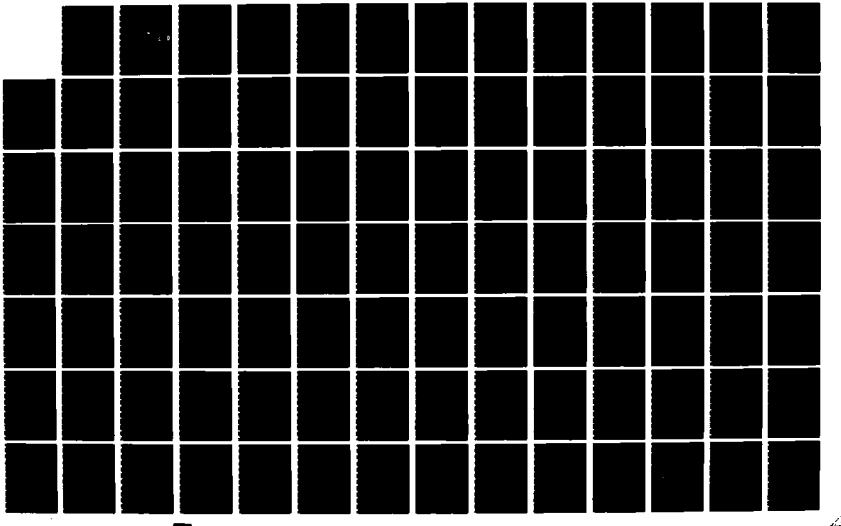
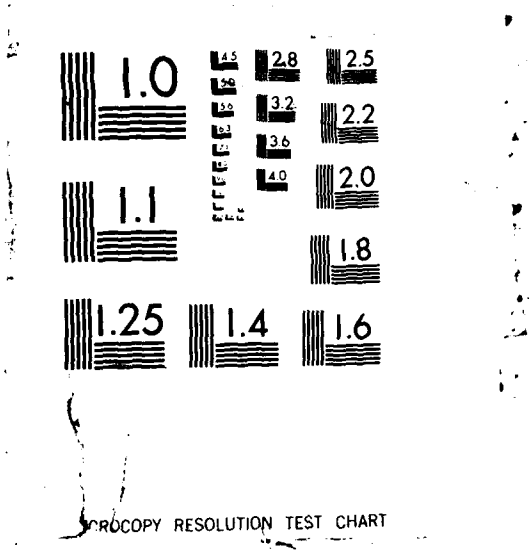
AD-A175 145 LEXICAL TRANSLATOR FROM ARABIC TO LATIN IN PASCAL ENVIRONMENT(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA S S ALJUHAIMAN SEP 86 1/2

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
ROCOPY RESOLUTION TEST CHART

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

LEXICAL TRANSLATOR FROM ARABIC TO
LATIN IN PASCAL ENVIRONMENT

by

Sadek Saleh Aljuhaiman

September 1986

Thesis Advisor: Daniel Davis

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
LEXICAL TRANSLATOR FROM ARABIC TO LATIN IN PASCAL ENVIRONMENT

12 PERSONAL AUTHOR(S)
Aljuhaiman, Sadek S.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1986, September	162

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Lexical Translator Bilingual Operating System

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The Lexical translator is a program written in Turbo PASCAL to generate a Latin PASCAL source code from an Arabic PASCAL source code. The Arabic code is written under a bilingual operating system transparent to the DOS on personal computers.

The bilingual operating system compatibility as well as the Arabic characters' code values is investigated. The Latin code is fed into a computer to be compiled and run with a Latin interpreter (i.e., Turbo PASCAL), in an Arabic environment.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. Daniel Davis	22b TELEPHONE (Include Area Code) (408) 646-3091 22c OFFICE SYMBOL Code 52Dv

Approved for public release; distribution is unlimited.

Lexical Translator from Arabic to Latin in Pascal Environment

by

Sadek Saleh Aljuhaiman
Captain, Royal Saudi Air Defense Forces
B.S., Arizona State University, 1979

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1986

Author: _____
Sadek Saleh Aljuhaiman

Approved by: _____
Daniel Davis, Thesis Advisor

_____
Ron Rautenberg, Second Reader

_____
Vincent Y. Lum, Chairman,
Department of Computer Science

_____
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

The Lexical translator is a program written in Turbo PASCAL to generate a Latin PASCAL source code from an Arabic PASCAL source code. The Arabic code is written under a bilingual operating system transparent to the DOS on personal computers.

The bilingual operating system compatibility as well as the Arabic characters' code values is investigated. The Latin code is fed into a computer to be compiled and run with a Latin interpreter (i.e., Turbo PASCAL), in an Arabic environment.

TABLE OF CONTENTS

I. INTRODUCTION

The English language is the most popular scientific language used today. The language descended from Latin and has had wide use in the scientific field. The English alphabet is familiar to people in Europe and all countries who use languages descended from Latin. There are slight changes between the various alphabets that have descended from Latin.

The wide use of Latin alphabets has made it easy to set standards for typewriters and console keyboards. The similarity in grammar common to most of them, their fonts and direction of flow (i.e., left to right) has made it easy to standardize.

Keep in mind that, many of the computer pioneers made an effort not to limit the implementation of their software to one spoken language. Software is the key to any limited use of computers in any language. Typically lack of knowledge of programmers in a foreign language limits their ability to write applications acceptable to the user. Not so many nations are blessed with the computer development technology. However all nations have people who, as users, are capable of contributing to humanity using this technology.

Given the technology existing today, if we can create an interface between a host foreign language and a target application language there will be fewer barriers to nations that do not use a standard English, French, or German-based computer operating systems and software. The interface will accept user commands from the host environment and translate it to the syntax of the target environment. It is assumed that the user is knowledgeable in the semantics of the target environment in his spoken language terms.

The question may be asked, "what good will this approach do such a nation?" There are several good points. Two of the most important reasons--One, there is a good library of software that exists; and two, the price of software (even with the addition of an interface communicator) is less than newly-written customized software. It is faster and easier to write an interface than to rewrite a large body of software.

Two user environments should not be confused. The customized foreign alphabets used in many countries on mainframes for specific applications are developed by contractors who are expert in that application but not necessarily the foreign language. The mainframes must use the software provided by the original contractors. It takes a lot of effort and capital to develop new software application for the special machine. This limits the use of the computer to operators and data entry personnel with

minimum creative programming from the user side. Users do not share the expertise of others and the continuously improving software. This is because there are limited users and minimum feedback to software developers.

The second user environment is the average user who has some scientific background but has no access nor the capital to invest in mainframe hardware. This user is often an educator, student, or a professional. This category of users has great potential. The use of software with a native language interface would be very helpful and affordable at the same time to this group. This group is very capable of contributing in their respective fields with the powerful processing features available with personnel computer technology today.

This thesis is concerned with the second user environment for several reasons. The second group of users are the creative ones. Their understanding of computers and its applications is a major step toward building the target machine with compatible native standards. This will eliminate the ad hoc design by the contractor who most of the time has to hire a non-technical translator and dictate to them the language specification, key words, and commands of the operating system, or query language. Usually a translator will translate the machine native language key words to the target language using its alphabet. The translator may have minimal programming or computer experience. This will most

likely lead to an ambiguous environment for users to work with.

