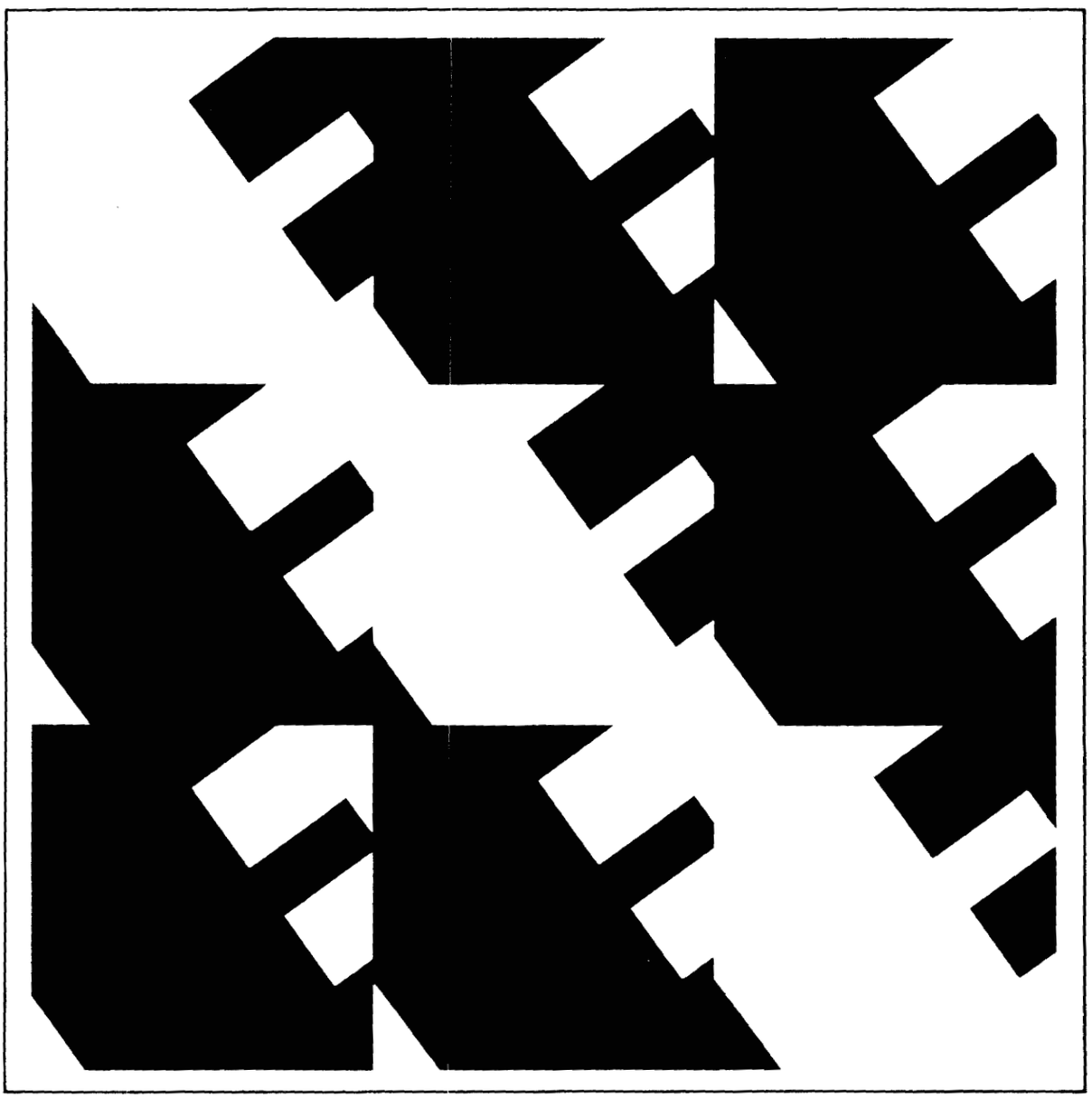
AMERICAN NATIONAL STANDARD

ANSI/IEEE Std 830-1984

IEEE Guide to Software Requirements Specifications



ANSI/IEEE Std 830-1984

Published by The Institute of Electrical and Electronics Engineers, Inc 345 East 47th Street, New York, NY 10017, USA

February 10, 1984 *SH08714*

THIS PAGE WAS
BLANK IN THE ORIGINAL

ANSI/IEEE
Std 830-1984

An American National Standard

IEEE Guide to Software Requirements Specifications

Sponsor

**Software Engineering Technical Committee
of the
IEEE Computer Society**

Approved September 30, 1983

IEEE Standards Board

Approved July 20, 1984

American National Standards Institute

**The Institute of Electrical and Electronics Engineers, Inc
345 East 47th Street, New York, NY 10017, USA**

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least once every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
345 East 47th Street
New York, NY 10017
USA

Foreword

(This Foreword is not a part of IEEE Std 830-1984, IEEE Guide to Software Requirements Specifications.)

This guide describes alternate approaches to good practice in the specification of software requirements. The requirements may be explicitly stated by the user or they may be allocated to computer software (that is, programs) by the system requirements analysis process. This guide does not suggest that a hierarchy of software requirements specifications exists, of which each, in turn, defines a smaller subset of requirements.

As a guide, this document should help:

(1) Software customers to accurately describe what they wish to obtain.

(2) Software suppliers to understand exactly what the customer wants.

(3) Individuals to accomplish the following goals:

(a) Develop standard software requirements specifications (SRS) outline for their own organizations.

(b) Define the form and content of their specific software requirements specifications.

(c) Develop additional local supporting items such as an SRS quality checklist, or an SRS writer's handbook.

To the customers, suppliers and other individuals, a good SRS provides several specific benefits. It will accomplish the following goals:

(1) Establish the basis for agreement between the customers and the suppliers on what the software product is to do. The complete description of the functions to be performed by the software specified in the SRS will assist the potential user, to determine if the software specified meets their needs or how the software must be modified to meet their needs.

(2) Reduce the development effort. The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

(3) Provide a basis for estimating costs and schedules. The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates. The SRS also provides a clear description of the required software and makes it easier to estimate and plan the necessary resources. The requirements which, together with a development plan, can be used to measure progress.

(4) Provide a baseline for validation and verification. Organizations can develop their validation and verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured. (However, that the converse is not true; a standard legal contract cannot be used as an SRS. Such documents rarely contain the detail required and are often incomplete.)

(5) Facilitate transfer. The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.

(6) Serves as a basis for enhancement. Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a solid foundation for continued production evolution.

This guide is based on a model in which the result of the software requirements specification process is an unambiguous and complete specification document. In principle, the SRS can be mechanically translated into the specified software program directly. As such, the resulting SRS document itself is the specified software, and the supplier's only duty (after completing the SRS) would be the mechanical compilation of the SRS into machine code for the target computer. The present state of the art does not support such a compiler with an optimizer of such efficiency to make it practical but this limitation need not, and should not, restrict the intermediate objective of an unambiguous SRS.

This guide is consistent with IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology; ANSI/IEEE Std 730-1981, IEEE Standard for Software Quality Assurance Plans; and IEEE Std 829-1983, IEEE Standard for Software Test Documentation. This guide may be used in conjunction with those standards or separately.

This guide was prepared by the Software Requirements Working Group of the Software Engineering Standards Subcommittee of the Technical Committee on Software Engineering of the IEEE Computer Society.

