

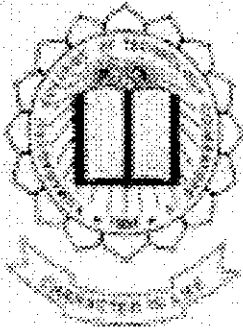
C/C++ API SPECIFICATION TOOL

PROJECT REPORT

P-932

Submitted in partial fulfillment of the requirements for
award of degree

M.Sc.,[Applied Science] Software Engineering



Submitted By

M. Ranjith

9837S0060



UNDER THE GUIDANCE OF,

External Guide

Mr. Rajesh Kumar

General Manager,

Tata Elxsi

Internal Guide

Ms. S.Devaki,

Asst Professor

CSE Department

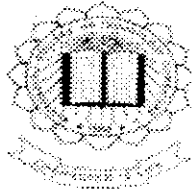
Department Of Computer Science and Engineering

Kumaraguru College Of Technology

(Affiliated to Bharathiar University)

Coimbatore-641006.

CERTIFICATE



Department of Computer Science and Engineering
Kumaraguru College of Technology
Coimbatore-641006.

This is to certify that the project work entitled "C/C++ API Specification Tool"
Has been submitted by

Mr. M.Ranjith

In partial fulfillment of the award of the degree of
Master of Science in Applied Science- Software Engineering of
Bharathiar University, Coimbatore
during the academic year 2002-2003.

Guide

Head of the Department

Certified that the candidate was examined by us in the Project Work Viva Voce
Examination held on 5/04/03 and the University Register Number
was 9837S0060.

Internal Examiner

External Examiner



March 20, 2003

TO WHOMSOEVER IT MAY CONCERN

This is to certify that Mr. Ranjith M, a Final Year M.Sc. Software Engineering student of Kumaraguru College of Technology, undertook a three months project and worked on a project titled C/C++ API Specification Tool starting from December 9, 2002 – March 20, 2003

We wish him all the best in his future endeavors

For Tata Elxsi Ltd

A handwritten signature in black ink, appearing to read "G. Sriyan".

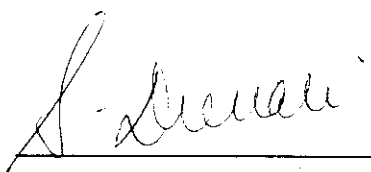
G. Sriyan

Associate Manager - Human Resources

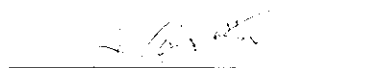
DECLARATION

I hereby declare that this project work titled ' C/C++ API Specification Tool' is a record of original project work done by me under the guidance of Mr. Rajesh Kumar as external guide and Ms. S. Devaki, Asst Professor as internal guide, and this project work has not formed the basis for the award of any Degree/ Diploma / Associate ship/ Fellowship on similar titles to any other candidates of any university.

Date .



Internal Guide,
Ms. S. Devaki, B.E, M.S
Dept CSE,
Kumaraguru College of Technology,
Coimbatore.



M. Ranjith

External Guide,
Mr. Rajesh Kumar
Tata Elxsi Limited
Bangalore.

Acknowledgement

“Nothing concrete can be achieved without an optimal combination of inspiration & perspiration”

I thank the almighty; he has been very generous and kind to me. My parents have been my source of inspiration. They have sacrificed almost every thing to provide me with an excellent foundation. I will never be able to translate my gratitude in the form of words.

I am sincerely thankful to Mr. Vijay KrishnaMurthy Chief Finance Officer, & A. Krishna Bhagavan General Manager, Operations. Tata Elxsi, Bangalore for allowing me to undertake the project in the organization.

Special thanks to Mr. Rajesh Kumar, General Manager, & Mr. Ajax Thomas, Project Leader, TATA ELXSI, Bangalore for their constant encouragement, technical inputs, and valuable suggestions and support despite their busy schedule. For the inspiration part of my tasks, I owe my deepest gratitude to the individuals who directly or indirectly came into contact during the completion of my project and without the help and guidance of whom, the successful completion of this project would have not been possible.

Kumaraguru College of Technology was the best thing that could have happened to me. My sincere gratitude to its principal Dr.K.K.Padmanaban B.Sc, M.Tech, Ph.D.

I extend my gratitude to Dr. S.Thangaswamy, our beloved HOD of CSE, & our course coordinator Ms. S. Devaki for their constant support, encouragement and valuable internal guidance.

Last but not the least, I thank all my lecturers, friends and colleagues who made life much easier for me.

Abstract

ABSTRACT

The idea of C/C++ API Specification Tool is to provide a tool that supports the *programmer* for writing *high quality* documentation (Headers) while keeping concentration on the program development. In order to do so, it is important, that the programmer can add the headers right into the source code he/she develops. Only with such an approach, a programmer would really write some documentation to his/her functions, classes, methods etc. and keep them up to date with upcoming changes of code. Hence, the only place where to put headers are as comments.

This is exactly what C/C++ API Specification Tool is used for, generating headers. However, headers are comments describing classes, functions, methods, reserved words and system calls etc. such that he/she or someone else would be able to use the code later on.

Now, let's consider what "high quality" documentation means. Many programmers like to understand the previous implementation before he/she could make any enhancements. Thus C/C++ API Specification Tool has been designed to produce formatted headers of functions, classes reserved words, system calls in a structured way.

For the output format, it is important that the headers are well structured. C/C++ API Specification Tool inserts headers into the source code developed by the programmer. The headers are placed in such a way that the body follows the comment header.

Tool : C, C++ and Gcc Compiler.

Techniques : Scanner, Parsing and Code Generation.

Contents

CONTENTS

1. ABOUT THE COMPANY.	1.
2. INTRODUCTION.	6.
2.1 Organization of Writing.	9.
3. OVERVIEW.	10.
4. NEED FOR C/C++ API SPECIFICATION Tool.	17.
4.1 Purpose.	18.
4.2 Document Conventions.	18.
4.3 Product Scope.	18.
5. SYSTEM REQUIREMENT SPECIFICATION.	20.
5.1 Purpose.	21.
5.2 Scope.	21.
5.3 Definitions, Acronyms, Abbreviations.	21.
5.4 References.	21.
5.5 General Description.	22.
5.5.1 Product Perspective.	22.
5.5.2 Product Function.	22.
5.5.3 User Characteristics.	22.
5.5.3.1 Use Case Diagram.	23.
5.6 Operating Environment.	24.
5.7 User Interface.	24.
5.8 Design Constraints.	24.
5.8.1 Hardware Limitations.	24.
5.9 General Constraints.	24.
5.10 Functional Requirements.	24.
5.11 List Of Inputs.	24.
5.12 Performance Requirements.	25.
5.13 Safety Requirements.	25.
5.14 Software Quality Attributes.	25.
5.15 Business Rules.	25.

5.16	Other Requirements.	25.
5.17	Risk Analysis.	25.
6.	HIGH LEVEL DESIGN	26.
6.1	Introduction.	27.
6.2	Purpose.	27.
6.3	Scope.	27.
6.4	Conventions.	28.
6.5	Document Status.	28.
6.6	System Description.	28.
6.6.1	System Requirements.	28.
6.6.2	System Context.	29.
6.6.2.1	Target Environment.	29.
6.6.3	External Architecture.	30.
6.7	Architecture Goals.	32.
6.8	General specifications.	32.
6.9	Scope of Implementation.	33.
6.9.1	Limitations.	33.
6.10	Design Trade-off Decisions.	33.
6.11	Test & Debug Requirements.	33.
6.12	Internal Architecture.	34.
6.13	Interconnectivity.	35.
6.14	Input / Output.	35.
6.14.1	I/O Diagram.	35.
6.14.2	I/O Diagram Representations.	36.
6.15	Functional Specifications.	37.
6.16	Requirements (From rest of the system).	37.
6.17	Performance Specifications.	37.
6.18	Timing/Sequencing/Synchronization/Buffering.	37.
6.19	System Specifications.	38.
6.19.1	OS Calls.	38.
6.19.2	Dependencies.	38.

6.19.3 Initialization.	38.
6.19.4 Termination.	38.
6.19.5 Error Messages	38.
6.20 Other Specifications.	39.
6.21 Compile Time Configuration.	39.
6.22 Tools Specifications.	39.
6.23 Resource Specifications.	39.
6.24 MIPS/Task Priority Requirements.	39.
6.25 Memory Requirements.	40.
6.26 Register Usage.	40.
6.27 Device Usage.	40.
6.28 Other Resources.	40.
6.29 Hardware Interfaces.	40.
6.30 Activity Diagram.	41.
6.31 Sequence Diagram.	43.
6.31.1 Sequence Diagram 1.	43.
6.31.2 Sequence Diagram – 2.	44.
7. ALGORITHMS FOR SCANNER AND PARSER.	45.
8. CONCLUSION.	52.
9. TOOLS.	54.
9.1 C Overview.	55.
9.2 C++ Overview	56.
9.3 LINUX.	58.
9.4 GCC 3.2.1 Compiler.	60.
9.5 Windows NT Overview.	63.
10 FUTURE PROSPECTS	66.
11 FUTURE ENHANCEMENTS AND RECOMMENDATIONS.	68.
12 REFERENCES.	71.



About The Company



1. ABOUT THE COMPANY



About Tata Elxsi

Design is at the core of our business. We adhere to high quality product design processes to offer cost-effective, time-to-market solutions. A highly motivated skilled workforce driven by strong design principles and ethical business practices not only ensure quality products and services time and again but also guarantee maximum returns for all our stakeholders.

Vision

Our Vision is to achieve global leadership in designing, by providing pioneering products, content and solutions for our customers

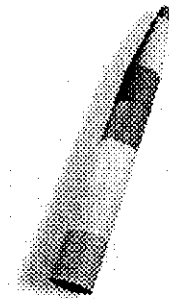
Business Area



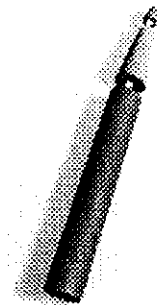
*Product Design
Services*



*Design And
Engineering Services*



Multimedia



Integration Services



Team Profile

The company's backbone comprising a 900 strong team of highly talented engineers, provide expertise in Product Designing, Design & Engineering and Systems Integration which spans multiple disciplines of VLSI design, Embedded software, Network Telecom, Multimedia, Storage, Visual Computing and High Performance Computing

Technology Focus

Automotive

- Embedded Automotive Controls
- Telematics/Infotainment
- Electronic Control Modeling
- Layered Competencies

Consumer & Office Products

- Services
- Success at work
- Layered Competencies

Network Telecom

- Broadband Infrastructure
- IP Networks
- Wireless Communications
- Convergence
- Layered Competencies

Semiconductors

- ASIC/SoC design
- FPGA design
- Chip Support Tools
- Board design
- Layered Competencies



Media

- Content Creation
- Content Distribution
- Content Hosting and Delivery
- Content Consumption
- Layered Competencies

Storage

- Storage Devices
- Storage Networking
- Storage Management

Scientific Applications

- Test & Measurement
- Medical Imaging
- Scientific Simulation
- Layered Competencies

Projects

Software Development

- Requirement Analysis.
- Architecture & Design.
- Code development & Testing.
- Project Management.
- Development Support such as Process, Tools & Configuration Management.