The feasibility of such an approach is constrained by several factors. The language or the user environment is one factor. How is the language implemented or emulated on standard Latin language hardware? The target machine (i.e., micro to mini computers) compatibility with others in the same family is also a factor. These are factors that affect feasibility. Economical feasibility is based on demand and supply and a developer must evaluate the benefit vs. the development cost in order to develop such interface software.

The Arabic language is a very rich language in vocabulary and historical background. The Arabian alphabet is very old. The language was used for several centuries by leading ancient mathematicians, physicians, biologists, and chemists. They successfully contributed in their fields using the Arabic alphabet. Their numerals, symbols, and equations were all written in Arabic. However this does not make it simple to use the Arabic alphabet in the modern computer environment.

One reason is that the direction of flow in reading and writing is from right to left. Secondly, Arabic characters are not printed like Latin characters. Arabic words are printed like calligraphy. Arabic characters must be either written in stand alone or connected form. The character may

be located in one of three ways: at the beginning of a word, in the middle, or at the end of a word. With a set of complicated rules the shape of a character is determined by its location with respect to the word. This difficulty has complicated attempts to provide a software emulation to the Arabic environment in personal computers.

The goal of this thesis is to provide an approach to solving this problem. The steps that must be followed will be described in addition to special consideration. To show that translation is possible, we will develop an interface to communicate between an Arabic form of source code in the PASCAL language and an existing English PASCAL compiler. The interface will use sample source code written in Arabic and Lexically Translate it to English source code. The goal is, given correct Arabic source code, the interface will produce correct English source code. This should be done once. Once the program is compiled the interface step is no longer needed with the compilation.

II. BACKGROUND ON ARABIC CHARACTER

A. INTRODUCTION

There are 28 basic characters in the Arabic alphabet (Figure 1). However, these basic characters are not sufficient for use with computers or typewriters. Authorities agree [Ref. 1] that the optimum set should use a minimum of 31 characters (Figure 2), three more characters than the original set. The additional 3 characters are needed to constitute the optimum set for representing Arabic texts. One may check the Kufic script, which is over 1500 years old, to realize that engravings by ancient Arabs were done with close to 31 characters. Each character has one shape. Over the years, variations of the characters have developed for ease of writing and reading. Each character may have from two to five shapes depending on its location within a word. All applications must use these variations as standards to represent Arabic texts. Implementing the variation is critical for compatibility issues. Code representation of any variation must follow a strict standard to insure survival among other implementations.

The Arabic alphabet has only three vowels in the 28 characters (see Figure 3 for the alphabet names). Vowelization is also performed through the use of diacritics (see Figure 4). Most Arabic texts do not show diacritics.

Readers have learned to read and understand the word based on the context of its use. If misinterpretation is critical, verifications are provided in parentheses. Most applications today do not require diacritic symbols.

The Arabic numerals and Hindu are used in the Arabic world. North African countries use the Arabic numerals (as used in Latin). The Arabic name is given to the numerals used in Latin, and Hindu numerals are used by most of the Arabic world (Figure 5). However, history books show that both systems originated in India. The Arabic language uses the Latin comma for a decimal digit to be distinguished from the Arabic number zero which is the Latin decimal digit ".".

B. ARABIC LANGUAGE

The Arabic language differs from languages descended from Latin in several ways. The primary differences are:

* Arabic is written right to left instead of left to right.
* The representation of vowels by using diacritics in the form of over or under scores with most letters within the words.

Secondary differences are:

* Letters in Arabic may be joined or not according to location within the word. A particular letter may be joined to the preceding letter, and/or following letter.
* Each letter has between two and five different forms dependent on its contextual position.

Lexically the Arabic language can be defined in BNF notation as follows [Ref. 1:p. 28]:

<language> ::= {<sentence>}$_1^*$

<sentence> ::= {<word>}$_1^*$

<word> ::= {<characters><voc.sym><character>}$_1^*$

<character>::= { see Figure 1. }$_1^*$

<voc.sym> ::= { see Figure 4 . }$_1^*$

C. WRITING ARABIC

Writing in Arabic flows from right to left, additional lines start from right to left beginning below the previous line. A word is entered by typing the first character at the cursor position followed (to the left) by the next character. An example of this is the word "hello." If the same word is entered in Arabic it will be entered as follows:

cursor position ---------- _<

step 1. enter character "h" ----------_h<

step 2. enter character "e" ----------_eh<

step 3. enter character "l" ----------_leh<

step 4. enter character "l" ----------_lleh<

step 5. enter character "o" ----------_olleh<

This demonstrates the direction of flow, however if one should worry about each character shape, it may seem tedious for long text. In some applications one must provide diacritics also. In short, typing one vocalized word seems like a puzzle.

There are rules governing the shape (form) of the letter based on its contextual position.

Dewachi, Abdulilah [Ref. 1:p. 27] has the following opinion on the rules:

> These rules have, in my opinion, been developed for ease of handwriting and have no bearing on the semantic and/or syntactic requirement of the language.

In spite of the cause or the reason for the development of the rules, all books, newspapers, and magazines in the Arab countries today are written using those rules. They will also stay that way for years to come.

Arabic letters are cursive in shape. The implementation of the alphabets is highly dependent on how legitimate the characters look. The cursive nature of characters requires that both monitor and graphic adapter provide good resolution. High resolution is also required for supporting correct vocalization, as previously discussed.

D. ARABIC NUMERALS

Both the eastern Arabic numerals and the western Arabic numerals (Figure 5) are used. Countries like Algeria, Morocco and Tunisia use the western Arabic numerals. The numeral system is not a critical issue since in both representations they have the same value.

Many people believe that the Arabs write the numbers from left to right. This is a misconception. The language books and schools teach the classical way of writing and reading the numerals. The classical way is to either use the words ("one","two",...) or the numbers ("1","2",...) in writing starting from right to left. For example the number

523 will be written in Arabic as "three and twenty and five hundred." It may sound wrong in English composition but this is the syntax that classical books use. This method should be encouraged. This is also followed in reading the numbers.

The most common method in handwriting numbers is to write in the order they are said. An example of how numbers are read and written today is the year 1986--pronounced as "One thousand nine hundred six and eighty. Notice the six comes before the eighty. Writing the number "1986) using numbers is done as follows:

first digit	1___
second digit	19__
third digit	19_6
fourth digit	1986

This method is far too complicated to be adopted by mechanical machines. The classical method should be encouraged for another obvious reason. The numbers are entered least significant bits first in low memory. From the computer hardware point of view the adders/subtractors may work on the number before the complete number is loaded [Ref. 1]. This is the more efficient way. Also both numbers and strings will be right justified.

This chapter has outlined the major concerns and differences between the Arabic and Latin alphabet. There are a few more things worth noticing. The opening brackets "[",

"{", and "(" are the closing brackets in Arabic and vice versa. The Arabic question mark has the same look as "?" but rotated 180 degrees around its vertical center. A list of a complete code set including special characters is included in the ARCII code set (Appendix D). ARCII will be discussed in detail in later chapters.