At the time the guide was approved, the Software Requirements Working Group had the following membership:

A. M. Davis, *Chairperson*

M. Bariff
H. Berlack
F. Buckley
E. Byrne
F. Calm
K. Foster
S. Frankel
T. L. Hannan
P. W. Kendra
T. M. Kurinara
R. A. C. Lane
R. Lechner
E. Levinson
P. Lindemann
S. Mroczek
A. J. Neumann
W. Newson
D. Paster
B. Pope
P. B. Powell
G. R. Niedhart
J. Russell
A. Salwin
N. B. Schneidewind
D. Schultz
R. W. Szczech
A. Weigel
P. Willis
H. Willman
A. W. Yonda

At the time that it approved this guide, the Software Engineering Standards Subcommittee had the following membership:

F. J. Buckley, *Chairperson*

R. J. Abbott
A. F. Ackerman
L. Beltracchi
D. W. Bragg
D. W. Burt
E. R. Byrne
H. Carney
J. W. Center
A. M. Cicu
G. G. Cooke
A. J. Cote, Jr
P. W. Daggett
G. Darling
B. Dasarathy
A. M. Davis
P. A. Denny
J. A. Dobbins
M. L. Eads
J. D. Earls
L. G. Egan, Jr
D. W. Fife
J. Flournoy
J. J. Forman
F. K. Gardner
D. Gelperin
E. L. Gibbs
G. Gladden
S. A. Gloss-Soler
J. W. Grigsby
R. M. Gross
D. A. Gustafson
R. T. Gustin
T. L. Hannan
H. Hecht
L. R. Heselton, III
S. Horvitz
P. Howley
R. N. Hurwitz
S. Hwang
J. H. Ingram
J. P. Kalasky
R. Kessler
T. M. Kurinara
D. V. LaRosa
R. A. C. Lane
G. R. Lewis
F. C. Lim
G. S. Lindsay
M. Lipow
W. M. Lively
M. Lubofsky
D. Lindquist
A. K. Manhindru
P. C. Marriott
C. F. Martiny
M. McCollough
B. Menkus
B. Meyer
E. F. Miller, Jr
G. S. Morris
G. T. Morun
W. G. Murch
J. Nebb
G. R. Niedhart
M. A. Neighbors
J. O. Neilson
D. J. Ostrom
D. J. Pfeiffer
R. M. Poston
P. B. Powell
J. W. Radatz
J. C. Rault
S. T. Redwine
L. K. Reed
W. E. Riddle
C. W. Rutter, III
P. E. Schilling
N. F. Schniedewind
A. D. Schuman
L. W. Seagren
R. L. Skelton
W. Smith
H. M. Sneed
K. C. Tai
B. J. Taute
R. H. Thayer
G. D. Tice, Jr
T. L. Tillmans
W. S. Turner, III
E. A. Ulbrich, Jr
D. Usechak
U. Voges
R. Wachter
J. P. Walter
D. Webdale
A. H. Weigel
N. P. Wilburn
W. M. Wong
T. Workman
A. W. Yonda
P. F. Zoll

Special representatives to the software engineering standards subcommittee were:

J. Milandin:	ANSI Z1
W. G. Perry:	Data Processing Manufacturers Association
R. Pritchett:	EDP Auditors Association
T. L. Regulinski:	IEEE Reliability Society
N. C. Farr:	Nuclear Power Engineering Committee, IEEE Power Engineering Society

Suggestions for improvement of this guide are welcome. They should be provided to:
The Secretary
IEEE Standards Board
345 East 47th St
New York, New York 10017

At the time the IEEE Standards Board approved this standard on September 20, 1983 it had the following members:

James H. Beall, *Chairman* **Edward Chelotti,** *Vice Chairman*

Sava I. Sherr, *Secretary*

J. J. Archambault
John T. Boettger
J. V. Bonucchi
Rene Castenschiold
Edward J. Cohen
Len S. Corey
Donald C. Fleckenstein
Jay Forster
Donald H. Heirman
Irvin N. Howell
Joseph L. Koepfinger*
Irving Kolodny
George Konomos
John E. May
Donald T. Michael*
John P. Riganati
Frank L. Rose
Robert W. Seelbach
Jay A. Stewart
Clifford O. Swanson
Robert E. Weiler
W. B. Wilkens
Charles J. Wylie

*Member emeritus

THIS PAGE WAS
BLANK IN THE ORIGINAL

Contents

SECTION PAGE

FIGURES

TABLES

IEEE Guide to Software Requirements Specifications

1. Scope and Organization

1.1 Scope. This is a guide for writing software requirements specifications. It describes the necessary content and qualities of a good Software Requirements Specification (SRS) and presents a prototype SRS outline.

This guide does not specify industry-wide SRS standards nor state mandatory SRS requirements. This guide is written under the premise that the current state of the art does not warrant or support such a formal standards document.

This guide is applicable to in-house and commercial software products. Special care, however, should be used in its application because:

(1) This guide is aimed at specifying requirements of software to be developed. Application of this material to already-developed software is counter-productive.

(2) This guide does not cover the specification of requirements for software being developed using the techniques of rapid prototyping.

1.2 Organization. The remainder of this guide is organized as follows:

(1) Section 2 provides the references used throughout the guide.

(2) Section 3 provides definitions of specific terms used throughout the guide.

(3) Section 4 provides background information for writing a good SRS.

(4) Section 5 provides specific guidance for expressing software requirements.

(5) Section 6 discusses each of the essential parts of an SRS and provides alternate prototype outlines.

2. References

[1] ANSI/IEEE Std 100-1977, IEEE Standard Dictionary of Electrical and Electronics Terms.

[2] ANSI/IEEE Std 730-1981, IEEE Standard for Software Quality Assurance Plans.

[3] ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology.

[4] BRUSAW, C. T., ALRED, G. and OLIU, W., *Handbook of Technical Writing*, New York, St. Martin's Press, 1976.

[5] DASARATNY, B., *Timing Constraints of Real-Time Systems: Constructs for Expressing Them*, IEEE Real-Time Systems Symposium, Dec 1982.

[6] DAVIS, A., The Design of a Family of Applications-Oriented Requirements Languages, *IEEE Computer*, 15, 5 May 1982, pp 21–28.

[7] FREEDMAN, D. and WEINBERG, G., Handbook of Walkthroughs, *Inspections and Technical Reviews*, 3rd Ed, Little and Brown Publishers, New York.

[8] KAIN, R., Automata Theory: Machines and Languages, McGraw Hill, New York, 1972.

[9] KOHAVI, Z., Switching and Finite Automata Theory, McGraw Hill, New York, 1970.

[10] KRAMER, J., Editor, Application Oriented Specifications Glossary of Terms, *European Workshop on Industrial Computer Systems (EWICS)*, Imperial College, London, England, May 6, 1981.[1]

[1] Copies of this document are available from EWICS, c/o G. R. Koch, BIOMATIK Gmbh, Carl-Mez Str 81–83, D-7800 Freiburg, Federal Republic of Germany.

[11] MILLS, G., and WALTER, J., *Technical Writing*, New York, Holt, Rinehart and Winston, 4th Ed, 1978.

[12] PETERSON, J., *Petri Nets*, ACM Computing Surveys, 9, 4, Dec 1977, pp 223–252.

[13] RAMAMOORHY, C. and SO, H. H., *Software Requirements and Specifications: Status and Perspectives*, Tutorial: Software Methodology, RAMAMOORTHY, C. and YEH, R. T., Editors. IEEE Catalog no EHO 142-0, 1978, pp 43–164.

[14] TAGGART, W. M. Jr, and THARP, M. O., *A Survey of Information Requirements Analysis Techniques*, ACM Computing Surveys, 9, 4, Dec 1977, pp 273–290.

[15] TEICHROEW, D., *A Survey of Languages for Stating Requirements for Computer-Based Information Systems*, 1972 Fall Joint Computer Conference, 1972, pp 1203–1224.