System Integration

- System Analysis and Design.
- Network Planning.
- Implementation, Customization, and Migration.
- Deployment, Integration, and End-to-End Testing.



- Network Optimization and Monitoring.
- IP (VoIP-H.323, SIP, MGCP), PSTN (SS7, TDM), ATM, Frame Relay,
- LAN/WAN.

System Enhancement And Maintenance (Legacy)

- Feature development/support to enhance installed base.
- Quick turnaround for client problems and client required features.
- Global Tier 3 and Regional Tier 2 support.
- Migration support.



2. INTRODUCTION

Commenting, C and C++ programs that will help the developer keep track of what he is trying to do. Proper indentation allows the developer to step through the code more easily. "I tend to err on the side of commenting less, rather than more". There appears to be a school of thought that has taken this one step further, and believes that comments are at best a necessary evil, and that good code should be self-evident enough to obviate the need for comments. Everybody should be writing code that's clean enough that you don't need to explain what it does. But the code only tells you what the code does; it doesn't tell you what the code was intended to do, what it ought to do, what it doesn't do, or why it looks the way it does.

Until now the developer writes the comments manually. The document generating tools like (Javadoc, Cocoon, doc++ etc) generates HTML documents with respect to the comments given by the developer in the source file. These document-generating tools do have certain specific way of commenting conventions. e.g. Every phrase in the comments and documentation should start with a capital letter and end with a dot and 2 spaces. GNU Coding Standards (the Emacs sentence commands will work), Documentation in javadoc/doc++ style: `/** */, ///
Example:`

```
/**  
 * Search a string in a buffer.  
 *  
 * @param buffer the buffer in which to search.  
 * @param string the string to look for.  
 * @return the index of the first occurrence.  
 */  
  
int findString(Buffer& buffer, String& string);
```

This allows the automatic generation of HTML documentation for the interface using doc++, comment everything in headers, this is where the interface is, and



where people usually look for help, leave two empty lines between functions in the *.cc files. It makes it clearer where the body of a function starts/ends. The developer has to follow these conventions during development.

To make the work easier and efficient, The **C/C++ API Specification Tool** is being developed; the developer need not follow any conventions. It is not necessary for him to manually type in the comments. The developed code will be compiled using Gcc compiler. The C/C++ specification tool is embedded into the code of the Gcc compiler. This tool will parse for Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls, Including the Parameters, Called functions, Global Data, Return Value and Exceptions, after parsing is done it searches for these data in the file being compiled. Then it places the header just above the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls. The headers will also depend on the Parameters, Called functions, Global Data, Return Value and Exceptions. This tool follows the basic conventions being used for commenting in C and C++ languages.



2.1 ORGANISATION OF WRITING

For the clarity of presentation and the project report is organized as follows. **Section 3** deals with the overview of the C/C++ API Specification Tool Architecture with special emphasis on Context in which the system will be used and the problem(s) that it solves, The services, i.e. Functionality, that the system provides, and the qualitative characteristics of these services. **Section 4** gives the need for c/c++ API specification tool. **Section 5** this section looks into the interconnectivity issues involved in the architectural design. This section also provides an insight into the functional requirement of the different components of the system and the design trade off being made for system performance optimization. **Section 6** Explains the working of the architecture in a systematic manner. **Section 7** illustrates the various algorithms being used for parsing. **Section 8** Conclusion. **Section 9** The tools used for implementation. **Section 10** discusses the future scope of the proposed architecture. **Section 11** the enhancements possible in the future releases of the tool. Finally the **Section 12** contains the References, which is a collection of materials referred by me at the development stage of the project.



Overview



3. Overview

Introduction

So far information about the program functionality to the user were based on manual typing of the headers in the source file, where the user traverses through the code and receives the information from the headers. It was the manual approach, which involves typing of headers into the source file from the developer's side. C/C++ API Specification tool adopts a different approach, a new era of generating headers into the source file. Using this tool the developer can put in the comments into his code with out much difficulty. There are various ways in which the document-generating tools are defined.

The automatic generation of document to developers source file; Headers defined by a developer organize documents and users receive information from these headers.

The C/C++ API Specification Tool can also be called as a document-generating tool with a small difference in the definition.

The automatic generation of headers to developers source file; The Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls being used by a developer organize the headers and users receive information from these headers.

Three elements thus integrate a would-be complete definition:

- Structured headers.
- Automatic header generation.
- High end formatting.



Basic idea

The basic idea is the same: users inquire a *“Header”* for information about the code, traverse through the code according to information from the headers i.e. the process of header generation from developer initiated manual typing to automatic generation. Instead of developer manual typing the headers into the code, the compiler generating the headers into the code had been introduced, the developer compiles the code once and sees that the headers are placed in the appropriate places. (e.g.- functions)

```
#include <stdio.h>

void main()
{
  int length(Char String [])

  { int i=0;len;

    while(String[i] !='\0')

    ++i;

    if (i==0)

    len=0;

    else

    len=i;

    print("length=",len); return(len);

  }
}
```

(Save this code and compile using GCC compiler
with the tool embedded into the compiler)



After compilation .

```
/* Funcion Name : length
   Description : To find the length
   Parameters : String[]
   Called function : none
   Global Data : none
   Return value : len
   Exception : none
*/

#include <stdio.h>

void main()
{
    int length(Char String [])
    {
        int i=0,len;
        while(String[i] !='\0')
            ++i;
        if (i==0)
            len=0;
        else
            len=i;
        print("length=",len); return(len);
    }
}
```



Locating

Currently locating control flow in the code is a major problem for users. Even though good document generation tools exist, the quality of information found still depends on the developers comments. C/C++ API specification tool promises to remedy this by the concepts of automatic generation of headers and formatting and by shifting the active role to the compiler.

Focusing

The developer is required to compile the code using the compiler with which the C/C++ API specification tool is being embedded for the automatic generation of headers, and also place the headers into the source file, at appropriate places.

Customization

With C/C++ API specification tool the developer is required to install his/her preferences of GCC compiler version with respect to the operating system he/she uses.

Available documentation tools

This section is provided with a list of various documentation tool available for documenting code developed on C/C++ language:

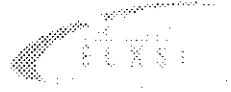


Current tools

Tools	Output	Languages	Comments
ABC+	HTML & RTF	C/C++	Before declaration. Claims to be capable of understanding a large number of commenting styles.
Autoduck	HTML & RTF	C, C++, Assembly, and Basic	Apparently this tool has been around for quite a while but is not used much these days.
c2man	Produces LinuxDoc, SGML	C only	Strict and unique to this tool.
CcDoc	HTML (like old Javadoc)	C++	Javadoc
CC-RIDER	HTML, RTF, (also wall-sized charts)	C/C++	?
Cocoon	HTML	C++	Before object with section name with various conventions unique to this tool.
Cxref	LaTeX, HTML, RTF, and SGML (Linuxdoc DTD)	C only; No plans for C++	Unique to this tool.
Cxx2HTML	HTML	C++ (Header files)	Unique to this tool.
Doc-o-Matic	HTML, HTMLHelp, WinHelp, others?	C/C++ and Pascal	flexible, claims to support Javadoc style
Doc++	HTML, LaTeX	C/C++/Java/IDL	Similar to Javadoc, though some special Doc++ markup is necessary.
DocBuilder	HTML, LaTeX	C/C++, Pascal, Delphi	Flexible, includes support for Javadoc and Doc++ style comments, but has its own style too.
DocJet	HTMLHelp, HTML, WinHelp, RTF	C/C++, IDL, Visual Basic, Java	Flexible, supports Javadoc, thought some preprocessing required.
Doxygen	HTMLHelp	C/C++	Configurable (some special "regular expressions" required?)
George	WinHelp, HTML based help, mif, RTF	C/C++	Configurable (some special "regular expressions" required?)
gtk-doc	DocBook (Apparently)	C	??
Imagix 4D	Text, RTF, HTML	C/C++	??
Kdoc	HTML, LaTeX, "Man" pages	C++, IDL (for the KDE lib's")	Javadoc
MkHelp for C++	HTML or RTF (and soon DHTML)	C++	??



Object Manual	HTML, RTF, MIF, and "man' (nroff/troff)"	C++	Javadoc
Object Outline	HTML, RTF, and WinHelp	C/C++	flexible
Perceps	flexible	C/C++	Unique to PERCEPS (e.g. //: or //!)
ROBODoc	HTML, ASCII, AmigaGuide, LaTeX, or RTF format	Assembler, C, C++, Java, Perl, LISP, Occam, Tcl/Tk, Pascal, Fortran, shell scripts, and COBOL,	Complicated and unique to ROBODoc
Scandoc	HTML but configurable with a template file.	C++	superset of Javadoc
Sdoc	Lout, LaTeX, HTML, and plain text	C, C++, Objective C, Pascal, Modula, Oberon, Perl, Tcl/Tk, Objective Caml, Pike, Python	Unique to Sdoc
Surveyor	HTML and RTF	C/C++	More of a code-analysis tool than a doc tool.
The Automatic Documentation System	HTML	C and SQL scripts	Code must contain specially formatted comments
Weasel	DocBook	C/C++	??



Need for C/C++ APP Specification Tool





4. NEED FOR C/C++ API SPECIFICATION TOOL

4.1 Purpose

To design the C/C++ API Specification tool as a part of GNU projects. It also includes the designing of existing documenting tools. The whole architecture is based on the parsing technique, which involves the automatic generation of headers into the source code of the developer (Languages – C & C++). The automatic generation of headers is done based on the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls being used by a developer and also the commenting conventions being used in C & C++ languages. The scanner tracks the token from the source file. Parsing, which is a technique used to identify the category of the token depending upon the developer's implementation, Headers of each category provided by the Tool and Compiler, finds the relevant category and places the headers into the source file. The whole process saves the developer effort of providing headers to his code.

4.2 Document Conventions

GNU (GCC) standards are used

4.3 Product Scope

The dominant paradigm of generating documents and headers for the source file is the developer typing the headers and generating a **LaTeX**, **HTML**, **RTF**, and **SGML** file with respect to the headers. In this model of generating documents, a user actively gets information from the **LaTeX**, **HTML**, **RTF**, or **SGML** documents. so in this model the availability of the relevant header information totally depends upon how efficiently developer frames the headers. The way technology is growing, writing headers manually has become a very



cumber some task. Keeping this, limitation of existing document generating tool, in mind I have proposed a C/C++ API Specification Tool for automatic generation of header into the source file. In this model the parser announces the availability of certain category, and the C/C++ API specification tool generates the headers, and places the headers systematically in the source file.

The whole architecture has been designed to achieve the following goals:

- To act as a bridge between the developer and user.
- To provide automatic generation of headers to the code being developed .
- To generate headers in a systematic and a conventional manner.