III. CONTEXTUAL PROBLEMS IN ARABIC WORDING

For any computer to work in Arabic it must also be able to handle English alphabets. Arabic users will pay a few extra dollars to add the bilingual features in purchasing a computer. The form of the bilingual feature is a controversial issue. This chapter will show why one should be concerned in using mixed mode or even alternative between the two alphabets--Latin and Arabic.

There are three major differences between alphabets descended from Arabic and Latin. The differences are direction of flow, diacritics, and variant location shape of characters. These issues are specific to the language. This chapter will discuss these issues with respect to the computer environment.

Each difference requires special attention in an Arabic alphabet implementation in hardware. The direction of flow in reading and writing is very complicated for users and developers alike. This is especially true where the keyboard, the display, and the printer are to operate in bilingual mode. Arabic is read and written in the opposite direction to Latin. The difficulty is when the user wants to flip to the other mode for another application, or within the same applications the user wishes to mix both character sets.

A boom in the introduction of electronic computing to the Arabic world lead manufacturers to make short cuts to meet the complicated needs of the Arabic alphabet. Also the Arabic alphabet is used in several countries with non-Arabic languages. This wide use invited companies to quickly develop a character set for Arabic, based on limited research. As a result important language needs such as diacritics were avoided. This also has lead to a delay in the realization of an effective solution.

The contextual problems, that is, the variant shape of characters, is the most difficult. To establish a solution is to decide the style or the method that developers should follow in implementing Arabic character sets. The problem is the complexity of providing to the user all shapes possible for the 28 character set on the keyboard. Each character has between two to four shapes, making for a total requirement of 84 codes to represent the minimum set of the Arabic alphabet. This number is higher by 50 percent than what the English alphabet (upper and lower case) requires. The rest of the special characters and diacritics require more codes. In some cases the applications of diacritics to some charac-ters requires a unique shape to represent it. This requires a unique code for the combination of characters and diacritics. The use of "Hamzah"[1] also

[1]The "hamzah" is one of the three characters that were added to the alphabet in addition to the original character set.

requires special attention when used with any of the three vowels in the alphabet. The limited number of codes the keyboard has is the limiting factor for planning the code assignments. A look at some efforts and proposals will be discussed in the following chapter.

A. DIRECTION OF FLOW

Working in mixed mode is considered a must in the Arabic environment. There are two approaches to handle the mixed modes data entry and storage problem. One approach calls for the data to be stored in aural order (i.e., logical order). The second approach is to store the data in the same order as it looks (i.e., visual order). Keep in mind that if an Arabic word is inserted in English text the last character of the word will be encountered first, scanning from left to right.

One approach places the burden on the display to translate the incoming data to the correct direction to be displayed. The display must translate an escape code or a mode bit sent with the data. The easiest method is to set a high bit (if it is not used) as to whether the character is Arabic or Latin. This option calls for smart display devices.

The second approach is to store the data in aural order. This approach places the burden on the computer to determine how to store data to cause no shifting of display direction. This means the display program will keep track of the

language mode and do order reversing to store the data in an appropriate order. In handwriting, handling mixed modes is done in the following fashion:

- continue typing until reaching a foreign character.
- count the number of spaces occupied by foreign characters up to the first native character.
- skip that number of spaces and write back to where you stopped before skipping. When done the writer should end where he/she jumped from.
- skip the same number of spaces you counted. This is where the next native character belongs.

It seems that humans can do this routine more easily than computers. The computer can only deal with incoming data as it arrives, one character at a time. This means the computer does not know in advance how many foreign characters are coming. The computer can use a logical device called a stack. Characters of different mode are stored (pushed on the stack) up to the next native character. At this point the computer has the foreign string in reverse order on the stack. In the next step the computer starts to write from the top of the stack until no more characters are in the stack. Then the program continues with the last encountered native character. In this approach the direction of flow for the display is maintained. Obviously this method has several disadvantages. One, it slows the storing of data in mixed mode. Two, it slows the computer from doing other functions, where a smart display could

handle the display of mixed mode data as they are stored logically.

The approach that should be taken is connected with resolving the contextual issue, the variant character shape problem.

B. ARE DIACRITICS REQUIRED?

By linguistic standards the omission of diacritics by computers murders the Arabic language. Linguists have always officially criticized the mispronunciation of statements by television and radio people. The use of diacritics is a must in the language even by recommendation of westerners involved with the Arabic alphabet [Ref. 1: pp. 39-46].

In a previous chapter diacritics were discussed. There are five basic diacritics. The five are (Figure 3) from right to left: "Fat_ha", "Dammah", "Kassrah", "Sukoon", and "Shadah". The first three can be doubled, in the same manner as double quotes in Latin. When any diacritic is doubled it is known as "Tanween" and adds an N sound to the character. The Shaddah has the same effect as doubling the consonant in English. It can be used inconjunction with any of the first three or their "Tanween." The Sukoon, when used, means that the character must be read in primitive form, versus using previous diacritics.

An example of one word using different diacritics will show how the sound and subsequently the meaning changes.

The word pronounced "tilmeeth" in Arabic means a student. The "th" at the end is the character "Thal" in Arabic. The example will show the different sounds per word when only the last character has different diacritics.

WORD	VOWELIZATION	PRONOUNCED
TILMEETH	"FAT_HA"	TILMEETHA
TILMEETH	"KASRAH"	TILMEETHI
TILMEETH	"DAMMAH"	TILMEETHO
TILMEETH	"SUKOON"	TILMEETH

Using the "Tanween" effect with the first three diacritics, the same word is pronounced as follows:

-	with "Fat_ha tanween"	TELMEETHAN
-	with "Kasrah tanween"	TELMEETHIN
-	with "Dammah tanween"	TELMEETHON

Shaddah has the ability to be used with all the above except the Sukoon.

The use of diacritics removes the ambiguity in the reading of text. It is powerful enough to change the meaning of the sentence completely. The vowelization of verbs by diacritics will change the sentence to passive tense. In Arabic the verb comes before the noun. So in Arabic the two statements, 'was stolen Ali a book,' and 'stole Ali a book' without the use of diacritics, especially on the verb, could not be distinguished. The effect of the "er" and "ee" in English as in "employer/employee" is also achieved by the use of diacritics in Arabic on the noun. In

combination, the failure to use diacritics can completely obscure the meaning of a sentence. For example, it would be as if in the sequenced fired/was_fired employee/er we did not know which of each alternative is meant. The employee either was fired, or fired someone. On the other hand, the employer either was fired, or fired someone. See Figure 6 for some examples using vowels and without vowels.

Clearly one can see the need of diacritics. In religious and history texts, they are used extensively. In an international symposium for standardization of character code sets and keyboards for Arabic language in computers held on 1-4 June 1980, several proposals were presented by researchers and companies that already have developed their own character sets [Refs. 1,2]. All the proposals and recommendations agreed on including the diacritics. This use of diacritics will be beneficial in the use of data bases, artificial intelligence and educational textbooks.