3. Definitions

Except for the definitions listed below, the definitions of all terms used in this guide conform to the definitions provided in IEEE Std 729-1983 [3][2], for example, the terms requirement, requirements specification. If a term used in this guide does not appear in that Standard, then ANSI/IEEE Std 100-1977 [1], applies.

The terms listed in this section have been adopted from Section 2, [10].

contract. A legally binding document agreed upon by the customer and supplier. This includes the technical, organizational, cost and schedule requirements of a product.

customer. The person, or persons, who pay for the product and usually (but not necessarily) decides the requirements. In the context of this document the customer and the supplier may be members of the same organization.

language. A means of communication, with syntax and semantics, consisting of a set of representations, conventions and associated rules used to convey information.

partitioning. Decomposition; the separation of the whole into its parts.

supplier. The person, or persons, who produce a product for a customer. In the context of this document, the customer and the supplier may be members of the same organization.

user. The person, or persons, who operate or interact directly with the system. The user(s) and the customer(s) are often not the same person(s).

4. Background Information for Writing a Good SRS

This section provides background information necessary for writing an SRS. This includes:

(1) Examination of the nature of the SRS

(2) Environmental considerations surrounding the SRS

(3) Characteristics required for a good SRS

(4) Recommendations for joint preparation of an SRS

(5) Evolutionary aspects of the SRS

(6) The use of automated tools to develop an SRS

4.1 The SRS. The SRS is a specification for a particular software product, program, or set of programs that does certain things. See ANSI/IEEE Std 730-1981 [2], 3.4.2.1.

The description places two basic requirements on the SRS:

(1) It must say certain things. For example, software developed from an SRS that fails to specify that error messages will be provided, will probably fail to satisfy the customer.

(2) It must say those things in certain ways. For example, software developed from an SRS that fails to specify the format and content of error messages and instead is developed from a vague and non-quantifiable requirement such as *All error messages will be helpful*, will probably be unsatisfactory. What is *helpful* for one person can be a severe aggravation to another person.

For recommended contents of an SRS see Section 6.

4.2 Environment of the SRS. It is important to consider the part that the SRS plays in the total software project. The provisions in ANSI/IEEE Std 730-1981 [2], define the minimum required documents for a software project. See [2], 3.4.2.

ANSI/IEEE Std 730-1981 [2] also identifies the other useful documents. See [2], 3.4.3.

[2]Numbers in brackets correspond to those of the references in Section 2.

Since the SRS has a definite role to play in this documentation scheme, SRS writers should be careful not to go beyond the bounds of that role. This means the following requirements should be met:

(1) The SRS must correctly define all of the software requirements, but no more.

(2) The SRS should not describe any design, verification, or project management details, except for required design constraints.

Such a properly written SRS limits the range of valid solutions but does not specify any particular design and thus provides the supplier with maximum flexibility.

4.3 Characteristics of A Good SRS. The previous sections describe the types of information that should be contained in an SRS. The following concepts deal with particular characteristics. A good SRS is:

(1) Unambiguous
(2) Complete
(3) Verifiable
(4) Consistent
(5) Modifiable
(6) Traceable
(7) Usable during the Operation and Maintenance Phase

4.3.1 Unambiguous. An SRS is unambiguous if — and only if — every requirement stated therein has only one interpretation.

(1) As a minimum, this requires that each characteristic of the final product be described using a single unique term.

(2) In cases where a term used in a particular context could have multiple meanings, the term must be included in a glossary where its meaning is made more specific.

4.3.1.1 Natural Language Pitfalls. Requirements are often written in a natural language (for example, English). SRS writers who use a natural language must be especially careful to review their requirements for ambiguity. The following examples are taken from Section 2, [7].

(1) The specification *The data set will contain an end of file character*, might be read as:

(a) There will be one and only one end of file character

(b) Some character will be designated as an end of file character

(c) There will be at least one end of file character

(2) The specification *The control total is taken from the last record*, might be read as:

(a) The control total is taken from the record at the end of the file

(b) The control total is taken from the latest record

(c) The control total is taken from the previous record

(3) The specification *All customers have the same control field*, might be read as:

(a) All customers have the same value in their control field

(b) All customer control fields have the same format

(c) One control field is issued for all customers

(4) The specification *All files are controlled by a file control block*, might be read as:

(a) One control block controls the entire set of files

(b) Each file has its own block

(c) Each file is controlled by a control block, but one control block might control more than one file

4.3.1.2 Formal Requirements Specifications Languages. One way to avoid the ambiguity inherent in natural language is to write the SRS in a formal requirements specification language.[3]

(1) One major advantage in the use of such languages is the reduction of ambiguity. This occurs, in part, because the formal language processors automatically detect many lexical, syntactic, and semantic errors.

(2) One major disadvantage in the use of such languages is the length of time required to learn them.

4.3.2 Complete. An SRS is complete if it possesses the following qualities:

(1) Inclusion of all significant requirements, whether relating to functionality, performance, design constraints, attributes or external interfaces.

(2) Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to valid and invalid input values.

(3) Conformity to any SRS standard that applies to it. If a particular section of the standard is not applicable, the SRS should

[3]For detailed discussion on this topic, suggested readings are [6], [13], [14], and [15].

include the section number and an explanation of why it is not applicable.

(4) Full labeling and referencing of all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

4.3.2.1 Use of TBDs. Any SRS that uses the phrase *to be determined* (TBD) is not a complete SRS.

(1) The TBD is, however, occasionally necessary and should be accompanied by:

(a) A description of the conditions causing the TBD (for example, why an answer is not known) so that the situation can be resolved.

(b) A description of what must be done to eliminate the TBD.

(2) Any project documents that are based on an SRS that contains TBDs, should:

(a) Identify the version or state the specific release number of the SRS associated with that particular document.

(b) Exclude any commitments dependent upon the sections of the SRS that are still identified as TBDs.

4.3.3 Verifiable. An SRS is verifiable if and only if every requirement stated therein is verifiable. A requirement is verifiable if and only if there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement.

(1) Examples of nonverifiable requirements include statements such as:

(a) *The product should work well,* or *The product should have a good human interface.* These requirements cannot be verified because it is impossible to define the terms *good* or *well.*

(b) *The program shall never enter an infinite loop.* This requirement is non-verifiable because the testing of this quality is theoretically impossible.

(c) *The output of the program shall usually be given within 10 s.* This requirement is nonverifiable because the term *usually* cannot be measured.

(2) An example of a verifiable statement is *The output of the program shall be given within 20 s of event* X, 60% *of the time; and shall be given within 30 s of event* X, 100% *of the time.* This statement can be verified because it uses concrete terms and measurable quantities.

(3) If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.

(4) If a requirement is not expressible in verifiable terms at the time the SRS is prepared, then a point in the development cycle (review, test plan issue, etc) should be identified at which the requirement must be put into a verifiable form.

4.3.4 Consistent. An SRS is consistent if and only if no set of individual requirements described in it conflict. There are three types of likely conflicts in an SRS:

(1) Two or more requirements might describe the same real world object but use different terms for that object. For example, a program's request for a user input might be called a *prompt* in one requirement and a *cue* in another.

(2) The specified characteristics of real world objects might conflict. For example:

(a) The format of an output report might be described in one requirement as *tabular* but in another as *textual.*

(b) One requirement might state that all lights shall be green while another states that all lights shall be blue.

(3) There might be a logical or temporal conflict between two specified actions. For example:

(a) One requirement might specify that the program will add two inputs and another specify that the program will multiply them.

(b) One requirement might state that *A* must always follow *B,* while another requires that *A and B* occur simultaneously.

4.3.5 Modifiable. An SRS is modifiable if its structure and style are such that any necessary changes to the requirements can be made easily, completely, and consistently. Modifiability generally requires an SRS to:

(1) Have a coherent and easy-to-use organization, with a table of contents, an index, and explicit cross-referencing.