The whole objective of the C/C++ API Specification Tool is to save time and effort of the developer and user understanding the code developed. The C/C++ API Specification Tool generates headers while in synchronization with the standard commenting conventions of C/C++ languages.

Thus the C/C++ API Specification Tool result in following benefits:

1. customized headers.
2. automatic header generation.
3. Generation of headers into the source file.
4. Formated headers.



System Requirement Specification



5. SYSTEM REQUIREMENT SPECIFICATION

This project has been done for TATA ELXSI an SEI-CMM Level 5 company. This document describes about the function headers generated as comments.

5.1 Purpose

This project deals with generating headers for code being developed and relies on the compiler to do its job. It calls a part of the compiler to compile the declarations, ignoring the member implementation. It builds a rich internal representation of the functions and "use" relationships, then generates the headers and also picks up user-supplied documentation from documentation comments in the source code. Which helps the third party to understand the code developed.

5.2 Scope

The scope of the project is that it parses the declarations and documentation comments and produces a corresponding set of commented lines describing (by default) the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls, Including the Parameters, Called functions, Global Data, Return Value and Exceptions. This generation of headers to the code being developed is not specific to one platform. It can be generated on platforms like Linux and Solaris.

5.3 Definitions, Acronyms, Abbreviations

-None-

5.4 References

- a) For reference books
Alfred V. Aho, "Compilers: Principles, Techniques, and Tools ",
Hardcover, 1980
- b) For Web sites
<http://www.java.sun.com>
<http://www.epaperpress.com/>



5.5 General Description

5.5.1 Product Perspective

This will be more in similar to the JavaDoc, which is used in generating comments when a java code is compiled. This tool will be generating headers, which will be kept in the source file. Generating in the form of comments.

5.5.2 Product Function

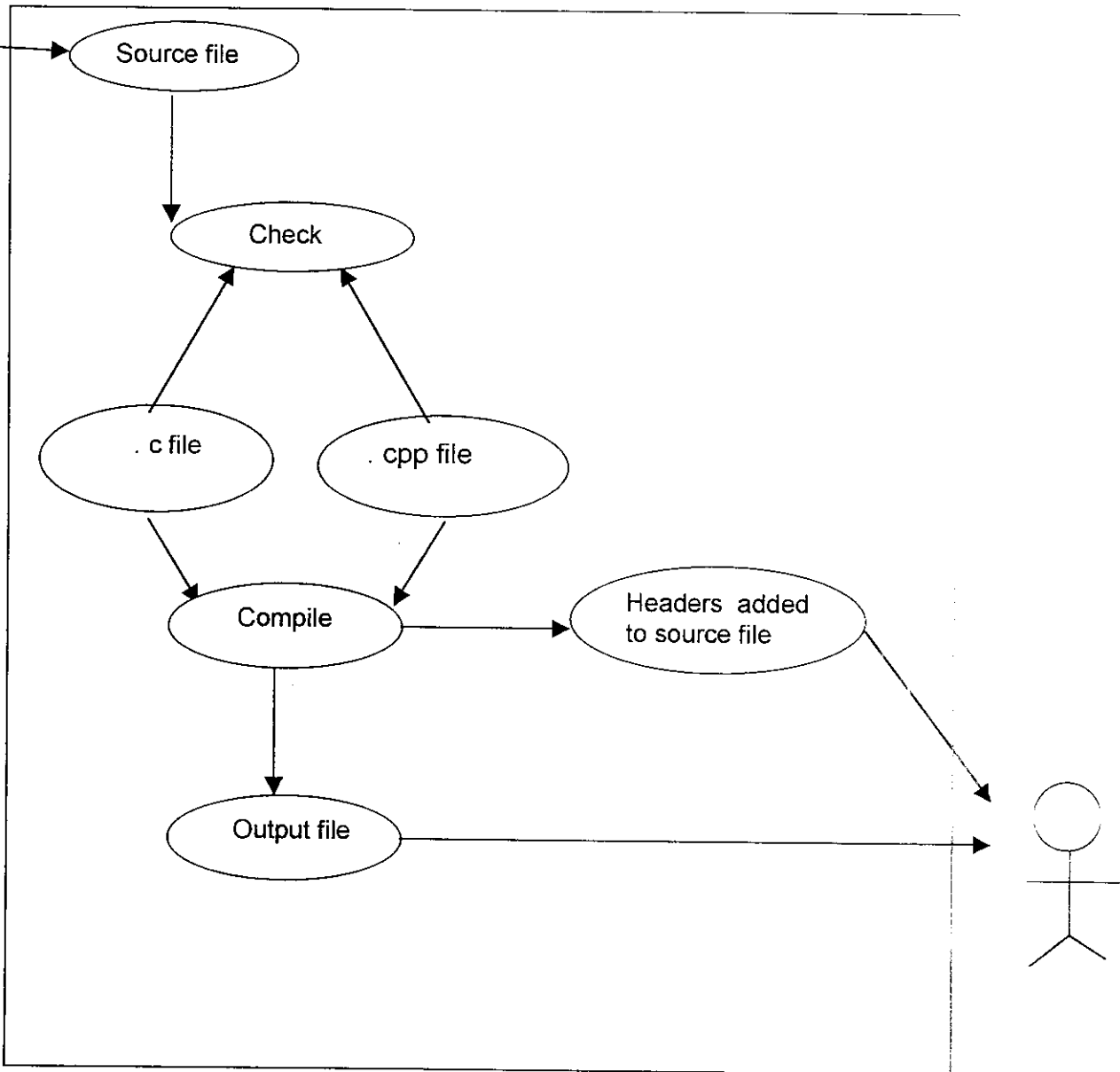
The tool being developed helps the user to trace out the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls being used. The tool will also list out the parameters, called functions, global value, return value, and exceptions. The tool will place these information's in the required places in an orderly manner as comments.

5.5.3 User Characteristics

The user should be specifying a c / cpp file. The user may compile the file on any platform he/she is comfortable with.

The use case diagram indicates the various use cases in the system:

5.5.3.1 Use Case Diagram





5.6 Operating Environment

Linux and Solaris

5.7 User Interface

Command line interface.

5.8 Design Constraints

5.8.1 Hardware Limitations

RAM	128MB
HD	4 GB
Processor	P II

5.9 General Constraints

There should be files and tables generated by the scanner containing all the information, for identifying the Classes, Functions, Variables, Reserved words, Macros, Enums, Unions, Structs, Typedefs, System calls, Parameters, Called functions, Global Data, Return Value and Exceptions. Identification would be done using sorting and searching algorithms.

5.10 Functional Requirements

The system developed with regards to the client's request to generate headers which would help the third party to trace out why the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls is being used, what are the parameters used, which function is being called, what value does it return and also to trace out the exceptions used inside the function.

5.11 List Of Inputs

A .c/.cpp file developed by the user will be used as the input for the tool.



5.12 Performance Requirements

The performance of the tool depends on the length of the file being inputted for compilation. The headers will be generated as and when the file is being compiled.

5.13 Safety Requirements

Not Applicable

5.14 Software Quality Attributes

Not Applicable

5.15 Business Rules

Not Applicable

5.16 Other Requirements

Not Applicable

5.17 Risk Analysis

Not Applicable



High Level Design



6. HIGH LEVEL DESIGN

6.1 Introduction

This template describes the C/C++ API Specification Tool. It also includes the designing of the Tool, This template provides an excellent high-level design document for the development team. The whole architecture is based on the GCC compiler, which generates headers for Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls. The automatic generation of headers is based on the programming languages, C and C++. The tool tracks the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls by the parsing and generates the headers when the user compiles the program. The whole process helps the third party user to understand the code developed.

6.2 Purpose

The purpose of this document is to provide the developers a basic framework to understand the C/C++ API specification tool architecture and the working of the tool; The tool modeled in the lines of the GCC compiler. This document also put light on issues of data flow and the inputs and outputs of the C/C++ API specification tool. It also incorporates the algorithm used by the C/C++ API specification tool.

6.3 Scope

This template talks about the C/C++ API specification tool architecture. It also covers the porting of the C/C++ API specification tool into the GCC compiler. These templates also throw light on the algorithm used by the C/C++ API specification tool.



6.4 Conventions

Referred GNU Project standards (GNU is a recursive acronym for "GNU's Not Unix"; it is pronounced "guh-NEW").

6.5 Document Status

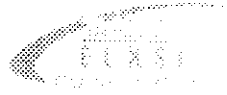
This section is created after finalizing the specification tool architecture and its portability with the GCC compiler. The algorithms to be used by the C/C++ API specification tool are also finalized. All the other sections like data flow diagram, activity diagram and, sequence diagram, flow chart are also finished and included in the template. This section elaborates the working of C/C++ API specification tool; it's algorithm and provides a clear picture of data flow.

6.6 System Description

6.6.1 System Requirements

The various requirements of C/C++ API Specification Tool are as follows:

- a) The first and the foremost requirement of the C/C++ API specification tool is the GCC 3.2.1 compiler.
- b) The C/C++ API specification tool requires a C/C++ code developed by the developer.
- c) To compile the code the tool requires an operating system or otherwise the developer should specify the environment.



