Software Requirements Specification

for

Lexical Analyzer

OR

Lexer

Version 1.0

EmertXe Information Technologies (P) Ltd.

10-November-2012

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
Team Emetxe	10/11/12	Initial Draft	0.1
Team Emetxe	16/11/12	Addition of more information	1.0

1. Introduction

1.1 Purpose

<Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.>

Lexical Analyser is a program which converts the stream of individual characters (normally arranged as lines) into the stream of lexical tokens (tokenization for instance of "words" and punctuation symbols that make up source code).

lexical analyzers are designed to recognize keywords , operators , and identifiers , as well as integers, floating point numbers , character strings , and other similar items that are written as part of the source program.

Additionally, it also filters out whatever (usually white-space, newlines, comments, etc) separates the tokens. The main purpose of this phase is to make the subsequent phase easier .

Lexical Analyzer is basically a part of compiler which:

i. Translates lexemes into tokens (arranged in symbol table for compilation references) with the help of Lex .

ii. Communicates with parser for serving token requests.

iii. Discards comments and skips over white spaces.

iv. Keeps track of current line number so that parser can detect errors .

The main purpose/goal of the project is to take in a c file, .c, and produce the sequence of tokens that can be used for the next stage in compilation.

1.2 Document Conventions

<Describe any standards or typographical conventions that were followed when writing this SRS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

The SDLC for this product uses the RDCT model:

R – Requirements

D – Design, both high and low level.

C – Coding

T – Testing (Unit Testing + System Testing)

The documents listed above will have tags against each important requirement, design, coding or testing item. Traceability can be documented and verified for completeness of the product.

The design and code will use structured design concepts. The modularity will be expressed using suitable notation such as flowcharts and pseudo code.

Some key definitions related to this project include:

Lex: A set of buffered input routines and constructs. It translates regular expression into lexical analyzer.

Tokens: Basic indivisible lexical unit or language elements. These are terminal symbols in a grammar, Constants, Operators, Punctuation, Keywords, Classes of sequences of characters with collective meaning, arbitrary integer values, etc that represent the lexemes .

Lexeme: These are original string (character sequence) comprised (matched) by an instance of the token. E.g. "sqrt" .

1.3 Intended Audience and Reading Suggestions

<Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SRS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.>

This document is the requirement specifications for the Lexical Analyser. It documents all the requirements for the product. It serves as a reference for validating the same with the customer. Once accepted by the customer, it serves as a reference for high level design for the "Lexical Analyzer". The designs and requirements are tagged for traceability.

1.4 Product Scope

<Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies. If a separate vision and scope document is available, refer to it rather than duplicating its contents here.>

Lexical Analyzer will be a small product demonstrating the use of C language. It will assume the source program stores in a file which will be accessed using the file operations and also assuming that the input is taken from the output of the preprocessor. It is designed only for the C language, not for any other language. It will not be using any optimal data structures or algorithms in the initial versions.

This project will be designed only for the sub-set of C.Later versions can incorporate error handling's more efficiently using suitable data structures and algorithms.

1.5 References

<List any other documents or Web addresses to which this SRS refers. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.>

1. http://en.wikipedia.org/wiki/Flex_lexical_analyser

2. Overall Description

2.1 Product Perspective

<Describe the context and origin of the product being specified in this SRS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SRS defines a component of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.>

Lexical Analyzer is a new product. It is designed to support a set of functionalities and for a sub-set of C language in the initial version. Later versions can extend it for full version of C and other languages and to perform more efficient operations.

Lexical Analyzer is basically a part of the compiler.A compiler translates and/or compiles a program written in a suitable source language into an equivalent target language through a number of stages. Starting with recognition of token through target code generation provide a basis for communication interface between a user and a processor in significant amount of time.

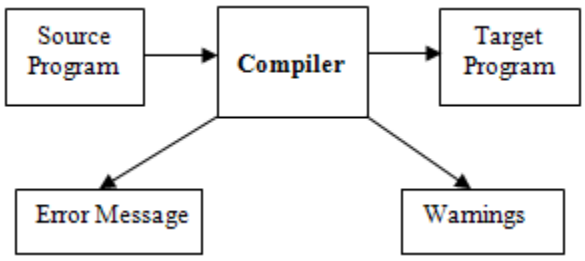


Figure 1: Working methodology of the compiler

Writing a compiler is a nontrivial task. It will be a very nice practice to structure its principles. Conceptually a compiler works in phases. The key phases include and undergo through Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Code Optimization, and Target Code Generation . These are shown in Figure 2.