(2) Not be redundant; that is, the same requirement should not appear in more than one place in the SRS.

(a) Redundancy itself is not an error, but it can easily lead to errors. Redundancy can occasionally help to make an SRS more readable, but a problem can arise when the redundant document is updated. Assume, for instance, that a certain requirement is stated in two places. At some later time, it is determined that the requirement should be altered, but the change is made in only one of the two locations. The SRS then becomes inconsistent.

(b) Whenever redundancy is necessary, the SRS should include explicit cross-references to make it modifiable.

4.3.6 Traceable. An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended:

(1) Backward traceability (that is, to previous stages of development) depends upon each requirement explicitly referencing its source in previous documents.

(2) Forward traceability (that is, to all documents spawned by the SRS) depends upon each requirement in the SRS having a unique name or reference number.

When a requirement in the SRS represents an apportionment or a derivative of another requirement, both forward and backward traceability should be provided. Examples include:

4.3.6.1 The allocation of response time to a data base function from the overall user response time requirement.

4.3.6.2 The identification of a report format with certain functional and user interface requirements.

4.3.6.3 A software product that supports legislative or administrative needs (for example, tax computations, reporting of an overhead ratio). In this case, the exact legislative or administrative document that is being supported should be identified.

The forward traceability of the SRS is especially important when the software product enters the operation and maintenance phase. As code and design documents are modified, it is essential to be able to ascertain the complete set of requirements that may be affected by those modifications.

4.3.7 Usable During the Operation and Maintenance Phase. The SRS must address the needs of the operation and maintenance phase, including the eventual replacement of the software.

(1) Maintenance is frequently carried out by personnel not associated with the original development. Local changes (corrections) can be implemented by means of a well-commented code. For changes of wider scope, however, the design and requirements documentation is essential. This implies two actions

(a) The SRS should be modifiable as indicated in 4.3.5.

(b) The SRS should contain a record of all special provisions that apply to individual components such as:

(i) Their criticality (for example, where failure could impact safety or cause large financial or social losses).

(ii) Their relation to only temporary needs (for example, to support a display that may be retired soon).

(iii) Their origin (for example, function X is to be copied from an existing software product in its entirety).

(2) Knowledge of this type is taken for granted in the developing organization but is frequently missing in the maintenance organization. If the reason for or origin of a function is not understood, it is frequently impossible to perform adequate software maintenance on it.

4.4 Joint Preparation of the SRS. The software development process begins with supplier and customer agreement on what the completed software must do. This agreement, in the form of an SRS, should be jointly prepared. This is important because usually neither the customer nor the supplier is qualified to write a good SRS by himself.

(1) Customers usually do not understand the software design and development process well enough to write a usable SRS.

(2) Suppliers usually do not understand the customer's problem and field of endeavor well enough to specify requirements for a satisfactory system.

The customer and the supplier need to work together to produce a well written and completely understood SRS.[4]

4.5 SRS Evolution. The SRS may need to evolve as the development of the software product progresses.

(1) It may be impossible to specify some details at the time the project is initiated. For example, it may be impossible to define during the Requirements Phase, all of the screen formats for an interactive program in a manner that guarantees that they will not be altered later.

(2) Additional changes may ensue as deficien-

[4]This guide does not specifically discuss style, language usage, or techniques of good writing. It is quite important, however, that an SRS be well written; for guidance, please refer to general technical writing guides such as [1] and [11].

cies, shortcomings, and inaccuracies are discovered in the SRS, as the product evolves.

Two major considerations in this process are:

4.5.1 The requirements should be specified as completely and thoroughly as possible, even if evolutionary revisions can be forseen as inevitable. For example, the desired screen formats should be specified as well as possible in the SRS as a basis for later design.

4.5.2 A formal change process should be initiated to identify, control, track, and report projected changes, as soon as they are initially identified. Approved changes in requirements should be incorporated in the SRS in such a way as to:

(1) Provide an accurate and complete audit trail of changes.

(2) Permit the review of current and superseded portions of the SRS.

4.6 Tools for Developing an SRS. The most obvious way to create an SRS is to write it in a natural language (for example, English). But because natural languages are rich, although imprecise, a number of more formal methods have been devised to assist SRS writers.

4.6.1 Formal Specification Methodologies. The degree to which such formal methodologies may be useful in preparing an SRS depends upon a number of factors:

(1) The size and complexity of the program

(2) Whether a customer contract requires it

(3) Whether the SRS is a vehicle for contracts or merely an internal document

(4) Whether the SRS document will become the top level of the design document

(5) What computer facilities are available to support such a methodology

No attempt is made here to describe or endorse any particular tool.[5]

4.6.2 Production Tools. A computer-based word processor is a most useful production aid. Usually, an SRS will have several authors, will undergo several revisions, and will have several reorganizations. A word processor that manages the text as a computer file facilitates this process.

Almost all computer systems have a word processor and often a document preparation package is associated with it. This automates paragraphing and referencing, the printing of headings and subheadings, the compilation of tables of contents and indexes, etc, all of which help in the production of a more readable SRS.

4.6.3 Representation Tools. Some words in the SRS, especially nouns and verbs, refer specifically to entities and actions in the system. There are several advantages to identifying them as such.

(1) It is possible to verify that an entity or action always has the same name everywhere in the SRS. Thus *calculate trajectory* would not co-exist with *determine flight path*.

(2) It is possible to identify every place in the specification where a particular entity or action is described.

In addition, it may be desirable to formalize the English structure in some way to allow automated processing of the content of the SRS. With such constraints it becomes possible to:

4.6.3.1 Display the requirements in some tabular or graphical way.

4.6.3.2 Automatically check the SRS requirements in hierarchical layers of detail, where each layer is complete in itself but may also be expanded upon in a lower hierarchical layer or be a constituent of an upper hierarchical layer.

4.6.3.3 Automatically check that the SRS possesses some or all of the characteristics described in 4.3.

5. Software Requirements

Each software requirement in an SRS is a statement of some essential capability of the software to be developed. The following subsections describe:

(1) Methods used to express software requirements

(2) Annotation of the software requirements

(3) Common pitfalls encountered in the process

5.1 Methods Used To Express Software Requirements. Requirements can be expressed in a number of ways:

(1) Through input/output specifications

(2) By use of a set of representative examples

(3) By the specification of models

5.1.1 Input/Output Specifications. It is often effective to specify the required behavior of a software product as a sequence of inputs and outputs.

[5]For detailed discussion on this topic, see, for example, [6], [13], [14], and [15].

5.1.1.1 **Approaches.** There are at least three different approaches based on the nature of the software being specified:

(1) Some software products (such as reporting systems) are best specified by focusing on required outputs. In general, output-focused systems operate primarily on data files. User input usually serves to provide control information and trigger data file processing.

(2) Others are best specified by focusing on input/output behavior. Input/output-focused systems operate primarily on the current input. They are required to generate the *matching* output (as with data conversion routines or a package of mathematical functions).

(3) Some systems (such as process control systems) are required to remember their behavior so that they can respond to an input based on that input and past inputs; that is, behave like a finite state machine. In this case the focus is on both input/output pairs and sequences of such pairs.

5.1.1.2 **Difficulties.** Most software products can receive an infinite number of sequences as input. Thus, to completely specify the behavior of the product through input/output sequences would require that the SRS contain an infinitely long set of sequences of inputs and required outputs. With this approach, therefore, it may be impossible to completely specify every conceivable behavior that is required of the software.