C. THE CONTEXTUAL ISSUES

The mere presence of a character in different locations within a word determines the shape to be written or read. Should the computer do the analysis and free the user from worrying about a large complex character set, or should the keyboard contain all possible variations of each character and have the user learn to master more than one hundred strokes for the alphabet in addition to numerals, special characters, and punctuation?

One popular approach is to provide only a minimum set of required characters, usually between 31 and 60 not including diacritics, numerals, and special characters. This approach is known as the single character single shape keyboard. The data is stored in memory or storage devices using this reduced code. The reduced code is analyzed by an interface to give the right form or shape. The interface is part of the display, when smart displays are used, or a shell on top of the "O.S." to contextually analyze the character form.

The issue is not quite settled and standardized among all Arabic alphabet users, nor Arabic countries. A successful meeting of authorized people from all concerned countries have not yet, to my knowledge, agreed on a standard. A few companies who stepped into the market early have generated their own version of character code sets. Some companies have realized the gap between their early implementation and actual language needs. The gap was realized more when the use of the produce was not utilized in all the areas and aspects for which it was designed. Some companies have realized that the survival and popularity of their product depends on compatibility with at least the codes of a character's internal representation. Some companies went further by investing in research for an optimum solution. Language experts were hired and/or consulted by companies like IBM, TI, and WANG. The companies

are following efforts for solutions and continuing further the research to achieve an effective solution.

In resolving the multiple character shapes, most companies have tried some reduction of all possible codes to a single code using several philosophies. Texas Instrument has presented [Ref. 1] three approaches to reduce the Arabic code.

The first approach was called "CORRESPONDENCE & DIFFERENCES." This approach divided the alphabet into groups. The first type A have characters with one, two, or three points (Appendix B). The second type B are without points. The last type C contains characters having at least one form of each, for example character "RA" and "ZA." The two characters have the same form with a point on the "RA" and no point on the "ZA." The idea is if the basic form has one key (code), two or more characters will have the same basic form, the points can be added later.

The second approach was called "ROOTS & APPENDICES" (Appendix B). The approach divided the alphabet into groups. Two groups have six characters in each. Another group has four characters. Each of the above groups have the same cursive and "APPENDICES." The "ROOT" of the character can be used at the beginning or in the middle of a word. One appendix will complement each root of a group. This will require a total of seven codes for a group of six roots. The group would require (for six characters, each

with three contextual forms) a total of 18 separate keys and/or codes. This approach implicitly asks for more software to analyze the appendices. A character may be represented by two codes internally. This will make text storage inefficient.

The last approach was "CONTEXTUAL ANALYSIS" (Appendix B). Texas Instruments has developed a product using this approach. The DS990 Bilingual System can handle Arabic/Latin modes and display them on a screen or line printer. The contextual analysis approach, in all the developments seen by the author, uses a reduced code set. The reduced code set is used for the internal representation of data. Keyboard keys of the Arabic set are kept to a minimum, usually the basic form. A software interface analyzes the character contextually and displays the characters in the right form. This interface software in some application is pushed further away from the responsibility of the CPU to the display terminals. Such terminals are called 'SMART' terminals. TI's DS990 system diagram (Appendix C) shows the configuration of a typical system.

TI realized the need for diacritics in the Arabic language after it introduced the system to the marketplace. TI, at an international symposium held in Riyadh, Saudi Arabia between 1-4 June, 1980 [Ref. 1:p. 68], in an effort at standardization of code, character sets, and keyboards, recommended that the Arabic computer systems standards

requirement include the use of diacritics. This is an example of the approach of the pioneer companies who had to define and develop the alphabet codes set. Premature standards will automatically be overriden by the authorized agency. The DS990 did not handle the use of diacritics. Since the use of diacritics was adopted by all standards committees, this lead a few companies to follow a new standard that supports diacritics.

ALIS, Inc., introduced BCON ™ as a bilingual operating system. BCON was geared toward MS-DOS based microcomputers. The bilingual operating system is an interface between the operating system (O.S.) and different applications [Ref. 2]. This bilingual operating system adopted the single key or single code approach. Each character is represented internally in memory by a unique code. BCON also fully supports the use of diacritics in text. The single code approach, as mentioned before, requires that a device or an interface (hardware or software) properly analyze the character and display the correct form. BCON uses Application Screen Image Compensations (ASIC) to perform the contextual analysis. BCON uses separate codes and fonts for each character. The internal character code gets translated (mapped) to its output code. The internal code has 4 to 5 output codes. The code to be displayed is based on the location of the character within the word (TI's and BCON's system will be covered in more detail in the next chapter).

IV. EFFORTS TO STANDARDIZE CODES

Several nations use the Arabic alphabet today, both Arabic speaking nations and non-Arabic speaking. It is a political challenge to gather concerned nations and succeed in establishing a standardized code set acceptable to all of them. It is difficult for any one country to take the initiative and responsibility to follow such a program until it comes to life. It is hard for a single country to conduct research and share knowledge with another country that is thousands of miles away. In recent years as cooperation between Arab nations has increased, and as methods of communication have improved, as well as travel, there have been more productive meetings and symposiums. Several countries have mutually cooperated to work and develop a possible solution to the standard codes set for Arabic in data processing.

Many countries like Kuwait, Iraq, Morocco, and Saudi Arabia have hosted meetings and symposiums, listening to experts on the language, and in the data processing field. Researchers, as well as company representatives, have brought up points to consider, shared their experiences, and given recommendations. Several existing systems have been developed or proposed by companies or individuals in the field. The countries that have been exposed to technology

and are more developed than other Arabic nations, have an urgent need to set standards in general. Countries like Morocco started as early as the 1950's to set standards for printing devices.

The north African countries have progressed further in this research. Morocco shared willingly with the Arab nations their latest research and developments in the area. The problem of choosing an existing system, with some or no modification, or to redefine once again a new standard, is also a political issue.

A. SOLUTION EFFORTS

Several companies have provided results of their research and in some cases have implemented systems, giving recommendations and results of conducted tests, in the case of keyboard layout proposals. Companies that have an interest in the market and have worked in the Arabic data processing field, have no authority to develop a code set standard. Government representatives are the authorized agency to do such a task. Several companies have proceeded, given a lack of standards, to develop Arabic code sets and implement them on hardware. This has resulted in several incompatible systems of code sets. Data in one system means different things in another code set system. This approach to the development of code sets has both disadvantages and some advantages to the companies involved.

Early development made companies as well as users understand the weaknesses of the developed system. For example, TI's DS990 system's omission of diacritics failed to fulfill the needs of the language. On the other hand, by just introducing a product early, companies make their name familiar to customers. The customer cannot complain about a reasonable attempt. This did establish a good reputation for such companies, especially when they adopt the approved standard and reintroduce their products. In addition to developing a good name, they gain experience in the process. This will help in introducing an earlier product complying with the standards. So a company's early efforts are not a total waste.