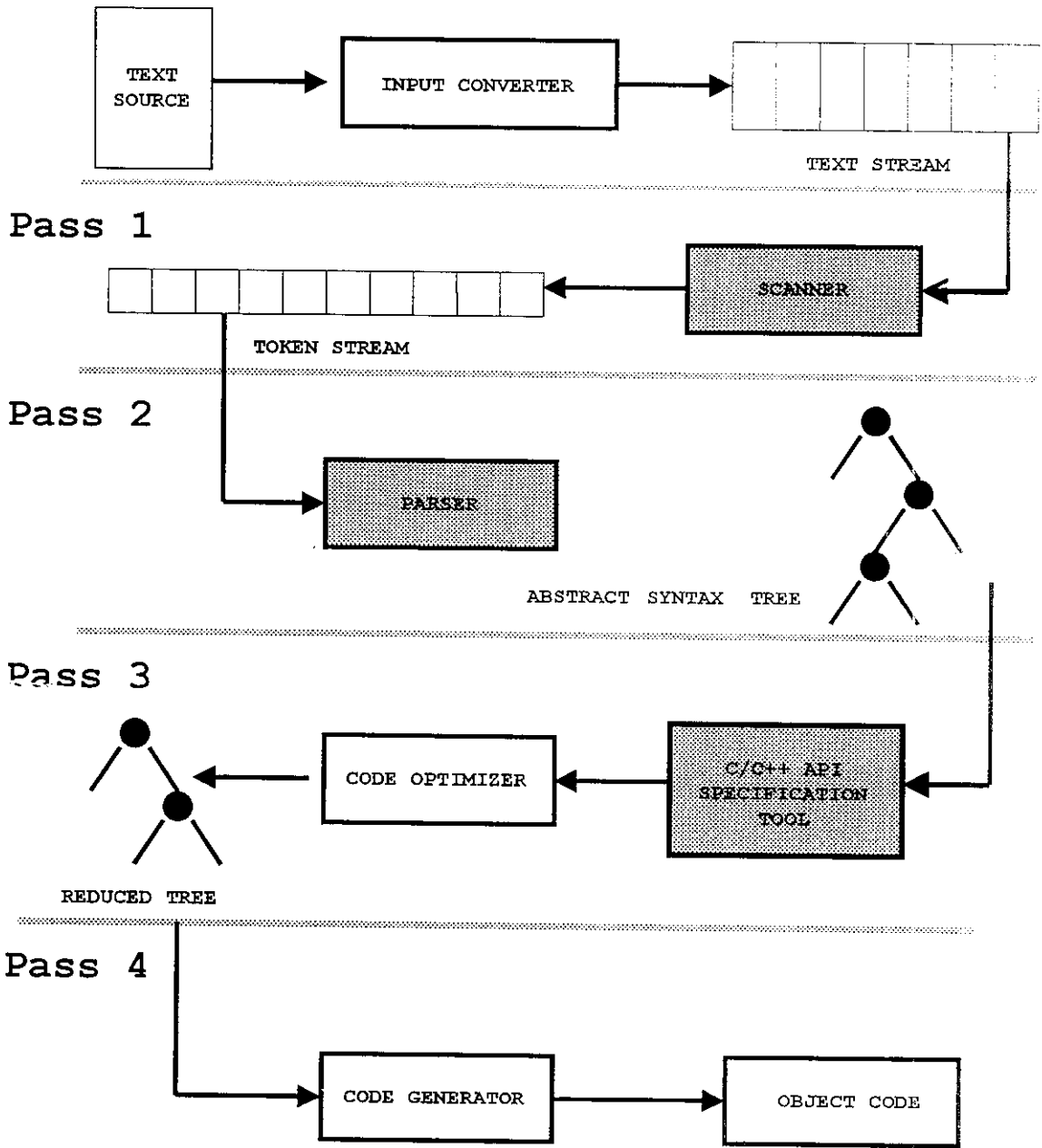
6.6.2 System Context

6.6.2.1 Target Environment

The C/C++ API specification tool is designed to service both the third party user and the developer. So the requirement and the limitations are kept in mind while designing the architecture. The compiler should be configured with respect to the operating system being used by the user. The input is the C/C++ code developed by the user. The code is compiled with the ported C/C++ API specification tool. The tool helps the user to trace out the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls being used. The tool will also list out the parameters, called functions, global value, return value, and exceptions. The tool will place this information's in the required places in an orderly manner as comments.



6.6.3 External Architecture





Scanner (Lexical Analysis)

This module has the task of separating the continuous string of characters into distinctive groups that make sense. Such a group is called a token. A token may be composed of a single character or a sequence of characters. Examples of tokens are identifiers or words that have a special meaning in the language like begin, end, etc. numbers and special character sequences. The Scanner also eliminates from the text source the comments that may exist in the program and white space. The output of the scanner gives the input to the next module that is the parser, or the Syntactical Analyzer.

Parser

It was the scanner's duty to recognize individual words, or tokens of the language. The scanner does not, however recognize if these words have been used correctly. The main task of the parser is to group the tokens into sentences, that is, to determine if the sequences of tokens that have been extracted by the scanner are in the correct order or not. In other words, until the parser is reached the tokens have been collected with no regard to the whole context of the program as a whole. The parser analyzes the context of each token and groups the tokens in declarations, statements, and control statements. In the process of analyzing each sentence, the parser builds abstract tree structures.

C/C++ API specification tool

The parser groups the tokens. The C/C++ API specification tool helps the user to trace out the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls from the group of tokens. The tool will also list out the parameters, called functions, global value, return value, and exceptions. The tool will place these information's in the required places in an orderly manner as comments.



6.7 Architecture Goals

Architecture of the proposed C/C++ API specification tool is as follows:

a) Human Communication

If a program cannot be understood by third party user. It is difficult to verify and it cannot be maintained or modified, Even if the program is still clear to the author. Programmers dislike writing exclusive comments and tend to avoid them. To help the programmers and the users the comments are automatically generated when the programmed is compiled.

b) Efficiency

Efficiency has been the most exclusively overemphasized topic in the history of programming. Is the justification of producing a program that executes twenty times as fast as the competition, but fails in the half the runs? Efficiency is important after, not before, reliability.

c) Machine Independent.

The designed system is C/C++ API specification tool and is not biased to a particular platform. So it can sits on any platform(Restricted to Windows and Mac).

6.8 General specifications

The whole C/C++ API specification tool revolves around the centralized entity called **Parser**. The parser makes sure that the user gets the relevant tokens. Parser has a sub component called **Symbol Table**. Though the parser determined the correct usage of tokens, and whether or not they appeared in the correct order, it still did not determine whether or not the program said anything that made sense. This type of checking occurs at the semantic level. In order to perform this task, the compiler makes use of a detailed system of lists, known as the symbol tables



6.9 Scope of Implementation

6.9.1 Limitations

The implementation of the proposed C/C++ API specification tool involves following Limitations:

- a) The user is expected to specify a C/C++ code for compilation and execution.
- b) The user is expected to choose the operating system.
- c) The GCC compiler should be installed and checked before specifying the C/C++ code.

6.10 Design Trade-off Decisions

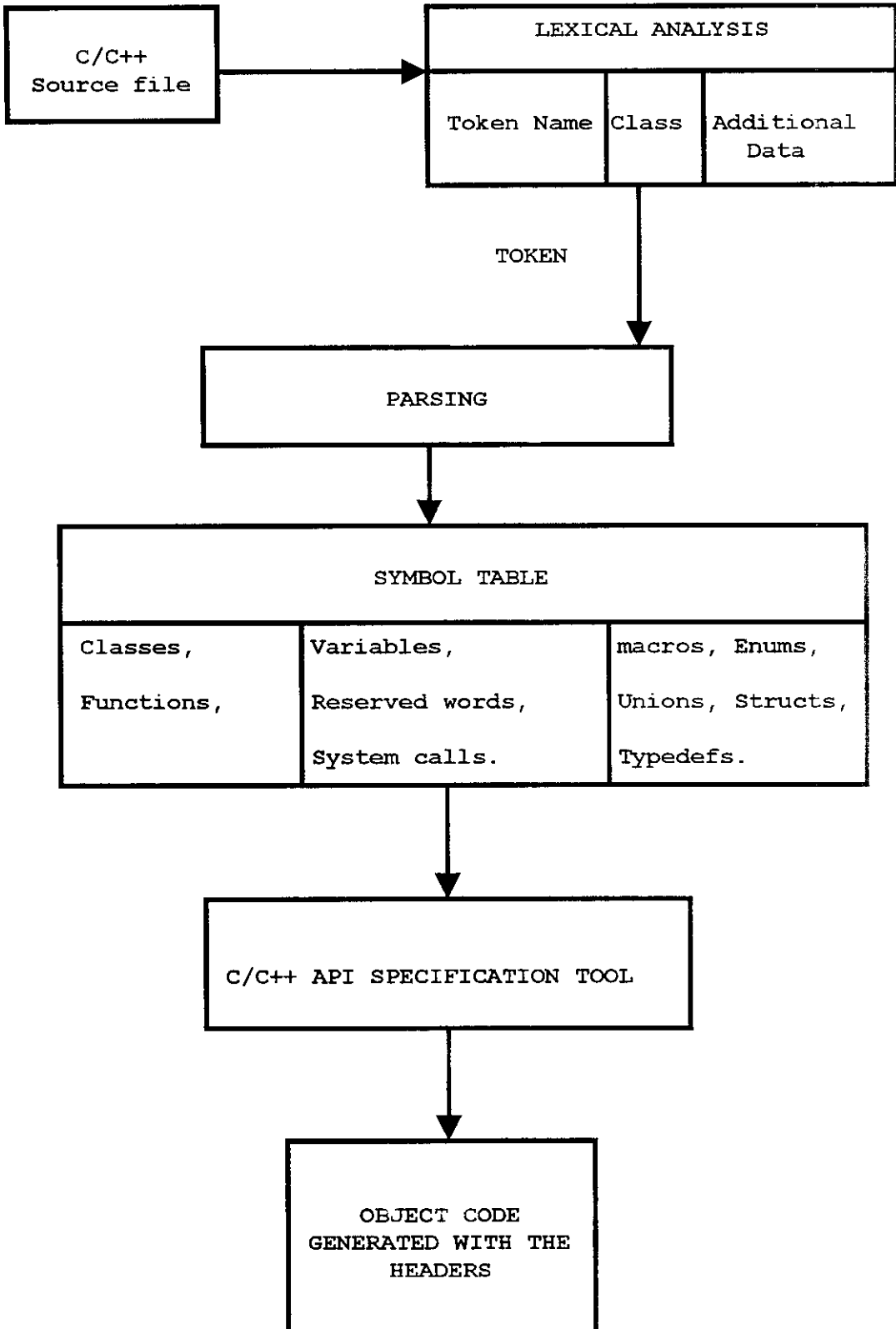
- a) During the installation phase each user will select the operating system he wants. If in case user doesn't do that then a default operating system of LINUX will be selected.
- b) The compilation time depends upon the size of the file specified by the user.
- c) The headers will be generated for Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls
- d) The generated headers will also contain Parameters, Called functions, Global Data, Return Value and Exceptions.