5.1.2 **Representative Examples.** One alternative is to indicate what behavior is required by using representative examples of that behavior. Suppose, for example, that the system is required to respond with a "1" every time it receives a "0". Clearly, a list of all possible sequences of inputs and outputs would be impossible. However, by using representative sequences one might be able to fully understand the system's behavior. This system's behavior might be described by using this representative set of four dialogues:[6]

0101
010101010101
01
010101

These dialogues provide a good idea of the required inputs and outputs but they do not specify the system's behavior completely.

5.1.3 **Models.** Another approach is to express the requirements in the form of a model.[7] This can be an accurate and efficient way to express complex requirements.

At least three generalized types of models are in common usage:

(1) Mathematical
(2) Functional
(3) Timing

Care should be taken to distinguish between the model for the application; that is, a linear programming model (with a set of linear inequalities and an objective function) and the model for the software which is required to implement the application model. See 5.1.3.5.

5.1.3.1 **Mathematical Models.** A mathematical model is a model that uses mathematical relations to describe the software's behavior. Mathematical models are especially useful for particular application areas, including navigation, linear programming, econometrics, signal processing and weather analysis.

A mathematical model might specify the response discussed in 5.1.2 like this:

(01)*

where * means that the parenthesized character string is repeated one or more times.

5.1.3.2 **Functional Models.** A functional model is a model that provides a mapping from inputs to outputs. Functional models, for example, finite state machines or Petri nets can help identify and define various features of the software or can demonstrate the intended operation of the system.

A functional model might specify the response, previously described by the mathematical model, in the form of a finite state machine as shown in Fig 1. In this figure, the incoming arrow points to the starting state. The double lined box represents the accepting state. The notation X/Y on the lines indicates that when X is accepted as an input, Y is produced as an output.

5.1.3.3 **Timing Models.** A timing model is a model that has been augmented with timing constraints. Timing models are quite useful for specifying the form and details of the software's behavior, particularly for real-time systems or for human factors of any system.

[6]Each of the four sample dialogues given here (one per line) represents a sequence of one-character user inputs and one-character system outputs.

[7]For details on using modeling techniques, see [5], [8], [9], and [12].

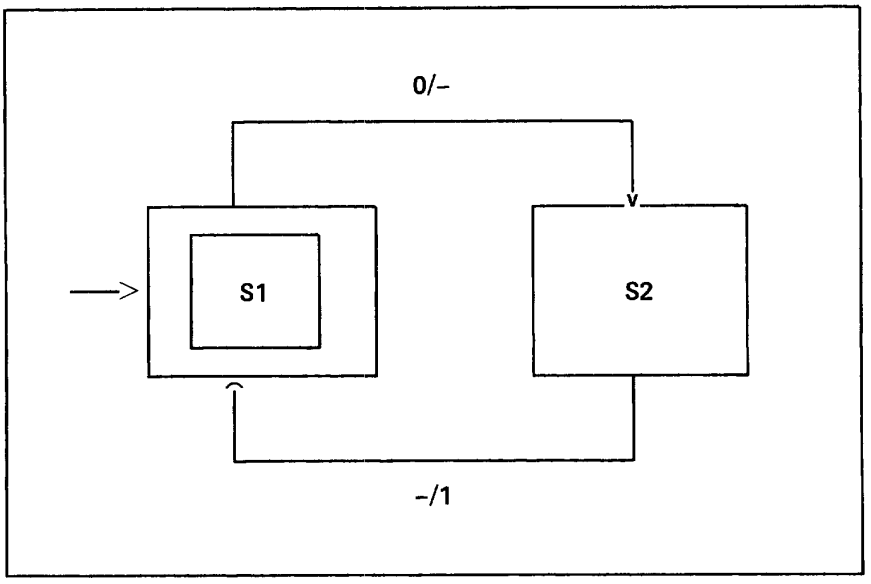


Fig 1
A Functional Model Specifying Any Sequence of Alternating 0s and 1s

A timing model might add these constraints to the model shown in Fig 1.

(1) The stimulus 0 will occur within 30 s of the arrival in state S1

(2) The response 1 will occur within 2 s of the arrival in state S2

5.1.3.4 Other Models. In addition to the aforementioned, specific applications have particularly helpful models. For example, a compiler specification might employ attribute grammars, or a payroll system might use tables. It is to be noted that the use of a formal requirements language for an SRS usually implies a need for the use of a particular model.

5.1.3.5 Cautions. Whatever type of model is used:

(1) It must be rigorously defined, either in the SRS or in a document referenced in the SRS. This definition should specify

(a) The required ranges of the model's parameters

(b) The values of constraints it uses

(c) The required accuracy of results

(d) The load capacity

(e) The required execution time

(f) Default or failure response

(2) Care must be taken to keep a model definition within the domain of requirements. Whenever an SRS uses a model:

(a) It means that the model provided an especially efficient and accurate way to specify the requirements

(b) It does not mean that the implementation of the software product must be based on that model.

A model that works effectively for explaining requirements in a written document may not be optimal for the actual software implementation.

5.2 Annotation of the Software Requirements. Typically, all of the requirements that relate to a software product are not equally important. Some requirements may be essential, especially for life-critical applications, while others may be just *nice to have.*

(1) Each requirement in the SRS should be annotated to make these differences in relative importance clear and explicit.

(2) Annotating the requirements in this manner, helps:

(a) Customers to give more careful consideration to each requirement, which often clarifies any hidden assumptions they may have.

(b) Developers to make correct design decisions and devote appropriate levels of

effort to the different parts of the software product.

5.2.1 Stability. One method of annotating requirements uses the dimension of stability. A requirement may be considered *stable* when it is thought that the needs which it addresses will not change during the expected life of the software, or it may be considered *volatile* and subject to change.

5.2.2 Degree of Necessity. Another way to annotate is to distinguish classes of requirements as mandatory, desirable, and optional.

(1) Mandatory implies that the software will not be acceptable unless these requirements are provided in an agreed manner.

(2) Desirable implies that these are requirements that would enhance the software product, but would not make it unacceptable if they are absent.

(3) Optional implies a class of functions that may or may not be worthwhile, which gives the supplier the opportunity to propose something which exceeds the SRS.

5.2.3 Annotation Caution. Prior to annotating the requirements, a thorough understanding of the contractual implications of such annotations, should be obtained.

5.3 Common Pitfalls Encountered in Expressing Requirements. An essential point about the SRS is that it should specify the results that must be achieved by the software, not the means of obtaining those results.

(1) The basic issues that the requirements writer must address are these:

(a) *Functionality* — what the software is supposed to do

(b) *Performance* — the speed, availability, response time, recovery time of various software functions, etc

(c) *Design Constraints Imposed on an Implementation* — any required standards in effect, implementation language, policies for data base integrity, resource limits, operating environment(s), etc

(d) *Attributes* — considerations of portability, correctness, maintainability, security, etc

(e) *External Interfaces* — interactions with people, hardware, other software and other hardware

(2) The requirements writer should avoid placing either design or project requirements in the SRS. The requirements writer should clearly distinguish between identifying required design constraints and projecting a design.

5.3.1 Embedding Design in the SRS. Embedding design specifications in the SRS unduly constrains the software designs and artificially places potentially dangerous requirements in the SRS.

(1) The SRS must specify what functions are to be performed on what data to produce what results at what location for whom. The SRS should focus on the services to be performed. The SRS should not normally specify design items such as

(a) Partitioning the software into modules

(b) Allocating functions to the modules

(c) Describing the flow of information or control between modules

(d) Choosing data structures

(2) It is not always practical to consider the design as being completely isolated from the SRS. Security or safety considerations may impose requirements that reflect directly into design constraints; for example, the need to

(a) Keep certain functions in separate modules

(b) Permit only limited communication between some areas of the program

(c) Compute check sums for critical quantities

In general, it must be considered that the selection of an appropriate high-level design for the software may require vast amounts of resources (perhaps as much as 10% to 20% of the total product development cost). There are two alternatives:

(1) Ignore the warning in this guide and specify the design in the SRS. This will mean that either a potentially inadequate design is stated as a requirement (because insufficient time was spent in arriving at it), or an exorbitant amount of time is spent during the requirements phase (because an entire design analysis is performed before SRS completion).