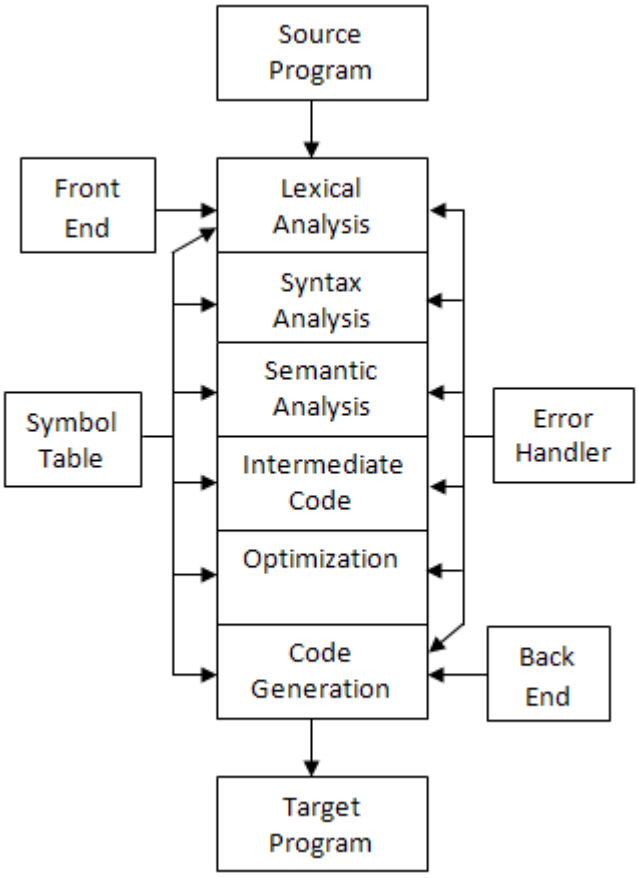
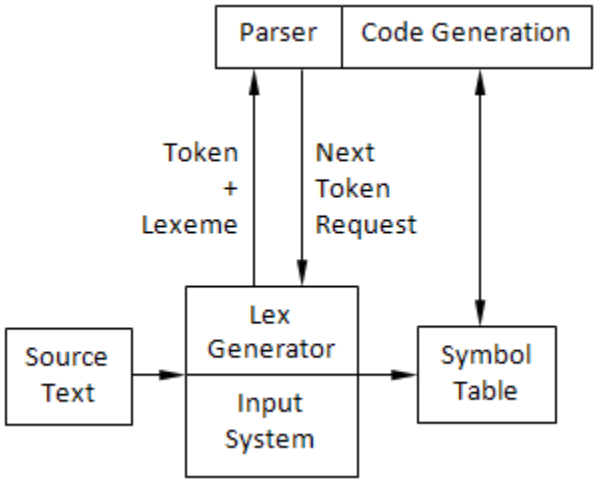


Figure 2: Phases of a Typical Compiler

Working methodology of lexical analyzer has been traced in some interesting phases as stated below:

First, it acts as an interface for parser and symbol table with input stream as reference as shown in Figure 3



Second, internal working procedure of Lexical Analyzer that generates a stream of tokens. Suppose the pseudo-code:

```
if(x*y<10)
{
        Z = x;
}
```

Let's consider the first statement of the above code. The corresponding token stream of pairs <type, value> is shown in Figure 4.