6.11 Test & Debug Requirements

- Tools
- Test case template



6.12 Internal Architecture



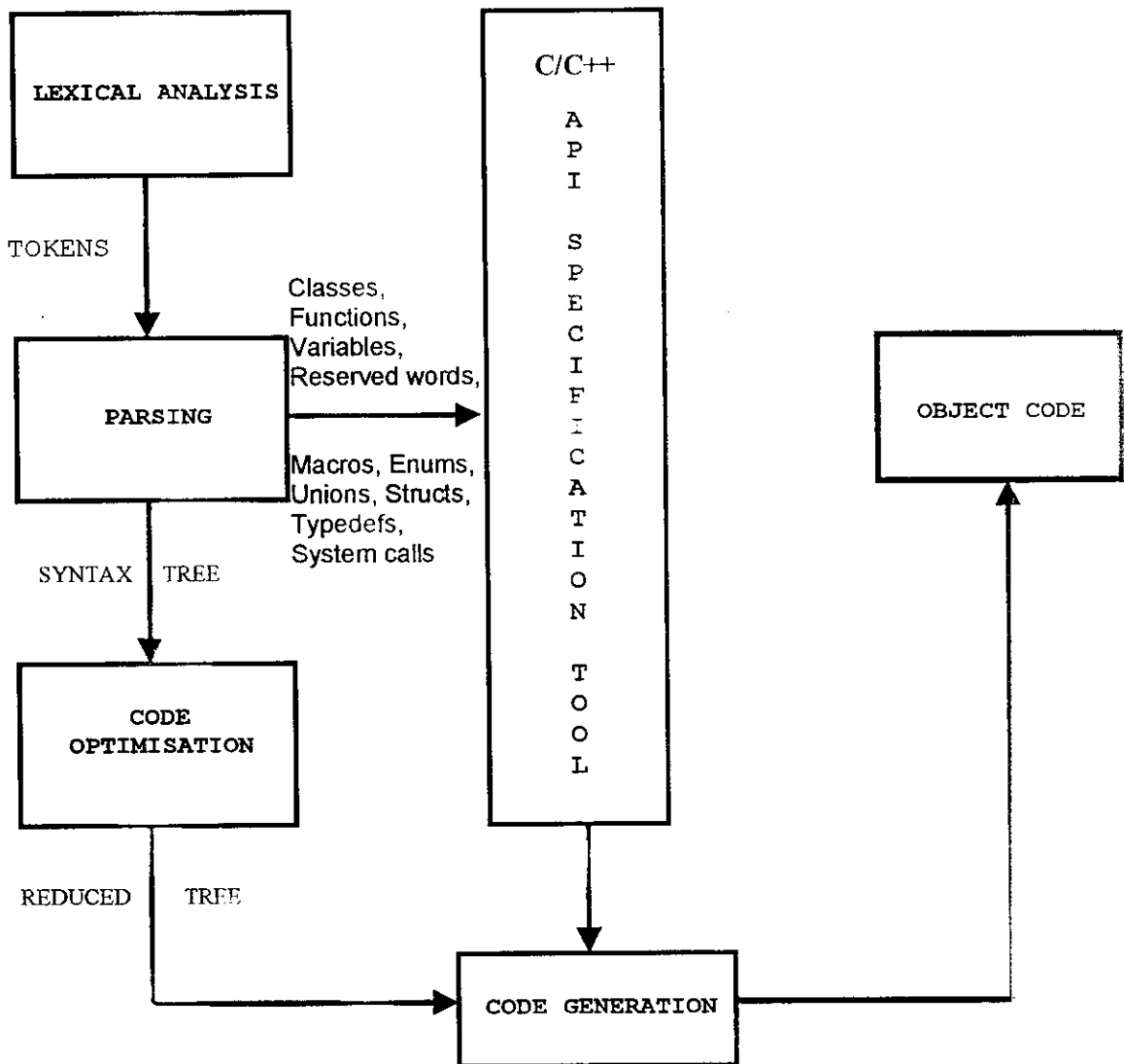


6.13 Interconnectivity

The interactions between the modules shown in I/O Diagram.

6.14 Input / Output

6.14.1 I/O DIAGRAM





6.14.2 I/O Diagram Representations

Field	Description
Tokens	Separating the continuous string of characters into group that make sense.
Functions	Collection of user defined and system defined function names.
Reserved Words	Words that can't be used as identifiers.
System calls	Predefined stmt's that are used to interact with the hardware device.
Classes	A user Defined data type which holds both the data and functions.
Variables	An object that may take on values of the specified type.
Macros	A substitution string that is placed in a program.
Enums	Enumeration data types.
Unions	Similar to structures, differs in the way data is stored and retrieved.
Structs	Heterogeneous data types can be grouped to form a structure.
Typedefs	New data items which are equivalent to the existing data types.



6.15 Functional Specifications

The system developed with regards to the client's request to generate headers. Would help the third party to trace out why the Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls are being used, what are the parameters used, which function is being called what value does it return and also to trace out the exceptions used inside the function.

6.16 Requirements (From rest of the system)

- a) A C/C++ file is needed as input to the system.
- b) The Gcc compiler should be installed.
- c) The parsed text of token should be passed on to the C/C++ API specification tool.
- d) The tool takes the functions, reserved words, system calls as input from the parsed text.

6.17 Performance Specifications

The performance of the tool depends on the length of the file being inputted for compilation. The number of Classes, functions, variables, reserved words, macros, Enums, Unions, Structs, Typedefs and system calls used in the program. The headers will be generated as and when the file is being compiled.

6.18 Timing/Sequencing/Synchronization/Buffering

Not Applicable

6.19 System Specifications

6.19.1 OS Calls

Not Applicable



6.19.2 Dependencies

The output of the C/C++ API specification tool depends upon the parsing done by the GCC compiler

6.19.3 Initialization

Not Applicable

6.19.4 Termination

Not Applicable

6.19.5 Error Messages

Scanner errors, Some of the most common types here consist of illegal or unrecognized characters, mainly caused by typing errors. A common way for this to happen is for the programmer to type a character that is illegal in any instance in the language, and is never used. Another way for this type of error to happen is to mistype an operator, like accidentally typing ";" instead of "=". Finally and quite commonly, another type of error that the scanner may detect is an unterminated character or string constant. This happens whenever the programmer types something in quotes, and forgets the trailing quote. Again, these are mostly typing errors.

The second class of errors is **syntactic in nature**, and is caught by the parser. These errors are among the most common. The really difficult part is to decide from where to continue the syntactical analysis after an error has been found. What happens if the parser is not carefully written, or if the error detection and recovery scheme is sloppy, the parser will hit one error and "mess up" after that, and cascade spurious error messages all throughout the rest of the program. In the case of an error what one would like to see happening, is to have the compiler skip any improper tokens, and continue to detect errors without generating error messages that are not really an error but a consequence of the first error. This aspect is so important that some compilers are categorized based



on how good their error detection system is. Anyone familiar to programming has certainly encountered this problem before in his or her own experience.

The third type of error is **semantic in nature**. The semantics that are used in computer languages are by far simpler than the semantics that are used in spoken languages. This is the case because in computer languages everything is very exactly defined, there are no nuances implied or used. The semantic errors that may occur in a program are related to the fact that some statements may be correct from the syntactical point of view, but they make no sense, and there is no code that can be generated to carry out the meaning of the statement.