(2) Use the advice in 5.1.3 of this guide. State the requirements using a *model* design used solely to assist in the description of the requirements and not intended to serve as the actual design.

5.3.2 Embedding Project Requirements in the SRS. The SRS should address the software product, not the process of producing the software product.

(1) Project requirements represent an understanding between a customer and a supplier about the contractual matters pertaining to the production of software (and thus should not be

included in the SRS). These normally include such items as:

(a) Cost
(b) Delivery schedules
(c) Reporting procedures
(d) Software development methods
(e) Quality assurance
(f) Validation and verification criteria
(g) Acceptance procedures

(2) Project requirements are specified in other documents, typically in a computer program development plan or a statement of work. The requirements for only the software product itself are given in the SRS.

6. An SRS Prototype Outline

This section discusses each of the essential parts of the SRS. These parts are arranged in Table 1 in an outline that can serve as a prototype for any SRS.

Software suppliers and customers should tailor the content requirements of this guide based on the particular package being specified, and individual companies might base their own SRS standards upon it. Remember that while an SRS does not have to follow this outline or use the names for its parts, any good SRS must include all of the information discussed here.

6.1 Introduction (Section 1 of the SRS). The following subsections of the SRS should provide an overview of the entire SRS.

Table 1
Prototype SRS Outline

Table of Contents
1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, Acronyms, and Abbreviations
   1.4 References
   1.5 Overview
2. General Description
   2.1 Product Perspective
   2.2 Product Functions
   2.3 User Characteristics
   2.4 General Constraints
   2.5 Assumptions and Dependencies
3. Specific Requirements
   (See 6.3.2 of this guide for alternate organizations of this section of the SRS.)

Appendixes
Index

6.1.1 Purpose (1.1 of the SRS). This subsection should accomplish the following:

(1) Delineate the purpose of the particular SRS

(2) Specify the intended audience for the SRS

6.1.2 Scope (1.2 of the SRS). This subsection should:

(1) Identify the software product(s) to be produced by name; for example, Host DBMS, Report Generator, etc

(2) Explain what the software product(s) will, and, if necessary, will not do

(3) Describe the application of the software being specified. As a portion of this, it should:

(a) Describe all relevant benefits, objectives, and goals as precisely as possible. For example, to say that one goal is to provide *effective reporting capabilities* is not as good as saying *parameter-driven, user-definable reports with a 2 h turnaround and on-line entry of user parameters.*

(b) Be consistent with similar statements in higher-level specifications (for example, the System Requirement Specification), if they exist.

6.1.3 Definitions, Acronyms, and Abbreviations (1.3 of the SRS). This subsection should provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendixes in the SRS or by reference to other documents.

6.1.4 References (1.4 of the SRS). This subsection should:

(1) Provide a complete list of all documents referenced elsewhere in the SRS, or in a separate, specified document.

(2) Identify each document by title, report number — if applicable — date, and publishing organization.

(3) Specify the sources from which the references can be obtained.

This information may be provided by reference to an appendix or to another document.

6.1.5 Overview (1.5 of the SRS). This subsection should:

(1) Describe what the rest of the SRS contains

(2) Explain how the SRS is organized

6.2 The General Description (Section 2 of the SRS). This section of the SRS should describe the general factors that affect the product and its requirements.

This section usually consists of five subsections, as follows:

(1) Product Perspective
(2) Product Functions
(3) User Characteristics
(4) General Constraints
(5) Assumptions and Dependencies

It should be made clear that this section does not state specific requirements; it only makes those requirements easier to understand.

6.2.1 Product Perspective (2.1 of the SRS). This subsection of the SRS puts the product into perspective with other related products or projects.

(1) If the product is independent and totally self-contained, it should be stated here.

(2) If the SRS defines a product that is a component of a larger system or project — as frequently occurs — then this subsection should:

(a) Describe the functions of each component of the larger system or project, and identify interfaces

(b) Identify the principal external interfaces of this software product.

NOTE: This is not a detailed description of these interfaces; the detailed description is provided elsewhere in the SRS.

(c) Describe the computer hardware and peripheral equipment to be used.

NOTE: This is an overview description only.

A block diagram showing the major components of the larger system or project, interconnections, and external interfaces can be very helpful.

This subsection should not be used to impose a specific design solution or specific design constraints on the solution. This subsection should provide the reasons why certain design constraints are later specified as part of the Specific Requirements Section of the SRS.

6.2.2 Product Functions (2.2 of the SRS). This subsection of the SRS should provide a summary of the functions that the software will perform. For example, an SRS for an accounting program might use this part to address *customer account maintenance*, *customer statement and invoice preparation* without mentioning the vast amount of detail that each of those functions requires.

Sometimes the function summary that is necessary for this part can be taken directly from the section of the higher-level specification (if one exists) that allocates particular functions to the software product. Note that, for the sake of clarity:

(1) The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.

(2) Block diagrams showing the different functions and their relationships can be helpful. Remember, however, that such a diagram is not a requirement on the design of a product itself; it is simply an effective explanatory tool.

This subsection should not be used to state specific requirements. This subsection should provide the reasons why certain specific requirements are later specified as part of the Specific Requirements Section(s) of the SRS.

6.2.3 User Characteristics (2.3 of the SRS). This subsection of the SRS should describe those general characteristics of the eventual users of the product that will affect the specific requirements.

Many people interact with a system during the operation and maintenance phase of the software life cycle. Some of these people are users, operators, and maintenance and systems personnel. Certain characteristics of these people, such as educational level, experience, and technical expertise impose important constraints on the system's operating environment.

If most users of the system are occasional users, a resulting specific requirement might be that the system contains reminders of how to perform essential functions rather than assuming that the user will remember these details from the last session or from reading the user's guide.

This subsection should not be used to state specific requirements or to impose specific design constraints on the solution. This subsection should provide the reasons why certain specific requirements or design constraints are later specified as part of the Specific Requirements Section(s) of the SRS.

6.2.4 General Constraints (2.4 of the SRS). This subsection of the SRS should provide a general description of any other items that will limit the developer's options for designing the system. These can include:

(1) Regulatory policies
(2) Hardware limitations; for example, signal timing requirements
(3) Interfaces to other applications
(4) Parallel operation
(5) Audit functions

(6) Control functions
(7) Higher-order language requirements
(8) Signal handshake protocols; for example, XON –XOFF, ACK – NACK.
(9) Criticality of the application
(10) Safety and security considerations

This subsection should not be used to impose specific requirements or specific design constraints on the solution. This subsection should provide the reasons why certain specific requirements or design constraints are later specified as part of the Specific Requirements Section of the SRS.

6.2.5 Assumptions and Dependencies (2.5 of the SRS). This subsection of the SRS should list each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption might be that a specific operating system will be available on the hardware designated for the software product. If, in fact, the operating system is not available, the SRS would then have to change accordingly.

6.3 The Specific Requirements (Section 3 of the SRS). This section of the SRS should contain all the details the software developer needs to create a design. This is typically the largest and most important part of the SRS.

(1) The details within it should be defined as individual specific requirements, following the guidelines described in Section 3 of this guide (verifiable, unambiguous, etc)

(2) Background should be provided by cross-referencing each specific requirement to any related discussion in the Introduction, General Description, and Appendixes portions of the SRS, whenever possible.