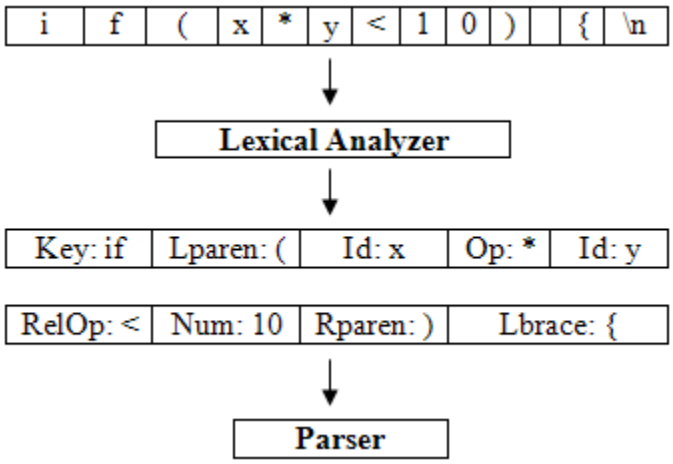


Figure 4: Output Stage of Lexical Analyzer

2.2 Product Functions

<Summarize the major functions the product must perform or must let the user perform. Details will be provided in Section 3, so only a high level summary (such as a bullet list) is needed here. Organize the functions to make them understandable to any reader of the SRS. A picture of the major groups of related requirements and how they relate, such as a top level data flow diagram or object class diagram, is often effective.>

The functionality to be implemented in the initial version are:

1. Tokenization.
2. Skipping white spaces in the source code.
3. Error handling.

2.3 User Classes and Characteristics

<Identify the various user classes that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the most important user classes for this product from those who are less important to satisfy.>

This product can be used by the people who are involved in the designing of the compiler. This product can be more used by the class of Developers who are involved in the designing and development of the parser of the compiler.

2.4 Operating Environment

<Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and any other software components or applications with which it must peacefully coexist.>

This product can be used on all systems where standard C library support is present. However, initial version will be for Ubuntu/Linux.

2.5 Design and Implementation Constraints

<Describe any items or issues that will limit the options available to the developers. These might include: corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; parallel operations; language requirements; communications protocols; security considerations; design conventions or programming standards (for example, if the customer's organization will be responsible for maintaining the delivered software).>

This project is meant more for demonstration than for an end user. No sophisticated data structures should be used for initial version. Later versions can add more to it after gaining sufficient knowledge of data structures, efficient algorithms .

This project requires compiler for compiling the source code of the lexer.

2.6 User Documentation

<List the user documentation components (such as user manuals, on-line help, and tutorials) that will be delivered along with the software. Identify any known user documentation delivery formats or standards.>

1. README.txt Release Notes files.
2. README.html browser based help.

2.7 Assumptions and Dependencies

<List any assumed factors (as opposed to known facts) that could affect the requirements stated in the SRS. These could include third-party or commercial components that you plan to use, issues around the development or operating environment, or constraints. The project could be affected if these assumptions are incorrect, are not shared, or change. Also identify any dependencies the project has on external factors, such as software components that you intend to reuse from another project, unless they are already documented elsewhere (for example, in the vision and scope document or the project plan).>

It will assume the source program stores in a file which will be accessed using the file operations and also assuming that the input is taken from the output of the preprocessor.

3. External Interface Requirements

3.1 User Interfaces

<Describe the logical characteristics of each interface between the software product and the users. This may include sample screen images, any GUI standards or product family style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. Define the software components for which a user interface is needed. Details of the user interface design should be documented in a separate user interface specification.>

1. Lexical Analyzer will be invoked using command line prompt.

   Example : **$ ./lex filename.c**

   Assumption : Output file name is lex

2. Errors generated by the Lexical analyzer will be directed to log file .

3.2 Hardware Interfaces

<Describe the logical and physical characteristics of each interface between the software product and the hardware components of the system. This may include the supported device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.>

3.3 Software Interfaces

<Describe the connections between this product and other specific software components (name and version), including databases, operating systems, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the system and going out and describe the purpose of each. Describe the services needed and the nature of communications. Refer to documents that describe detailed application programming interface protocols. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating system), specify this as an implementation constraint.>

3.4 Communications Interfaces

<Describe the requirements associated with any communications functions required by this product, including e-mail, web browser, network server communications protocols, electronic forms, and so on. Define any pertinent message formatting. Identify any communication

standards that will be used, such as FTP or HTTP. Specify any communication security or encryption issues, data transfer rates, and synchronization mechanisms.>

4. System Features

<This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product. You may prefer to organize this section by use case, mode of operation, user class, object class, functional hierarchy, or combinations of these, whatever makes the most logical sense for your product.>

The product will use multiple phases for delivering the requirements. The features below are scheduled for a first release and a second release. The first release will have the basic functionalities.