Since early implementation ignored including diacritics use with text, newer designs have to pay special attention to their use. Data base machines must pay attention when sorting and searching. The representation of diacritics will require special care from data processing machines. The priority of characters with or without diacritics must be known to the machine. A process of stripping diacritics from a given string to be located to match with a query, will facilitate the search. However, the target of the search, when found, must be displayed, and stored if updated, in the vocalized form. Unlike Texas Instruments, IBM chose to maintain domination in the market for type-writers and Arabic only EDP machines. IBM did conduct

studies on their own in an effort to develop a code set and keyboard layout. IBM, represented by Mr. R.P. Hajjar and Dr. A.M. Ismail, presented their attitude toward a bilingual code set standard at the symposium held in Riyadh, Saudi Arabia, in June 1980 [Ref. 1:p. 72]:

> Meanwhile, competent people from the Arab world and from elsewhere, have addressed the same subject and came up with a variety of solutions that are not compatible with each other, due to the fact that they reflect the requirements of a particular Arab country, but may not be totally acceptable by the neighboring Arab country. This is the main reason why IBM has not implemented such solutions, but will look forward to investigate the possibilities of their implementation, in case these solutions are adopted as part of an inter-Arab standard.

IBM, TI, and Wang have shared their research and willingness to achieve a solution and adopt it in their products.

This chapter will briefly cover three systems:

- TI DS990 System
- ALIS Inc., BCON System
- ASV-CODAR Proposed System.

B. TI DS990 BILINGUAL SYSTEM

DS990 is a bilingual system that generates seven bits for ASCII codes and generates an 8 bit code for Arabic codes. The system represents the Arabic alphabet with 32 unique codes in addition to 13 special characters. The thirty-two codes are the internal representations of the alphabet. TI's system uses the one key many shapes philosophy. The 32 codes are the basic character set of the system (Appendix C). The one key many shapes approach

requires the use of an interface with a smart display to display the correct form and shape. The DS990 block diagram (Appendix C), shows how the system is arranged. The 32 codes are mapped to 128 less 13 giving a total of 115 shapes that can be displayed. The display ROM interface contains all 128 shapes (Appendix C). The display service routine (DSR) and the display ROM interface contextually analyze the basic code set and display the data correctly by mapping one code to one or two display code(s).

DS990 does not handle diacritics. It also increased the optimum set from 31 to 32 unique characters. The system considers LAM ALEF as a single character. Two clear violations. The use of diacritics is a must in data processing. The LAMALEF (DC hex value in the basic character set) (Appendix C) is composed of the character LAM (D6 hex) followed by the character ALEF (C0 hex) which are two separate characters and should not have a unique code. The fact that the table shows no special code for eastern Hindu numerals indicates that the same code for Arabic numerals, known as western Hindu, is used for both representations (Figure 5). Depending on the display mode, the eastern (Hindu) and the Arabic (western Hindu) are displayed differently. So a user of a north African country cannot use the western Hindus (known as Arabic numerals) in Arabic mode. This is not desirable.

DS990 stores information in memory in logical order in Latin mode and Arabic mode. The display ROM interface and the control program map the internal representation of one code to one or two display codes. For example, to display the character 'SEEN' as in the basic character set (CB hex value) (Appendix H), the character is represented by two display codes. The first code is the value BC hex followed by the code 8B hex in the display ROM interface table.

The approach followed by TI is the typical way most companies are implementing their display techniques. However, the disadvantage is the omission of diacritics and considering "LAMALEF" as one character. TI has indicated they now believe the implementation must have diacritics. [Ref. 1]

C. ALIS INC., BCON SYSTEM

ALIS Inc., introduced BCON TM as a bilingual operating system that could be a standard to follow, or at least close to a standard. The bilingual operating system adopted the single key single code approach. Each character is represented by a unique code internally in memory. BCON also fully supports the diacritics use in text. BCON was geared toward MS-DOS based microcomputers. The bilingual operating system is an interface between the MSDOS operating system and applications. BCON is designed to facilitate the adaptation of the large number of existing MS-DOS applications to Arabic [Ref. 2]. The single code approach

as mentioned before requires that some device or interface (hardware or software) properly analyze the character and display the correct form. BCON uses Application Screen Image Compensations (ASIC) to per-form the contextual analysis, and then selects the correct display code (Appendix D).

1. Hardware and Software of BCON

BCON hardware is another board on top of the Latin character generator board. The new board has the Arabic character generator with the required wiring to allow concurrent operation of both character generators. The two boards are back to back and use one slot on the mother board--a microcomputer. Keyboard caps (or stickers) are provided for use on the keyboard. The stickers have both alphabets printed side by side.

The software is a program which when activated, resides in low memory and uses 19k bytes. Once BCON is activated, it can be set in Latin "native" mode or Arabic mode. The only way to free memory is to reset the system. Both modes of the operating system will allow bilingual insertion in the appropriate direction. In their early version (up to early 1985), ALIS introduced a reduced code called Arabic Reduced Code Information Interchange (ARCII). ARCII is the internal representation of the characters in memory and what is seen by the operating system.

2. ARCII Code Set

Arabic Reduced Code for Information Interchange (ARCII) is ALIS's early attempt to define a code set. The reduced code (ARCII) (Appendix D) is the internal representation codes of data in memory. The ALIS reduced code is completely different from early proposals for a target standard set proposed by ASMO (further details will be covered in the next section).

The code uses the graphic characters for the Arabic set. By assigning one to the 8th bit, 128 additional codes are available for Arabic codes. This allows the BCON bilingual system to mix codes and use both ASCII and ARCII. There are 46 different codes assigned for the alphabet, starting with code DO hex and ending with FD hex. ARCII places the diacritics early in the table to give them priority in sorting algorithms. This early positioning in the table was not favorable, however. The reasoning will be discussed when the standard code and the format justification are discussed. The escape codes and special characters should not be redefined for ARCII if similar ones in Latin exist. This minimizes the code set for ARCII, freeing more code for future expansion. Codes for functional codes could be minimized by using the international one.

ALIS reduced code is completely different from early proposals for a target standard set. The Arab Organization for Standardization and Metrology (ASMO), after several

years of research and after meeting with Arab representatives, recommended the use of CODAR U-F.D. as a standard for Arabic codes (further details will be covered in the next section). Subsequently, ALIS and other companies adopted the new code set in order to assure compatibility with other applications and implementations. BCON's original version of reduced code (ARCII) (Appendix D) is the internal representation of information in memory.

The form or appearance of characters is not a major issue as in how it should be displayed. This is dependent on the machine resolution and capabilities. The fonts and style of displayed texts vary from one machine to another. ASMO has recommended that the style of displayed text be left to developers. This has left a lot of room for manufacturers to be creative and compete for quality work for the benefit of the user.

3. Operating Principles of BCON

BCON, once loaded, resides in memory using 19k of low memory. BCON has three code sets. The three code sets are: reduced code (ARCII), key code and display code. Figure 7 shows how the three codes are integrated with each other. A list of the three code sets is provided in Appendix D. ARCII includes the diacritics as a part of the code set. This was set as a requirement of the CODAR U-F.D. standards. BCON receives the key code and stores it in memory in reduced code form. The reduced code form is

analyzed by BCON and contextually analyzed and displayed in the correct form. In the display process, BCON appends if necessary what is called "TAIL GENERATION" to some characters if they fall at the end of a word [Ref. 2].