(3) One way to classify the specific requirements is as follows:
(a) Functional Requirements
(b) Performance Requirements
(c) Design Constraints
(d) Attributes
(e) External Interface Requirements

The important points to be recognized are that:

(1) Specific requirements should be organized in a logical and readable fashion.

(2) Each requirement should be stated such that its achievement can be objectively verified by a prescribed method.

6.3.1 Information Required as Part of the Specific Requirements

6.3.1.1 Functional Requirements. This subsection of the SRS should specify how the inputs to the software product should be transformed into outputs. It describes the fundamental actions that must take place in the software.

For each class of function or sometimes for each individual function, it is necessary to specify requirements on inputs, processing, and outputs. These are usually organized with these four subparagraphs:

(1) Introduction. This subparagraph should provide a description of the purpose of the function and the approaches and techniques employed. It should contain any introductory or background material which might clarify the intent of the function.

(2) Inputs. This subparagraph should contain:
(a) A detailed description of all data input to this function to include:
(i) The sources of the inputs
(ii) Quantities
(iii) Units of measure
(iv) Timing
(v) The ranges of the valid inputs to include accuracies and tolerances.

(b) The details of operator control requirements should include names and descriptions of operator actions, and console or operator positions. For example, this might include required operator activities such as form alignment — when printing checks.

(c) References to interface specifications or interface control documents where appropriate.

(3) Processing. This subparagraph should define all of the operations to be performed on the input data and intermediate parameters to obtain the output. It includes specification of:
(a) Validity checks on the input data
(b) The exact sequence of operations to include timing of events
(c) Responses to *abnormal* situations, for example:
(i) Overflow
(ii) Communication failure
(iii) Error handling
(d) Parameters affected by the operations
(e) Requirements for degraded operation
(f) Any methods (for example, equations, mathematical algorithms, and logical operations) which must be used to transform the system inputs into corresponding outputs. For

example, this might specify:

(i) The formula for computing the withholding tax in a payroll package

(ii) A least squares curve fitting technique for a plotting package

(iii) A meteorological model to be used for a weather forecasting package

(g) Validity checks on the output data

(4) Outputs. This subparagraph should contain:

(a) A detailed description of all data output from this function to include:

(i) Destinations of the outputs

(ii) Quantities

(iii) Units of measure

(iv) Timing

(v) The range of the valid outputs is to include accuracies and tolerances

(vi) Disposition of illegal values

(vii) Error messages

(b) References to interface specifications or interface control documents where appropriate

In addition, for those systems whose requirements focus on input/output behavior, the SRS should specify all of the significant input/output pairs and sequences of pairs. Sequences will be needed when a system is required to remember its behavior so that it can respond to an input based on that input and past behavior; that is, behave like a finite state machine.

6.3.1.2 Performance Requirements. This subsection should specify both the static and the dynamic numerical requirements placed on the software or on human interaction with the software, as a whole.

(1) Static numerical requirements may include:

(a) The number of terminals to be supported

(b) The number of simultaneous users to be supported

(c) Number of files and records to be handled

(d) Sizes of tables and files

Static numerical requirements are sometimes identified under a separate section entitled *capacity*.

(2) Dynamic numerical requirements may include, for example, the numbers of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms, for example, *95% of the transactions shall be processed in less than 1 s*, rather than, *operator shall not have to wait for the transaction to complete.*

NOTE: Numerical limits applied to one specific function are normally specified as part of the processing subparagraph description of that function.

6.3.1.3 Design Constraints. Design constraints can be imposed by other standards, hardware limitations, etc.

6.3.1.3.1 Standards Compliance. This subsection should specify the requirements derived from existing standards or regulations. They might include:

(1) Report format

(2) Data naming

(3) Accounting procedures

(4) Audit Tracing. For example, this could specify the requirement for software to trace processing activity. Such traces are needed for some applications to meet minimum government or financial standards. An audit trace requirement might, for example, state that all changes to a payroll data base must be recorded in a trace file with before and after values.

6.3.1.3.2 Hardware Limitations. This subsection could include requirements for the software to operate inside various hardware constraints. For example, these could include:

(1) Hardware configuration characteristics (number of ports, instruction sets, etc)

(2) Limits on primary and secondary memory

6.3.1.4 Attributes. There are a number of attributes that can place specific requirements on the software. Some of these are indicated below. These should not be interpreted to be a complete list.

6.3.1.4.1 Availability. This could specify the factors required to guarantee a defined availability level for the entire system such as checkpoint, recovery and restart.

6.3.1.4.2 Security. This could specify the factors that would protect the software from accidental or malicious access, use, modification, destruction, or disclosure. Specific requirements in this area could include the need to:

(1) Utilize certain cryptographical techniques

(2) Keep specific log or history data sets

(3) Assign certain functions to different modules

(4) Restrict communications between some areas of a program

(5) Compute checksums for critical quantities

6.3.1.4.3 Maintainability. This could specify the requirements to ensure that the

software could be maintained. For example, as a part of this,

(1) Specific coupling metrics for the software modules could be required

(2) Specific data/program partitioning requirements could be specified for micro-devices

6.3.1.4.4 Transferability/Conversion. This could specify the user procedures, user interface compatibility constraints (if any) etc, required to transport the software from one environment to another.

6.3.1.4.5 Caution. It is important that required attributes be specified so that their achievement can be objectively verified by a prescribed method.

6.3.1.5 External Interface Requirements

6.3.1.5.1 User Interfaces. This should specify:

(1) The characteristics that the software must support for each human interface to the software product. For example, if the user of the system operates through a display terminal, the following should be specified:

(a) Required screen formats

(b) Page layout and content of any reports or menus

(c) Relative timing of inputs and outputs

(d) Availability of some form of programmable function keys

(2) All the aspects of optimizing the interface with the person who must use the system. This may simply comprise a list of do's and don'ts on how the system will appear to the user. One example might be a requirement for the option of long or short error messages. Like all others, these requirements should be verifiable and for example, *a clerk typist grade 4 can do function X in Z min* after 1 h *of training* rather than *a typist can do function X.* (This might also be specified in the Attributes section under a section titled *Ease of Use.*)

6.3.1.5.2 Hardware Interfaces. This should specify the logical characteristics of each interface between the software product and the hardware components of the system. It also covers such matters as what devices are to be supported, how they are to be supported, and protocols. For example, terminal support might specify full screen support as opposed to line by line.

6.3.1.5.3 Software Interfaces. This should specify the use of other required software products (for example, a data management system, an operating system, or a mathematical package), and interfaces with other application systems (for example, the linkage between an accounts receivable system and a general ledger system).

For each required software product, the following should be provided:

(1) Name

(2) Mnemonic

(3) Specification number

(4) Version number

(5) Source

For each interface, this part should:

(1) Discuss the purpose of the interfacing software as related to this software product.

(2) Define the interface in terms of message content and format. It is not necessary to detail any well-documented interface, but a reference to the document defining the interface is required.

6.3.1.5.4 Communications Interfaces. This should specify the various interfaces to communications such as local network protocols, etc.

6.3.1.6 Other Requirements. Certain requirements may, due to the nature of the software, the user organization, etc, be placed in separate categories as indicated below.

6.3.1.6.1 Data Base. This could specify the requirements for any data base that is to be developed as part of the product. This might include:

(1) The types of information identified in 6.3.1.1

(2) Frequency of use

(3) Accessing capabilities

(4) Data element and file descriptors

(5) Relationship of data elements, records and files

(6) Static and dynamic organization

(7) Retention requirements for data

NOTE: If an existing data base package is to be used, this package should be named under *Interfaces to Software* and details of using it specified there.