Phase 1: Features

These are the basic functionality of Lexical Analyzer.

4.1 Removal of white spaces

This is the feature by which lexical analyzer eliminates the white spaces. Many languages allow "white space" to appear between tokens. If white space is eliminated by the lexical analyzer, the parser will never have to consider it.

<Don't really say "System Feature 1." State the feature name in just a few words.>

4.1.1 Description and Priority

<Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>

Priority : 9

1. This feature has to remove all the white spaces such as space , newline, tab in the source code.

4.1.1.1 Exceptions

Under this exception , this feature should not remove the white space which is present along with the characters within the " and " .

Example: consider the string literal "Hello World"

Here the white space is present between the two string which is enclosed within the double quotation marks. So , the lexical analyzer should not remove this white space.

4.1.2 Stimulus/Response Sequences

<List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>

4.1.3 Functional Requirements

<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate when necessary information is not yet available.>

<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>
Table 1:

TAG	Function	Action
RS_LA_01	Removing white spaces	1. It should remove the space , tabs , newlines in the source code.

4.2 Tokenization

A **token** is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization**, and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each "(" is matched with a ")".

Consider this expression in the C programming language:

```
sum=3+2;
```

Tokenized in the following table 2:

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Integer literal
+	Addition operator
2	Integer literal
;	End of statement

The lexical structure of most languages is simple enough that lexical analyzers can be constructed easily by hand. In addition, automatically generated analyzers, such as those produced by LEX, tend to be large and slower than analyzers built by hand. Tools like LEX are very use- ful, however, for one-shot programs and for applications with complex lexical structures.

Table 3:

Lexical Pattern in the Inputted C program	**Token that Should Be Outputted**
Reserved Words	
auto,break,case,char,const,continue,default,do,double,else ,enum,extern,float,for,goto,if,int,long,register,return,short ,signed,sizeof,static,struct,switch,typedef,union,unsigned, void,volatile,while	KEYWORD
Arithmetic Operators	
+ , - , * , / , %	A_OPERARATOR
Comparison Operators	
== ,< > , < , > , <= , >=	C_OPERATOR
Identifiers and Numbers	
identifier	IDENTIFIER
Other Special Symbols	
;	SEMICOLON
:	COLON
,	COMMA
(	L_PAREN
)	R_PAREN
=	ASSIGN
{	L_FPAREN
}	R_FPAREN

4.2.1 Description and Priority

<Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>

Priority 9:

This feature should

1. Identify the keywords , literals , identifiers , delimiters .
2. Group them under the predefined tokens.

4.2.2 Stimulus/Response Sequences

<List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>

4.2.3 Functional Requirements

<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate when necessary information is not yet available.>

<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>

Table 4:

TAG	Function	Action
RS_LA_02_01	Recognizing the keywords	This should recognize all the keywords listed by the language. For all keywords: Refer table 3
RS_LA_02_02	Recognizing the operators	This should recognize all the different operators available in C and should group them under different category. For operators refer table 3
RS_LA_02_03	Recognizing Identifiers	This should recognize the VALID identifier in the c language. Rule 1: **Name of identifier includes alphabets, digit and underscore.** Valid name: world, addition23, sum_of_number etc. Invalid name: factorial#, avg value, display*number etc. Rule 2: **First character of any identifier must be either alphabets or underscore.** Valid name: _calulate, _5,a_, __ etc. Invalid name: 5_, 10_function, 123 etc.

		Rule 3: **Name of identifier cannot be any keyword of c program.** Invalid name: interrupt, float, asm, enum etc.
RS_LA_02_03	Recognizing Numbers Literals Doubles Integers	This should recognize all the four kinds of numeric constants in C. Such as constant: floating-constant integer-constant enumeration-constant character-constant enumeration-constant: identifier
RS_LS_02_04	Recognizing Character Constants and Strings	It should Recognize character constants and string literals is complicated by escape sequences like \n, \034, \xFF, and \", and by wide-character constants.

4.3 Error generations

This feature will track the error such as if wrong file is entered. If any unrecognized symbols are encountered . And also if any invalid identifier is encountered it should throw the error.