6.20 Other Specifications

Not Applicable

6.21 Configuration Information

6.21.1 Compile Time Configuration

Not Applicable

6.21.2 Runtime Configuration

Not Applicable

6.22 Tools Specifications

Not Applicable

6.23 Resource Specifications

Not Applicable

6.24 MIPS/Task Priority Requirements

Not Applicable

6.25 Memory Requirements

Not Applicable



6.26 Register Usage

Not Applicable

6.27 Device Usage

Not Applicable

6.28 Other Resources

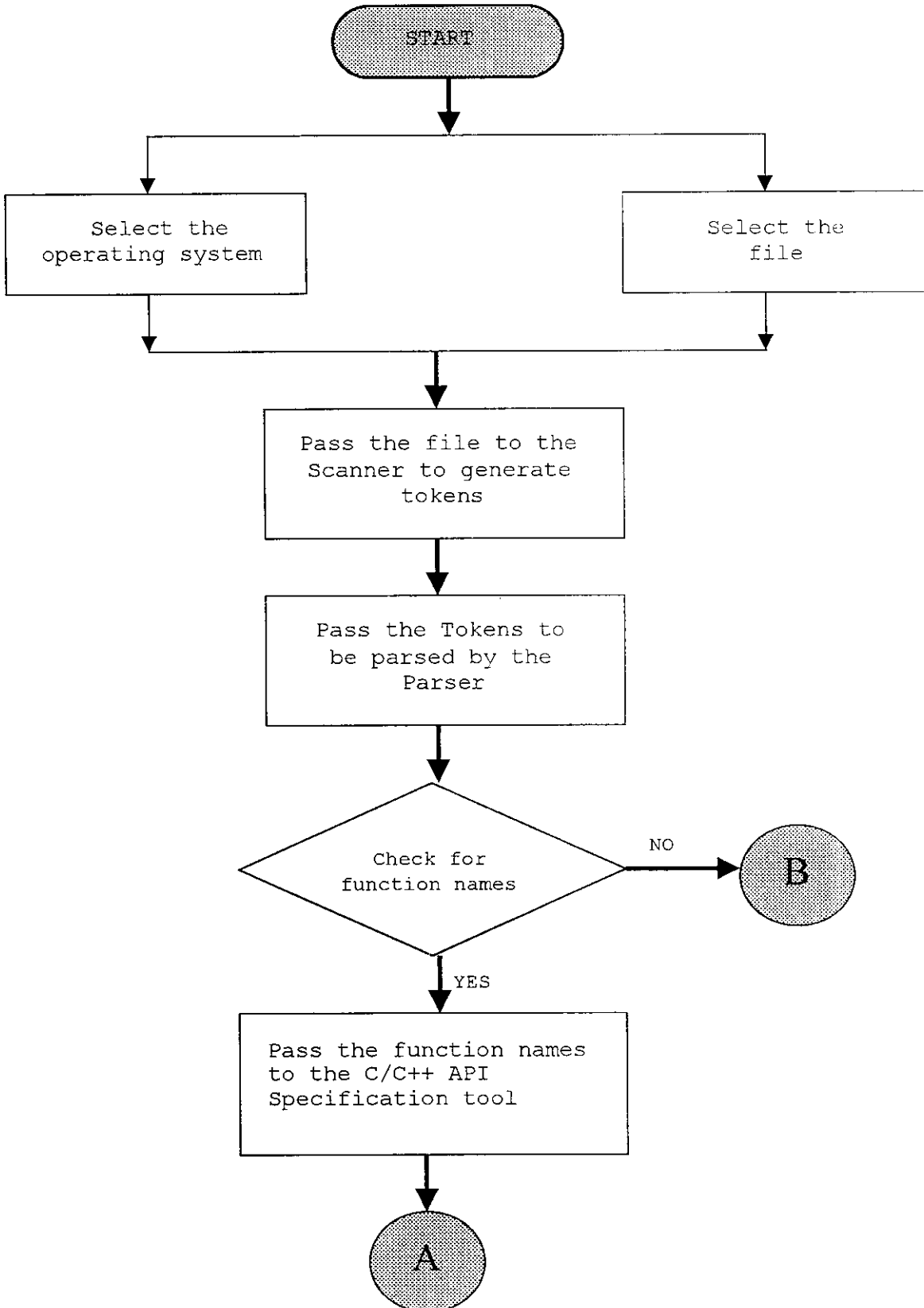
Not Applicable

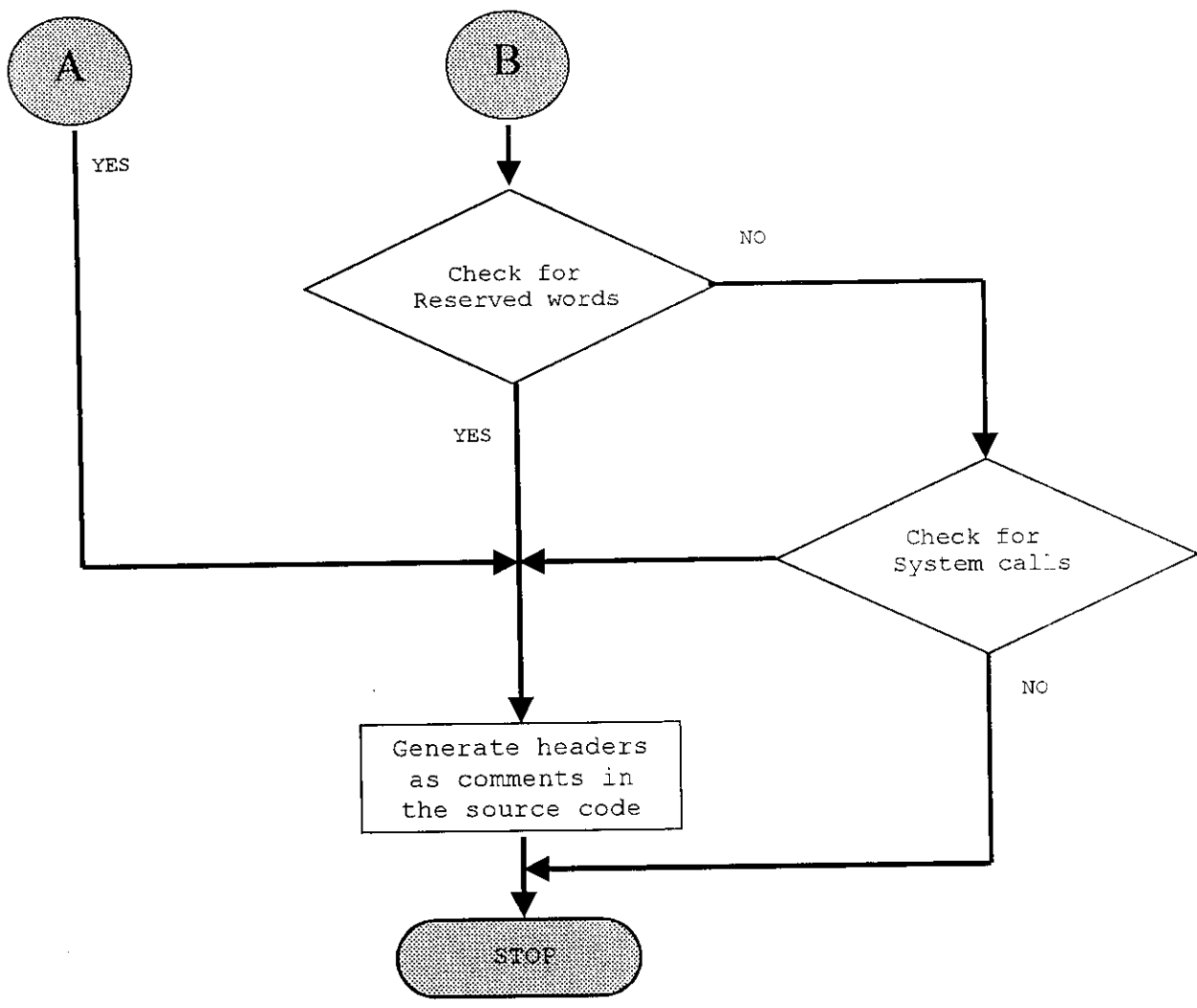
6.29 Hardware Interfaces

Not Applicable



6.30. Activity Diagram

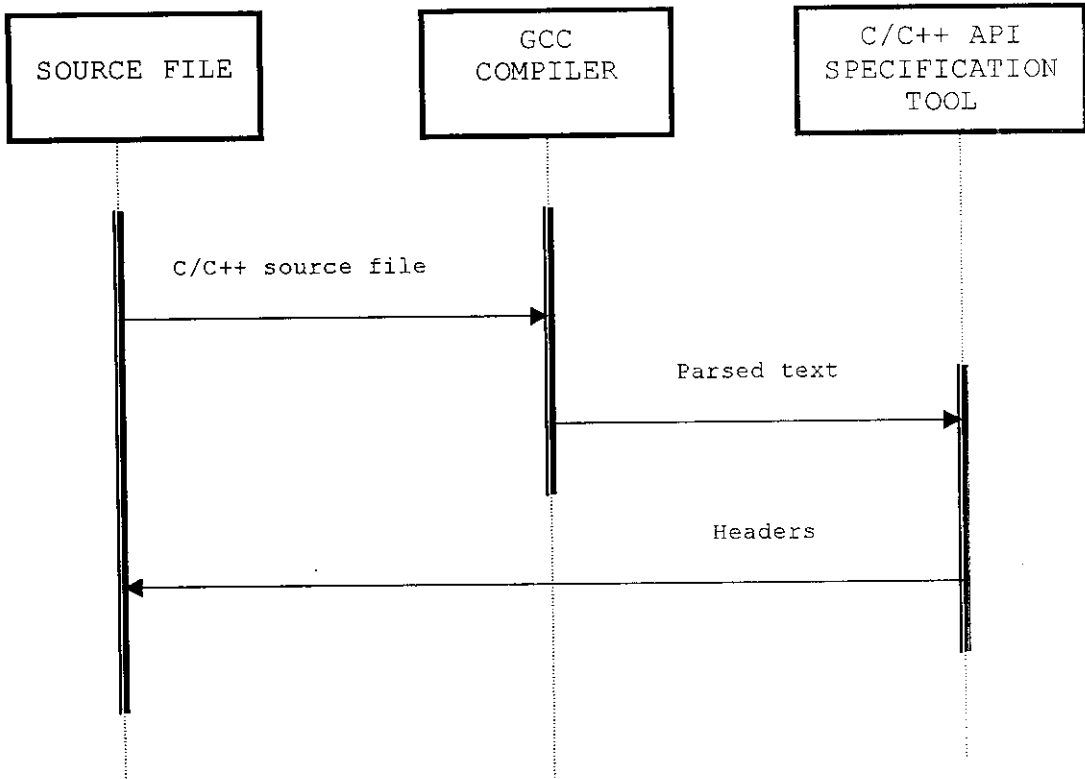






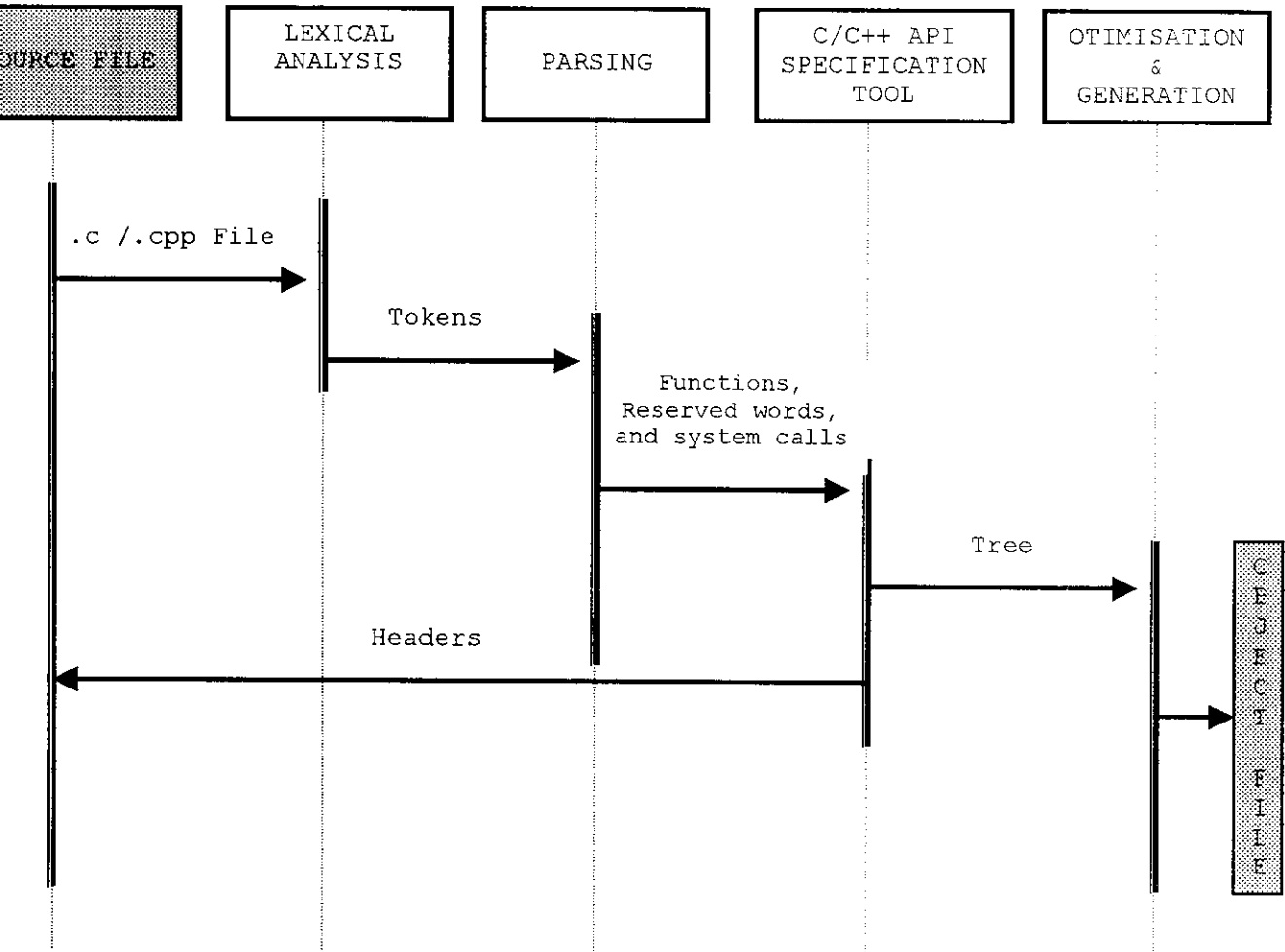
6.31 Sequence Diagram

6.32 Sequence Diagram 1



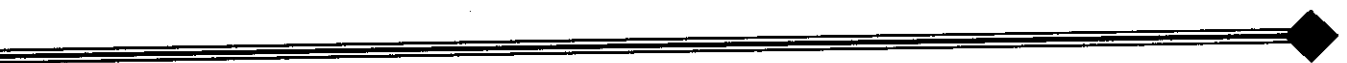


6.33 Sequence Diagram - 2





Algorithms





7. ALGORITHMS FOR SCANNER AND PARSER

A compiler accepts a sequence of *characters* in some alphabet, and *parses* or recognizes the sequence as defining a valid program in the compiler's source language. In general, parsing involves recognizing which sub-sequences of the input form recognizable units in the language, like assignment statements, or expressions. The first step in the recognition process is usually the replacement of long strings of characters with objects, called *tokens*, which are fixed-sized codes for the corresponding input string. The rest of the compiler then processes the sequence of tokens, and need not examine the individual characters making up each token's string.

Lexical analysis is the name given to the part of the compiler that divides the input sequence of characters into meaningful token strings, and translates each token string to its encoded form, the corresponding token. Tokens are fairly simple in structure, allowing the recognition process to be done by a simple algorithm. Examples of tokens might be:

- A C name -- a sequence of letters and digits beginning with a letter
- A C integer constant -- a sequence of digits
- A string -- a sequence of characters other than "" surrounded by "" characters
- A comment -- a sequence of characters surrounded by the sequences /* and */
- An operator – a single symbolic character from the set: +-*/&%|[]{}()<>:

During lexical analysis, the source program is treated as a sequence of tokens, separated by optional whitespace. Lexical analysis ignores the sequence of tokens, leaving the decision as to whether tokens occur in a meaningful sequence up to the rest of the parsing portion of the compiler.