6.3.1.6.2 Operations. This could specify the normal and special operations required by the user such as:

(1) The various modes of operations in the user organization; for example, user-initiated operations

(2) Periods of interactive operations and periods of unattended operations

(3) Data processing support functions

(4) Backup and recovery operations

NOTE: This is sometimes specified as part of the User Interfaces section.

6.3.1.6.3 Site Adaptation Requirements. This could:

(1) Define the requirements for any data or initialization sequences that are specific to a given site, mission, or operational mode, for example, grid values, safety limits, etc.

(2) Specify the site or mission-related features that should be modified to adapt the software to a particular installation.

6.3.2 Organizing The Specific Requirements. This subsection is often the largest and most complex of all the parts of the SRS.

(1) It may be necessary to organize this section into subdivisions according to the primary classes of functions to be performed by the software. For example, consider a large interactive accounting system. This may be broken down at the top level into operational software (which supports near-real-time transactions), support software (logging functions, disk backup, loading tapes, etc), and diagnostic software (primarily hardware and communications support), and at the next level into accounts receivable, accounts payable, etc.

(2) It should be remembered, however, that the purpose of this subdivided organization is to improve the readability of the SRS, not to define the high level design of the software being specified.

The best organization for Section 3, Specific Requirements, in an SRS, depends on the application area and the nature of the software product being specified. Tables 2 through 4 show four possible organizations.

(1) In Prototype Outline 1 (Table 2), all the Functional Requirements are specified, then the four types of interface requirements are specified, and then the rest of the requirements are specified.

(2) Prototype Outline 2 (Table 3) shows the four classes of interface requirements applied to each individual Functional Requirement. This is followed by the specification of the rest of the requirements.

(3) In Prototype Outline 3 (Table 4), all of the issues addressed by the Functional Requirements are specified, then the other requirements that apply to them are specified. This pattern is then repeated for each of the External Interface Requirement Classifications.

Table 2
Prototype Outline 1 for SRS Section 3

3. Specific Requirements
   - 3.1 Functional Requirements
     - 3.1.1 Functional Requirement 1
       - 3.1.1.1 Introduction
       - 3.1.1.2 Inputs
       - 3.1.1.3 Processing
       - 3.1.1.4 Outputs
     - 3.1.2 Functional Requirement 2
     - . . . .
     - 3.1.*n* Functional Requirement *n*
   - 3.2 External Interface Requirements
     - 3.2.1 User Interfaces
     - 3.2.2 Hardware Interfaces
     - 3.2.3 Software Interfaces
     - 3.2.4 Communications Interfaces
   - 3.3 Performance Requirements
   - 3.4 Design Constraints
     - 3.4.1 Standards Compliance
     - 3.4.2 Hardware Limitations
     - . . . .
   - 3.5 Attributes
     - 3.5.1 Security
     - 3.5.2 Maintainability
     - . . . .
   - 3.6 Other Requirements
     - 3.6.1 Data Base
     - 3.6.2 Operations
     - 3.6.3 Site Adaptation
     - . . . .

Table 3
Prototype Outline 2 for SRS Section 3

3. Specific Requirements
   - 3.1 Functional Requirements
     - 3.1.1 Functional Requirement 1
       - 3.1.1.1 Specification
         - 3.1.1.1.1 Introduction
         - 3.1.1.1.2 Inputs
         - 3.1.1.1.3 Processing
         - 3.1.1.1.4 Outputs
       - 3.1.1.2 External Interfaces
         - 3.1.1.2.1 User Interfaces
         - 3.1.1.2.2 Hardware Interfaces
         - 3.1.1.2.3 Software Interfaces
         - 3.1.1.2.4 Communication Interfaces
     - 3.1.2 Functional Requirement 2
     - . . . .
     - 3.1.*n* Functional Requirement *n*
   - 3.2 Performance Requirements
   - 3.3 Design Constraints
   - 3.4 Attributes
     - 3.4.1 Security
     - 3.4.2 Maintainability
     - . . . .
   - 3.5 Other Requirements
     - 3.5.1 Data Base
     - 3.5.2 Operations
     - 3.5.3 Site Adaption
     - . . . .

Table 4
Prototype Outline 3 for SRS Section 3

3. Specific Requirements
  3.1 Functional Requirements
    3.1.1 Functional Requirement 1
      3.1.1.1 Introduction
      3.1.1.2 Inputs
      3.1.1.3 Processing
      3.1.1.4 Outputs
      3.1.1.5 Performance Requirements
      3.1.1.6 Design Constraints
        3.1.1.4.1 Standards Compliance
        3.1.1.4.2 Hardware Limitations
        . . . .
      3.1.1.7 Attributes
        3.1.1.7.1 Security
        3.1.1.7.2 Maintainability
        . . . .
      3.1.1.8 Other Requirements
        3.1.1.8.1 Data Base
        3.1.1.8.2 Operations
        3.1.1.8.3 Site Adaption
        . . . .
    3.1.2 Functional Requirement 2
    . . . .
    3.1.*n* Functional Requirement *n*
  3.2 External Interface Requirements
    3.2.1 User Interfaces
      3.2.1.1 Performance Requirements
      3.2.1.2 Design Constraints
        3.2.1.2.1 Standards Compliance
        3.2.1.2.2 Hardware Limitations
        . . . .
      3.2.1.3 Attributes
        3.2.1.3.1 Security
        3.2.1.3.2 Maintainability
        . . . .
      3.2.1.4 Other Requirements
        3.2.1.4.1 Data Base
        3.2.1.4.2 Operations
        3.2.1.4.3 Site Adaption
        . . . .
    3.2.2 Hardware Interfaces
    3.2.3 Software Interfaces
    3.2.4 Communications Interfaces

Table 5
Prototype Outline 4 for SRS Section 3

3. Specific Requirements
  3.1 Functional Requirement 1
    3.1.1 Introduction
    3.1.2 Inputs
    3.1.3 Processing
    3.1.4 Outputs
    3.1.5 External Interfaces
      3.1.5.1 User Interfaces
      3.1.5.2 Hardware Interfaces
      3.1.5.3 Software Interfaces
      3.1.5.4 Communication Interfaces
    3.1.6 Performance Requirements
    3.1.7 Design Constraints
    3.1.8 Attributes
      3.1.8.1 Security
      3.1.8.2 Maintainability
      . . . .
    3.1.9 Other Requirements
      3.1.9.1 Data Base
      3.1.9.2 Operations
      3.1.9.3 Site Adaption
      . . . .
  3.2 Functional Requirement 2
  . . . .
  3.*n* Functional Requirement *n*

(4) In Prototype Outline 4 (Table 5), the interface requirements and the rest of the requirements are specified as they pertain to each Functional Requirement.

The organization of the Specific Requirements Section of the SRS should be chosen with the goal of properly specifying the requirements in the most readable manner.

6.4 Supporting Information. The supporting information; that is, the Table of Contents, the Appendixes, and the Index, make the SRS easier to use.

(1) The Table of Contents and Index are quite important and should follow the generally accepted rules for good documentation practices.[8]

(2) The Appendixes are not always considered part of the actual requirements specification and are not always necessary. They might include:

(a) Sample I/O formats, descriptions of cost analysis studies, or results of user surveys.

(b) Supporting or background information that can help the readers of the SRS.

(c) A description of the problems to be solved by the software.

(d) The history, background, experience and operational characteristics of the organization to be supported.

(e) A cross-reference list, arranged by milestone, of those incomplete software requirements that are to be completed by specified milestones. (See 4.3.2 and 4.3.3 (4).)

(f) Special packaging instructions for the code and the media to meet security, export, initial loading, or other requirements.

(3) When Appendixes are included, the SRS should explicitly state whether or not the Appendixes are to be considered part of the requirements.

[8]See, for example: [4] and [11].