4.3.1 Description and Priority

<Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>

Priority: 9

This feature will generate the errors in the following conditions

1. If no file is passed as an argument in the command line.
2. If the file name is not .c extension file.
3. If the source c file is not passed through the preprocessor.

4. Passing the file which is not present in the directory.
5. If any unrecognized symbols are encountered.
6. If any invalid identifiers are encountered.

4.3.2 Stimulus/Response Sequences

<List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>

Response: All the error are directed towards the log file.

4.3.3 Functional Requirements

<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate when necessary information is not yet available.>

TAG	Function	Action
RS_LA_03_01	Recognizing the different types of errors	1. If no file is passed as an argument in the command line. 2. If the file name is not .c extension file. 3. If the source c file is not passed through the preprocessor. 4. Passing the file which is not present in the directory. 5. If any unrecognized symbols are encountered. 6. If any invalid identifiers are encountered. Under all this error conditions, all the error generated by this function should be saved in the log file for further action.

Phase II : Features

In this phase we are adding the features like,

1. Symbol table.
2. More efficient error handling.
3. Adding more command line options to get the particular output.

5. Other Nonfunctional Requirements

5.1 Performance Requirements

<If there are performance requirements for the product under various circumstances, state them here and explain their rationale, to help the developers understand the intent and make suitable design choices. Specify the timing relationships for real time systems. Make such requirements as specific as possible. You may need to state performance requirements for individual functional requirements or features.>

5.1.1 Quick Loading and Initial Response (timing to be decided)

The program must quickly respond to command line and load the file quickly.

5.2 Safety Requirements

<Specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product. Define any safeguards or actions that must be taken, as well as actions that must be prevented. Refer to any external policies or regulations that state safety issues that affect the product's design or use. Define any safety certifications that must be satisfied.>

5.2.1 Data File Safety

5.3 Security Requirements

<Specify any requirements regarding security or privacy issues surrounding use of the product or protection of the data used or created by the product. Define any user identity authentication requirements. Refer to any external policies or regulations containing security issues that affect the product. Define any security or privacy certifications that must be satisfied.>

5.3.1 Information Security

5.4 Software Quality Attributes

<Specify any additional quality characteristics for the product that will be important to either the customers or the developers. Some to consider are: adaptability, availability, correctness, flexibility, interoperability, maintainability, portability, reliability, reusability, robustness, testability, and usability. Write these to be specific, quantitative, and verifiable when possible. At the least, clarify the relative preferences for various attributes, such as ease of use over ease of learning.>

1. SDLC will be using RDCT model.
2. Design should be modular.
3. At least about 4 independent modules should be developed in parallel.
4. Unit Testing should be done rigorously.
5. System Testing should be done.
6. Test Coverage should be above 95%.
7. Reviews will be done for all artifacts as per the model.

5.5 Business Rules

<List any operating principles about the product, such as which individuals or roles can perform which functions under specific circumstances. These are not functional requirements in themselves, but they may imply certain functional requirements to enforce the rules.>

6. Other Requirements

<Define any other requirements not covered elsewhere in the SRS. This might include database requirements, internationalization requirements, legal requirements, reuse objectives for the project, and so on. Add any new sections that are pertinent to the project.>

Appendix A: Glossary

<Define all the terms necessary to properly interpret the SRS, including acronyms and abbreviations. You may wish to build a separate glossary that spans multiple projects or the entire organization, and just include terms specific to a single project in each SRS.>

1. *SDLC – Software Development Life Cycle*
2. *SRS – **S**ystem **R**equirements **S**pecification*
3. *HLD – **H**igh **L**evel **D**esign (Specification Document)*
4. *ST – **S**ystem **T**esting (Specification and Test Cycles Document)*
5. *IT – **I**ntegration **T**esting (Specification and Test Cycles Document)*
6. *UT – **U**nit **T**esting (Specification and Test Cycles Document)*
7. *LLD – **L**ow **L**evel **D**esign (Specification Document)*

Appendix B: Analysis Models

<Optionally, include any pertinent analysis models, such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship diagrams.>

Appendix C: To Be Determined List

<Collect a numbered list of the TBD (to be determined) references that remain in the SRS so they can be tracked to closure.>