Lexical analysis replaces each token string with its encoded form:

- NUMBER <integer values>



- NAME <sequence number unique to this name> (every occurrence of this name should get this same number)
- STRING <number for this occurrence of this string>
- op (The internal code for this operator)

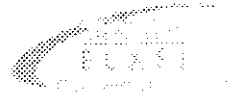
Here, the first column represents the value of a single byte (the "token type") in the output sequence, where NUMBER, NAME, STRING, and each value of "op" are chosen to be different. The designation <description> describes the value of a single integer which follows the token type byte, while parenthesized material is just commentary, not represented in the output of Lex. Note that comments are simply discarded – they do not give rise to any token in the output.

Algorithm: Scanner

The algorithm for lexical analysis is based on the concept of a Finite State Machine, a mathematical model of a simplified computer. An FSM maintains an internal state, and performs actions as it moves from one internal state to another. These moves, called *transitions*, take place when the machine in state X reads a character C from the input. The combination (X,C) selects a particular transition, to a target state T. During the transition, the machine may perform an *action* A. The entire machine is defined by giving the function $d(X,C) = (T,A)$, and specifying which state the thing *STARTs* in. As a special case, we will also allow a transition that reads no character, but performs an action, and moves to a new state. The character ϵ will be used to represent the absence of a character. If it appears in a table defining the $d()$ function, its transition will be performed only if the next input character does NOT match any character explicitly shown in $d()$ entries for the same state. The $d()$ table can be stored as a pair of arrays indexed by State and Character. Actions can be encoded, by giving each possible action a unique number, and performing

```
switch (Action[State][Character] { case 1: <first action>; break; case 2: ... }
```

The program can now execute



State = Next[State][Character];

and repeat these steps, until there is no more input. Action code can:

- Convert the digits which form an integer, one by one: $\text{num} = \text{num} * 10 + \text{val}[\text{Character}]$;

where val[] is a pre-initialized array which maps digits to their numeric values.

- Record the characters, which form a string or name, and enter the complete item into a data structure from which their sequence number can be determined.
- Record the val [Character] of an operator symbol.

When the end of a token string is recognized, the associated action routine usually produces the required output. This may involve recording the final form of the token string in an internal data structure, and producing the required token-type byte, and accompanying "value" information. In most languages, the recognition of "end-of-token" is triggered only by the observation of an input character that can't be incorporated into the current token. The I transition can be used, to perform the end of token action, and then proceed to process the input character that triggered that transition.

A lexical analyzer must be prepared to process any possible character, in any state. In the ASCII code, there are some 256 possible characters, and it becomes tedious to fill in the table entries for all 256 for each of perhaps 10 states. A simplification can ease this burden: As each character is read, a reference to an array, indexed by character, can retrieve the character's "character class code", $\text{Class} = \text{CI}[\text{Character}]$. The other tables can then be referenced, using Class instead of Character.



ALGORITHM : Parser

Construction of LALR parser requires the basic understanding of constructing an LR parser. LR parser gets its name because it scans the input from left-to-right and constructs a rightmost derivation in reverse.

A parser generates a parsing table for a grammar. The parsing table consists of two parts, a parsing action function **ACTION** and a goto function **GOTO**.

An LR parser has an input, a stack, and a parsing table. The input is read from left to right, one symbol at a time. The stack contains a string of the form $s_0 X_1 s_1 \dots X_m s_m$ where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called a state. Each state symbol summarizes the information contained in the stack below it and is used to guide the shift-reduce decision.

The function **ACTION** stores values for s_m that is topmost stack element and a_i that is the current input symbol. The entry $\text{ACTION}[s_m, a_i]$ can have one of four values:

1. shift s
2. reduce $A \rightarrow B$
3. accept
4. error

The function **GOTO** takes a state and grammar symbol as arguments and produces a state. Somewhat analogous to the transition table of a deterministic finite automaton whose input symbols are the terminals and nonterminals of the grammar.

A *configuration* of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpanded input:

$$(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

The next move of the parser is determined by reading a_i , the current input symbol, and s_m the state on top of the stack, and then consulting the action table



entry $ACTION[s_m, a_i]$. The four values mentioned above for action table entry will produce four different configurations as follows:

1. If $ACTION[s_m, a_i] = \text{shift } s$, the parser executes a shift move, entering the configuration

$$(s_0 X_1 s_1 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$$

Here the configuration has shifted the current input symbol a_i and the next state $s = GOTO[s_m, a_i]$ onto the stack; a_{i+1} becomes the new current input symbol.

2. If $ACTION[s_m, a_i] = \text{reduce } A \rightarrow B$, then the parser executes a reduce move, entering the configuration

$$(s_0 X_1 s_1 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$$

where $s = GOTO[s_{m-r}, A]$ and r is the length of B , the right side of the production. Here the first popped $2r$ symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r} . The parser then pushed both A , the left side of the production, and s , the entry for $ACTION[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move. Specifically, $X_{m-r+1} \dots X_m$, the sequence of grammar symbols are popped off the stack and will always match B , the right side of the reducing production.

3. If $ACTION[s_m, a_i] = \text{accept}$, parsing is completed.
4. If $ACTION[s_m, a_i] = \text{error}$, the parser has discovered an error and calls an error recovery routine.

The LR parsing algorithm is simple. Initially the LR parser is in the configuration $(s_0, a_1 a_2 \dots a_n \$)$ where s_0 is a designated initial state and $a_1 a_2 \dots a_n$ is the string to be parsed. Then the parser executes moves until an accept or error action is encountered.



I mentioned earlier that the GOTO function is essentially the transition table of a deterministic finite automaton whose input symbols (terminals and nonterminals) and a state when taken as arguments produce another state. Hence the GOTO function can be represented by a graph (directed) like scheme, where each node or state will be a set of items with elements that are productions in the grammar. The elements comprise the core of the items. The edges representing the transition will be labeled with the symbol for which the transition from one state to another is predetermined.

In the LALR (*lookahead-LR*) technique, LR items with common core are coalesced, and the parsing actions are determined on the basis of the new GOTO function generated. The tables obtained are considerably smaller than the LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by LALR grammar.



Conclusion



8. CONCLUSION

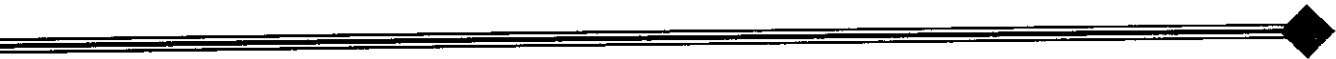
The architecture implemented is an excellent framework for the development of a tool based on generation of headers for the source code being developed. Though generic in nature the system addresses the issues of automatic generation of headers and written into the source file. The earlier versions of Document generation tools were lacking in few aspects specifically issues like generating headers into the source file, Generation of headers from the code, rather than from the comments of the developer. The prototype based on C/C++ API Specification Tool is not only free from the above mentioned problems but it also incorporates the additional features which makes this prototype an excellent tool for document and header generation. To avoid unformatted header generation, the C/C++ API Specification Tool uses the C & C++ language commenting conventions.

To incorporate customized header generation to the developer, parsing of the required category from the available tokens is performed. Over a period of time, different parsing techniques were proposed to parse from the token, but only LALR parsing technique has been used for parsing the category from the tokens generated by the scanner. Parsing the token is made easy by the implementation of tree structure to maintain the different token categories.

The tool developed is embedded into the GCC 3.2.1 compiler. So the header generation is done as and when the developer compiles the developed code. GCC 3.2.1 compiler is available for operating systems like LINUX, and SOLARIS. Which makes the C/C++ Api Specification Tool also available only for these operating systems.



Tools





9. TOOLS

9.1 C Overview

C is a programming language designed for a wide variety of programming tasks. It is used for system-level code, text processing, graphics, and in many other application areas.

The C language described here is consistent with the Systems Application Architecture Common Programming Interface (also known as the SAA C Level 2 interface), and with the International Standard C (ANSI/ISO-IEC 9899:1990[1992]). SAA Level 2 is an IBM definition of the C language that allows programmers to develop applications that can be easily transported across different SAA environments. It specifies several features of the C language that the ANSI C standard designates as implementation-defined.

On the Intel platform, the IBM C and C++ Compilers conforms to changes adopted into the International Standard C by ISO/IEC 9899:1990/Amendment 1:1994.

C supports several data types, including characters, integers, floating-point numbers, and pointers -- each in a variety of forms. In addition, C also supports arrays, structures (records), unions, and enumerations.

The C language contains a concise set of statements, with functionality added through its library. This division enables C to be both flexible and efficient. An additional benefit is that the language is highly consistent across different systems.

The C library contains functions for input and output, mathematics, exception handling, string and character manipulation, dynamic memory management, as well as date and time manipulation. Use of this library helps to maintain program portability, because the underlying implementation details for the various operations need not concern the programmer.



9.2 C++ Overview

C++ is a language derived from C, developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories.. C++ provides a number of features that "spruce up" the C language, but more importantly, it provides capabilities for object-oriented programming.

It was developed as a fast and efficient language to be used to produce any kind of software. C++ was developed significantly after its first release. In particular, "ARM C++" added exceptions and templates, and ISO C++ added RTTI, namespaces, and a standard library.

C++ was designed for the UNIX system environment. C++ is a general purpose programming language with a bias towards systems programming that

- Is a better C
- Supports data abstraction
- Supports object-oriented programming
- Supports generic programming.

Speed and Efficiency

When a compiler runs through your source files it changes all of the high-level C/C++ instructions and function calls into processor-dependent assembly language. This means that your programs run directly on top of the processor and can pull as much performance from the machine as need be, which speeds up the execution of programs significantly.

Dividing code into self-contained classes means that if a certain change needs to be made to the behavior of an object, that change only has to be made once in the object's code whereas, in procedural languages, such changes may have to be made in several places in the code.



Access to Operating System-Dependent Functions

Most operating systems are written in C or C++ (or a mixture of both) and therefore expose their function calls in C/C++. With C++, your program has the operating system at its fingertips, telling it exactly what to do with your program, whether it be manipulating windows, accessing a network, handling input, or reading and writing files.

Advantages

- New programs would be developed in less time because old code can be reused.
- Creating and using new data types would be easier than in C.
- The memory management under C++ would be easier and more transparent.
- Programs would be less bug-prone, as C++ uses a stricter syntax and type checking.
- 'Data hiding', the usage of data by one program part while other program parts cannot access the data, would be easier to implement with C++.

Disadvantages

C++ is portable to the extent that the language itself is the same across platforms (this too is disputable). If your program uses any platform dependent code, which it almost certainly will, you will have to change it considerably to work on a different platform.

C/C++ has no runtime error checking. If you try to access memory that doesn't exist (reading past the bounds of an array, overwriting parts of the operating system, etc...), you will probably end up in a crash. In many cases, such crashes are extremely difficult to debug.



9.3 LINUX

Linux was developed by Linux Torvalds at the University of Helsinki in Finland. He started his work in 1991. The effort expanded with volunteers contributing code and documentation over the internet. It is currently developed under the GNU public license and is freely available in source and binary form.