The early work on BCON, as well as the work of other companies, must be modified to correspond to the new standards. ALIS in early 1986 introduced a new mode in addition to ARCII. The new mode uses the ASMO approved code set. No documents are available at this time. However, as mentioned before, previous effort was not totally lost. The company still utilizes the contextual analysis developed earlier, with minor modifications. The same is true for their printer driver software. This is a good example of how early development enables a company to react quickly to new demands.

D. ASV CODAR-U SYSTEM

In researching the early efforts initiated by official organizations or government agencies for inter-Arab unification of the codes set, two names were always associated: CODAR and Dr. Lakhdar. A few acronyms are important here:

CODAR : Code Arabs (French)

ASV : Arabe Standard Voyelle (French)

IERA : Institute d'Etudes et de Recherchers I'Arabisation

IBI : Intergovernmental Bureau for Informatics

COARIN: IBI Committee on the use of Arabic in Informatics

ALESCO: Arab League Education Cultural and Science Organization

SASO : Saudi Arabian Standards Organization

ASMO : Arab Organization for Standards and Metrology

Dr. Ahmed Lakhdar Gazal, Director of IERA (Institute for Research and Studies for Arabization in Rabat, Morocco) has been associated with the CODAR project for several years. Dr. Lakhdar proposed that the Arab nations adopt the CODAR system as a standard for telecommunications. IERA was working as far back as 1955. The standardized Arabic Code was a dream many people were expecting and needed for many years. However they have no power over defining it or making it official, assuming it is acceptable.

The CODAR system is a long-going project that is geared for setting standards for several fields of interest. The project covers:

PRINTING

- TYPEFACES
- TRANSFER LETTERS, SELF-ADHESIVE TYPES
- SLUG-CASTING MACHINES
- MOVABLE TYPE COMPOSITIONS-CASTER
- PHOTOCOMPOSITION

TYPEWRITERS

INFORMATICS AND DATA TRANSMISSION

TELECOMMUNICATIONS

This chapter is concerned with Informatics and Data Transmission. However, a lot of credit must be given to

personnel behind CODAR. It took CODAR a lot of effort and dedication by IERA's staff to accomplish a unification. A long list of acknowledgments, appreciation, and financial support letters were coordinated by CODAR from several countries and organizations. A list of participants include:

Moroccan Ministry of Education (1956)

First Conference of the Arab National Commissions for UNESCO (1958)

First Conference on Arabization (Rabat, 1961)

UNESCO (Arab book-keeping experts meeting) (Cairo, 1972)

A long list of occasions and dates are listed [Ref. 1:pp. 207-210].

Under Informatics and Data Transmission there were three versions of the 7-bit code system. They are:

Seven bit CODAR I :	first coding scheme of the ABV characters
Seven bit CODAR II:	a proposition for a unified Arabic coding scheme, discussed at regional (IBI) meeting at Bizzert, Tunisia, June 1976
Seven bit CODAR U :	unified coding scheme for the Arab countries proposed by COARIN (IBI committee on the use of Arabic in informatics) at a meeting in Rome, June 1977.

The seven bit CODAR I, CODAR II, and CODAR U (Appendix E) are code set proposals. CODAR I was produced by EURAB and the printers were manufactured by the Italian firm SELI. CODAR II is a subsystem of CODAR I. The subsystem can be obtained by removing all possible combinations of "Harakat" (i.e., Fat'ha, Kassrah, and Dammah) with the "Shaddah." The

subsystem also leaves out three Persian characters, opening and closing square brackets, backslash and a few character variant shapes.

CODAR U fully supports vocalization with all possible "Shaddah" combinations with the "Harakat." This system is the closest to being acceptable by ASMO and approved as a standard. ASMO's approval will give the system official status.

E. THE STANDARDIZED SET

In 1980 CODAR U was accepted as a working basis for a basic code set. Recommendations and modifications were to be presented to ASMO in order to formalize the code set. The next step was to distribute it to ASMO's members. Member countries insure that it is implemented accurately.

During a meeting held between 22-24 April in Rabat (Morocco), the final code for the proposed standard, called CODAR U-F.D. was finalized and submitted to ASMO along with six recommendations (Appendix F). The conference recommended ASMO to distribute and test the code by IERA, SASO, and the National Center for Information in Tunisia before enforcing the code. ALESCO and ASMO were also recommended to make every effort for the adoption of the code by all Arab countries.

Finally, on October 21, 1982 ASMO adopted the code prepared by IREA, and ALESCO. This code was the result of the CODAR U-F.D. proposed in April, 1982 at Rabat. The

modifications and changes are included (Appendix G). There are a few points to consider. There are 31 codes for the alphabets, 3 codes for "Harakat," 2 codes for "Shaddah" and "sukoon," 5 codes for "Hammzah," 3 codes for "Tanween," totalling 44 codes. Their location must not be changed in the table under any circumstances. The "Hamzah" in all variations, on top or under characters, are considered forms of "Hammzah." The "Hamzah" is placed in the beginning of the code table, which in searching means any character with "Hamzah" associated with it should be expected higher in order (equivalent to "A" in Latin). This concept will confuse users when searching or sorting. The results may be surprising for sorting algorithms. In sorting, the table allows a simple sort. Errors will result from the occurrence of diacritics and the code 60 hex in the table (6/0). The code 60 hex is used for connection or extending a word for formatting purposes. So a sorting algorithm should strip text of the diacritics and the connection dash (similar to Latin underscore) first, then sort the text according to the basic 31 character code. The user must be educated about all the remarks mentioned in the reasoning in ASMO's final form of the code set. Another convention was that the character comes first in words that are vocalized. The form to follow is:

$$\text{WORD} ::= \{ \langle\text{CHARACTER}\rangle \langle\text{SHADDAH}\rangle \langle\text{DIACRITICS}\rangle \}_{1}^{*}$$

So the "Shaddah" comes before the diacritics if used for a character. The second convention is if the pure word matches in sorting, the diacritics then should be used by the sorting algorithm as qualifiers. In my opinion, this violates the Regularity Principle in programming, where the user must be concerned and remember all the exceptions. This does not in any way mean there is an easier way.

F. CONCLUSION

The ASMO code set is the standard Arabic code set the Arab countries must enforce in their countries. Subsequently all companies in the area must adopt and use a standard code set. The competition is now directed toward improving the display application with high resolution and graphic capabilities. Printing devices also are an area for manufacturers to compete in printing different Arabic styles and fonts. The contextual issue is left as a flexible issue to the implementors to research and develop for their individual products. The display form of text on monitors and printing devices will not affect the internal representation of the data, which must be compatible with the standard code set. This may result in several display sets developed by the companies as their view and intention of displaying a good Arabic text. Hopefully this should create a stable base to work with and encourage development of products based on the ASMO standards and conventions listed in Appendix G.