Some of its features include:

1. Virtual memory, allowing the system to use disk room the same as RAM memory.
2. Networking with TCP/IP and other protocols.
3. Multiple user capability.
4. Protected mode so programs or users can't access unauthorized areas.
5. Shared libraries
6. True multitasking
7. X - A graphical user interface similar to windows, but supports remote sessions over a network.
8. Advanced server functionality
 - o FTP server
 - o Telnet server
 - o BOOTP server
 - o DHCP server
 - o Samba server
 - o DNS server



- SNMP services
 - Mail services
 - Network file sharing
 - much, much more...
9. Support of filesystems that other operating systems use such as DOS (FAT), Windows95,98 (FAT32), Windows NT, 2000 (NTFS), Apple, minix, and others

Reasons to use:

1. Free
2. Runs on various machine architectures
3. Works well on machines that are not "modern". Recommended 8MB RAM, with 16MB swap drive space. It will run in hard drives as small as 500MB or less.
4. Linux is stable and even if a program crashes, it won't bring the OS down.
5. Source code is available.



9.4 GCC 3.2.1 Compiler

Several versions of the compiler (C, C++, Objective-C, Ada, Fortran, and Java) are integrated; this is why we use the name "GNU Compiler Collection". GCC can compile programs written in any of these languages. The Ada, Fortran, and Java compilers are described in separate manuals.

"GCC" is a common shorthand term for the GNU Compiler Collection. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs (as the abbreviation formerly stood for "GNU C Compiler").

When referring to C++ compilation, it is usual to call the compiler "G++". Since there is only one compiler, it is also accurate to call it "GCC" no matter what the language context; however, the term "G++" is more useful when the emphasis is on compiling C++ programs.

Similarly, when we talk about Ada compilation, we usually call the compiler "GNAT", for the same reasons.

We use the name "GCC" to refer to the compilation system as a whole, and more specifically to the language-independent part of the compiler. For example, we refer to the optimization options as affecting the behavior of "GCC" or sometimes just "the compiler".

Front ends for other languages, such as Mercury and Pascal exist but have not yet been integrated into GCC. These front ends, like that for C++, are built in subdirectories of GCC and link to it. The result is an integrated compiler that can compile programs written in C, C++, Objective-C, or any of the languages for which you have installed front ends.

In this manual, we only discuss the options for the C, Objective-C, and C++ compilers and those of the GCC core. Consult the documentation of the other front ends for the options to use when compiling programs written in other languages.



G++ is a *compiler*, not merely a preprocessor. G++ builds object code directly from your C++ program source. There is no intermediate C version of the program. (By contrast, for example, some other implementations use a program that generates a C program from your C++ source.) Avoiding an intermediate C representation of the program means that you get better object code, and better debugging information. The GNU debugger, GDB, works with this information in the object code to give you comprehensive C++ source-level editing capabilities.

GCC Command Options

When you invoke GCC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the `-c` option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Most of the command line options that you can use with GCC are useful for C programs; when an option is only useful with another language (usually C++), the explanation says so explicitly. If the description for a particular option does not mention a source language, you can use that option with all supported languages.

The `gcc` program accepts options and file names as operands. Many options have multi-letter names; therefore multiple single-letter options may *not* be grouped: `-dr` is very different from `-d -r`.

You can mix options and other arguments. For the most part, the order you use doesn't matter. Order does matter when you use several options of the same



kind; for example, if you specify `-L` more than once, the directories are searched in the order specified.

Many options have long names starting with `-f` or with `-W`--for example, `-fforce-mem`, `-fstrength-reduce`, `-Wformat` and so on. Most of these have both positive and negative forms; the negative form of `-ffoo` would be `-fno-foo`. This manual documents only one of these two forms, whichever one is not the default.



9.5 Windows NT Overview

The "NT" in Windows NT stands for "New Technology". Originally developed as a project called "OS/2 NT" by Microsoft and IBM, "OS/2 NT" was designed from the ground up to be the operating system of the future. The goal was to create an enterprise OS with the power of Unix, the ease of use found in the Windows, and networking functionality found in Novell NetWare... Then combine all that with new ideas such as "Zero Administration", and a wide array of other innovations that were secretly in development by Microsoft and IBM at the time.

The result is a faster, more reliable, and better overall OS than the previous DOS ancestors (DOS, Win1.0-3.11, and Win9x). Windows NT was built from the ground up. It does not run on top of DOS (like Win3.x), nor does it get a kick start from DOS (like Win9x). Windows NT does not have a COMMAND.COM, CONFIG.SYS, AUTOEXEC.BAT, IO.SYS, MSDOS.SYS, etc. Even the WIN.INI and SYSTEM.INI files are empty after you first install NT, only to be used if you install an older 16-bit Win3.x application.

Windows NT technology will be used in all future Windows operating systems. Windows 2000 will be built on Windows NT technology (Windows 2000 was previously known as Windows NT 5.0). Windows 98 was the last DOS/Windows OS.

Multi-processing

Windows NT supports more than one processor. This means you are not limited to only one processor like you are with Win95/Win98, or with a G3 power Macintosh (note that 604e based Mac's can use multiple processors, but your applications must be written to do so). Currently, Windows NT supports up to 32 processors, and is a true symmetric processing operating system. Without any additional tweaking, NT supports up to 4 Pentium Pro processors, 2 Pentium II processors, 8 Pentium Xeon processors, or 16 Katami processors.



Pre-emptive multitasking

Windows NT is a true pre-emptive multitasking operating system. Roughly speaking, this means that you can run more tasks simultaneously with less overhead, better overall performance, and more responsiveness. Windows 95 and Windows 98 only support pre-emptive multitasking with 32-bit applications, and is not supported "while" ANY 16-bit applications or device drivers are running. Mac OS does not support pre-emptive multitasking at all. NT supports pre-emptive multitasking with ALL applications and device drivers.

32-bit

Windows NT is a true 32-bit operating system, unlike Windows 95 and Windows 98 which are only semi 32-bit operating systems. The result of a true 32-bit operating system is better overall performance, as well as a better multitasking experience. The Win32 (Windows 32-bit) application structure was originally developed for NT.

Lack of conventional memory

Windows 3.1, 95, and 98 users are all too familiar with the memory resource problem. Windows 95/98 uses what's known as conventional memory (left over from the days of DOS) which is kind of like a 640k cache for icons, pointers, and other small tid-bits stored in memory whenever you open a window, application, etc. After some time these resources diminish rapidly. To see your system resources, open any system explorer window (try opening the Recycle Bin), and choose "About Windows" from the "Help" menu.

Windows NT does not use conventional memory, as everything allocated to memory goes in "Physical Memory". The result is consistent performance when running multiple applications, a better overall multitasking experience, as well as better system reliability.



Reliability

Protected Memory

Protected memory means that your system will be more reliable. By preventing applications to write over each other (or the OS) in memory, both you applications and OS are less likely to go down. One of the most common reasons for application and system crashes in Win9x or Mac OS are applications writing over each other, or the system, in memory. Neither Windows 95, Windows 98, Mac OS 8.0, or Mac OS 8.5 support protected memory.

Hardware Abstraction Layer

Windows NT prevents applications from accessing your hardware directly, greatly reducing the risk of a system crash. In the past it is was common for applications (especially DOS games) to access your hardware directly (most commonly: your memory, or video card), which would put your systems stability on the line. NT does prevents this type of activity, and will greatly reduce the risk of system crashes and serious virus problems as a result.

Lack of conventional memory

Just as mentioned in the performance section, NT does not use conventional memory. This will not only increase performance, but will also reduce the number of system crashes compared to Windows 95, and Windows 98. Conventional memory (left over from the days of DOS) is kind of like a 640k cache for icons, pointers, and other small tid-bits stored in memory whenever you open a window, application, etc. After time these resources diminish rapidly. To see your system resources, open any system explorer window (try opening the Recycle Bin), and choose "About Windows" from the "Help" menu. Windows NT does not use conventional memory, as everything allocated to memory goes in "Physical Memory". The result is consistent performance when running multiple applications, a better overall multitasking experience, as well as better system reliability.



Future Prospects



10 FUTURE PROSPECTS

- Supported platforms – Native
 - HP-UX 10.20 (PA 1.1), HP-UX 11.0 (PA 1.1)
 - Windows NT 4.0 , Windows 9x/2k.
 - AIX 4.3.3.

- Supported platforms – Embedded
 - PowerPC
 - Intel XScale
 - Pentium II & III

- Easily migrate from native development to embedded.
- Improved exception handling and extensive template support.
- Enhanced support for Windows developers.
- Open Source.



Future Enhancements



11. FUTURE ENHANCEMENTS AND RECOMMENDATIONS

This work can be extended in several directions in the future that addresses issues concerning operating system and languages. Following are the future enhancements, which can be, incorporated in the future extensions of the push architecture design.

A. Porting:

The tool developed is ported into GCC compiler, which makes the tool available for a few specific operating systems (LINUX, SOLARIS). Breaking this barrier of porting only into the GCC compiler can make the tool platform independent. Porting the tool into other compilers like TURBO, BORLAND etc can make the tool available for WINDOWS operating system.

B. Languages Support:

The tool is specific to C and C++ languages. This tool can be extended to languages like Visual C++, JAVA, PERL, COBOL, PASCAL, FORTRAN, and Visual Basic. Extending the tool to these languages the compiler should be able to parse more data for the code. For E.g. Visual C++ has a source browser information in a function. The tool should be able to parse this information and add it into the header. More over VC++ have MACROS that are not available in C and C++. The tools should be able to parse those MACROS.

C. Optimization:

Enhancements can be made on the performance of the tool. Enhancements can be made on memory management (Faster Memory Allocation and Garbage Collection). The tool is flexible; in the sense that changes in the target architecture can be quickly accommodated.



A major use of the tools is as follows:

- Propose an architectural change;
- Add an optimization to exploit/work around that feature;
- Measure the performance impact and thus accept/reject the change.

Architectural features, which have been considered, include:

- Address compare buffers.
- Combining dependent operations.

List of optimization that can be improved:

- Procedure Integration.
- Dead Code elimination.
- Loop unrolling.
- Live variable analysis.
- Leaf routine optimizations.
- Local register allocation.
- Global register allocation.
- Instruction scheduling.
- Unreachable code elimination.
- Code straightening.
- If expression simplifications.



References



12. REFERENCES

For Web sites:

- #R1 www.java.sun.com/j2se/javadoc/
- #R2 www.zib.de/Visual/software/doc++/
- #R3 www.doxygen.org/
- #R4 www.cs.bris.ac.uk/~ian/formal/index.html
- #R5 www.topaz.cs.byu.edu/text/html/Textbook/
- #R6 www.epaperpress.com
- #R7 www.gcc.gnu.org/onlinedocs/gcc-3.2.1
- #R8 www.interactivetools.com/products/docbuilder/
- #R9 www.freeware.sgi.com/Installable/gcc-3.2.1.html

For Books:

- #R10 Theory And Paractice Of Compiler Writing.
Jean. Paul Tremblay,
McGraw – Hill International Editions.
- #R11 Software Engineering
Rodger. S. Pressman.
Mc GRAW – HILL International Edition 1997