V. INTERFACE DESIGN GENERAL APPROACH

The lexical translator will generate Latin code from an Arabic source code in Pascal syntax. The Pascal compiler can compile/run the Latin code to generate an output. The interface will generate a correct Latin code given that the Arabic source code is in correct syntax. The translator will give minimum help to correct the Arabic code. The user must understand the syntax and the semantics of the language to write correct source code. The interface is not an interactive type of translator. The design is generally the same for all Pascal compilers. The interface must always consider the environment it will work in. The interface has two environments to consider: the source code bilingual system, and the compiler environment. From the portability and compatibility point of view, the translator will be limited to a particular Arabic standard, and a particular PASCAL implementation.

The bilingual implementation has its own function codes. Those codes are embedded within the Arabic source code, if generated under the bilingual operating system. The bilingual operating system used here is BCON from ALIS, Inc. There is a list of function codes in Appendix D. The PASCAL compiler used here is TURBO PASCAL from Borland, Inc.

The Arabic implementation utilizes the upper half of the 255 character set used by graphics to display Arabic fonts. Some Pascal compilers will accept any of the 255 characters as legal characters for use in string data. Turbo Pascal, for example, allows the entire set of 255 characters. This is one reason why Turbo Pascal is used in this thesis as a target environment for the generated code. The interface will, however, generate a correct PASCAL code even if the source code follows standard Pascal.

The compiler will always refer to the Turbo Pascal compiler even though, from a theoretical point of view, it should be any Pascal compiler. Similarly, since there is no standard representation of Arabic data, i.e., available and implemented, we use the BCON operating system, using ARCII, as the internal representation of data in memory.

A. MAJOR CONCEPTS

The interface looks at any piece of code (token) as one of several types. These types are:

- Literal string
- Comment
- Integer
- Identifier
- Functional operator.

Literal strings are constants and the interface does not alter the ASCII value. The comments are surrounded by '(*' and '*)' in Arabic equivalent codes. Integers are important

and easy to handle since there is an isomorphic relationship between Arabic integer tokens and Latin. A real number token is made up of two integer tokens separated by a functional operator. An identifier is any legal name in Pascal, either a reserved word or user-defined. Functional operators are all the codes that are used for addition, brackets, pointer arrows, etc. In setting the specification for programming in Arabic Pascal, the optimum goal is to have a one-to-one relationship between the Latin and the Arabic special characters. Also we want to avoid overloading the use of special characters.

1. Literal Strings

Literal strings are used for assigning into string variables and for read and write commands. Strings are used to interact with the user in an application and understand the performance of the program. Therefore we do not alter these strings. The literal string is any string of characters surrounded by single or double quotes. It is the programmer's responsibility to verify the content of an assigned string. The literal string can have any character of the entire set 80 hex ... FF hex.

2. Comments

The comment length is limited to one line. The comment is enclosed by an opening bracket followed by an asterisk, and ends with an asterisk followed by a closing bracket. When the translator encounters the beginning of a

comment it looks for the end of the comment. The comment is considered as one token. The translator will not alter the content of the comment since it is for the use of the programmer only.

3. Integers

Integers are any consecutive digits from 0-9 with no separation in between. For example, the integer printing format "2245:6" is considered as three tokens as far as the translator is concerned. The first token is the integer "2245," the second is functional operator ":", the third is the integer "6" token.

Real numbers are made up of three parts as one would expect. They are integer token, Arabic numeric comma, and integer token.

4. Identifiers

All legal Pascal names fall under this category. This includes reserved words, and variable names. The token is identified first as an identifier, then looked up in the reserved words group. If it is not in the list then it is a variable name. Variable names include variables, labels, procedure and function names. When an identifier is encountered and it is not a reserved word, then it is given an identifier number. The identifier number is stored with other information about the token in a hashing scheme in a symbol table. The token is looked up in the symbol table. If it is not entered, then it will be entered in the

beginning of the link list of the same hash key. Since the primary user of the translated code is the compiler, the program will have meaningless variable names. However, the translator will generate a file called "DICTIONARY" containing each identifier number and the Arabic token associated with it.

5. Functional Operator

Tokens are identified by separators and terminators. Blanks are separators, as well as other codes that have a function other than being separators. For example, the plus and minus sign as well as the up_arrow symbol in PASCAL are separators. If, for example, the variable root^.left_sun was the Arabic token it will be translated into something like, id_1^.id_2, where the identifier numbers are entered for the Arabic tokens.

The scope of the variables will distinguish frequently occurring variable names. If id_1 occurred in two declarations, the compiler will distinguish between two occurrences of id__1, depending on the location of the declarations. Therefore the translator does not need to concern itself with multiple uses of the same name.

B. OPERATING PRINCIPLES

The translator goes through several phases and each phase has a sub-task. The process begins with the name of the Arabic source code file. The file is opened, the target output file is initialized and a dictionary table file is

opened. The second phase fills a buffer with a code segment of the source code, a line at a time. The line is broken into tokens. Each token is given a type and then translated. The cycle is repeated for each lineup to the end of the source file.

1. File Opening and Initializing Phase

The program starts with the prompt for the user to input the source file name. The file name is checked for existence and then reset for reading. The file name is used to open two more files, the dictionary file, and the output file. The initialization is concerned with the hash table that has information regarding the record structure of the identifier's symbol table. The rest of the parameters are optional features such as to list the source comments with the output code. Another feature is debugging for tracing the program in the translation while the translator is scanning and translating the source code. Both comments and debugging features should be easily set at any point in the source code. The rest of the parameters, for example, line number, identifier number, are initialized.

2. Reading and Decomposing the Source Code

An input buffer is filled from the source code and scanned. A line at a time is read from the buffer and checked for special instructions (directives) for the translator. If the line is not a directive, it is checked to see if it is a comment. If the line is a comment or

starts with one, then the comment is either omitted or written out depending on the comment option. The comment option is a Boolean variable set by the user within the program source code, to either omit or write out the comment tokens in the generated file. The line, or the remainder of the line then, is decomposed into tokens. Tokens are identifiers, integers, blanks, or special characters. Identifiers are either reserved words or user-defined identifiers. Reserved words are matched with their associate Latin reserved word. User-defined identifiers are given a label number in the sequence of their first appearance, if it does not already exist. Integer tokens are scanned and each digit is mapped into its matching Latin digit. Special characters are given their equivalent Latin characters, such as Arabic and Latin semicolon. Blanks are copied as it makes for better formatting of the generated code.

The investigation of the token type is based on the first character of the next token in the input buffer. For example, if the first character is a:

- Letter: Then investigate the possibility that it is an identifier.
- Digit: The token must be an integer.
- Other: Then it must be a special character.

In this phase only the identifiers are translated. When a user-defined identifier is encountered, and, if it has not previously been recognized, it is given the next identifier

number in sequence. Reserved word tokens are stored in a constant table, in a record format. Each record has an Arabic word and the matching Latin one. Any identifier token is first looked up in the table. If found then the index of the matched record is passed back to the main program. The integer tokens are given the type integer and passed back to the main program. If any of the above is not true then we get one character and pass it individually.

In short, each token is given a token type, length, and passed back to the main program. Reserved words are passed back with the match index additionally. Identifiers are also inserted in the symbol table. If not found, their identifier number (in Latin characters) is passed back.

3. Token Translation Phase

The tokens are translated into Latin-based on the token type. The integer tokens are translated by mapping each Arabic (Eastern Hindu) digit into its Latin (Western Hindu) associated digit. Reserved word tokens are translated by writing their matched Latin reserved word, using the match index found earlier. User-defined identifiers are replaced by the identifier number assigned to it. The rest of the special characters are looked up in a "CASE OF" (a PASCAL control statement) list or assigned into a constant table (array). This model uses a case statement. As each user identifier is trans-lated and written out in

the output file, it is also written out in the dictionary table along with the Arabic token associated with it.

4. File Closing and Ending

The last phase is to close the source file, dictionary, and the generated output file. This phase will only be reached at normal program execution. The program will terminate if there is a character code not in the range of the Arabic alphabet defined by the bilingual operating system. Long tokens and comments will cause errors and should stop the translation, since translating a comment makes no sense.

C. DESIGN GOALS

The interface is supposed to generate from any Arabic source code a Latin code in PASCAL syntax. The Arabic programmer must master PASCAL programming in his native language. Essentially little syntax and no semantic checking will be performed on the source code. The compiler job is to scan and perform the syntax and semantics on the translated code. Some help must be provided for tracing, and debugging should be incorporated into such an interface. The compiler gives the error messages in Latin. This could be utilized in several ways. One way is to keep the line numbers of the source code and the generated code as close as possible. The error messages usually are stored in a text file and can be translated. This, along with the line number of the error location, can be combined to give the location and type of the source code error.

A second way, if the error messages cannot be translated in their file, is to translate the error messages and return them out with the error number. The Arabic programmer can look up the error number in Latin and the line number of the error, then look up the translation of the error and explanations. In both ways a few hints regarding the errors and possible causes should be provided to the user.

D. DESIGN LIMITATIONS

The design does not use or handle diacritics at all as far as reserved words are concerned. This could cause error and personal interpretations of how the reserved word is written. Since most reserved words are clear once read, the user must not type any vowels with the reserved words in the program. Similarly, to not duplicate the translation of a single user-defined identifier, and eliminate the complication of debugging of such cases, the user should not use the vowels in his defined identifiers. The diacritics may be used in literal strings and headings of reports. Several factors may affect and prevent the use of diacritics. Some sorting routines sort independently of diacritics. Since vowelization can upset the sorting order and the rules for sorting the same name with different vowelization. A second reason is that the location of the vowelization of the character is not standardized. A third reason is that the resolution of terminals is poor and hard on the eye to

distinguish, for example, between the "FAT'HA" and the single quote symbol in printed or displayed form.

The design therefore will not handle vowels in the Arabic source code. However, it should be noted that the option of including the diacritics requires few changes in the design, and a lot of attention from the Arabic programmer. The attention is required to rewrite his own sorting routine that sets the ARCII value for the vowelized source code. Also the programmer must be consistent with his use of vowels with identifiers for the above reasons.

The display and print justifications cannot be controlled easily within the program since the bilingual operating system does not use a standard unified code for Arabic display and print mode. For example, in BCON, the operating system used for the implementation of this thesis, if you are editing an Arabic screen mode then the curser in the entire code will start at the far right of the screen. This right justification is for the Arabic format and indentation in Arabic texts. Therefore, if you exit the editor you must set the screen mode to Latin screen mode, otherwise the "C:>" prompt will be displayed in the far right of the screen. So for the sake of simplicity to the user and consistency on the behalf of the generated codes, the display codes are left out of the translator control and are under the control of the display system of the bilingual operating system. The modes can be set with an external

escape code to the printer or a sequence of key strokes to set the screen to Arabic mode.

These limitations can be resolved once there is a standard set. I believe the bilingual operating system should by default handle the justification issue, and allow the user to turn this option off. This is in the range of two to five years to come in the industry involved with Arabic text handling.

VI. PROGRAM MODEL

A. INTRODUCTION

The Lexical Translator program is intended to be simple, flexible, and to demonstrate feasibility of the concept. Speed and efficiency was not a primary goal. Features can be added as needed based on the response of usérs of the program.

The program will require the supervision of a good PASCAL programmer to assist the compilation and execution of the translated code. The assistance could be achieved by simple detailed instructions on how to use the program to generate output code.

B. PROGRAM ENVIRONMENT

The Translator is developed under a certain environment, and until there is a unified standard for a bilingual operating system, program portability and compatibility will be limited.

1. Hardware Environment

The program is developed using an IBM XT personal computer, It can be just as well developed using an IBM PC Jr., or IBM At. The IBM XT has 640 kilobytes of RAM memory, 20 megabyte hard disk, two half height floppy disks, and the ALIS Inc., graphics board. The board is made up of two boards back to back. The first board is a Paradise color

graphics board. The second board is on top of the paradise board and it has the Arabic character generator and the necessary connection circuitry needed. The two boards fit in one slot on the mother board of the XT computer.

The keyboard is an IBM PC keyboard with cap stickers for the keys. Each sticker has two to four different characters, for Arabic and Latin. The keyboard layout is displayed in Appendix D.

An Epson FX 85 dot matrix printer is used for the listing of the program. The printer has an Arabic driver to display Arabic characters.

2. Software Environment

ALIS Inc., BCON bilingual operating system was used in developing the thesis program and test runs. BCON resides in low memory using about 20K bytes. The BCON is supposed to be transparent to the DOS operating system. DOS stands for Disk Operating System used by IBM microcomputers. The BCON operating system requires special skill and more than average user knowledge. BCON is mainly required for generating the Arabic fonts, and interpreting and mapping the key strokes to their associated ARCII values. The interpretation and mapping are performed under the Arabic mode only. The Arabic characters are stored as hex values ranging from 80 hex up to FF hex. This range of values is reserved for graphics under the DOS operating system. This

means any Arabic character code is considered a graphic character in the absence of BCON.

An important concept must be pointed out. The presence of BCON is to display the right form, font, and the indentation of Arabic text. So with minimum skill, a programmer can develop, review, correct Arabic characters in any DOS compatible machine. Then the result can be displayed under BCON, where BCON can interpret the graphics character as ARCII code, and display the correct textual form of the ARCII code by sending the appropriate display code to the terminal or the printer.

When writing long Arabic texts, it is much easier to do so under BCON, with the aid of an Arabic word processor. The simple EDLIN editor available on DOS distribution disk, or Turbo PASCAL editor of version 2.1 and below, will work also. There is some limitation to what one can use under BCON and still display Arabic characters. BCON requires two conditions for compatibility when using any application. First BIOS interrupts[2] 16 Hex and 10 Hex are called to access the keyboard and the screen respectively. Second, the application must handle 8-bit characters. [Ref. 2: p. 3-1]

Turbo PASCAL version 2.1 was used to write the main program and resource file. The printer interface, called

---

[2]Information about the interrupts can be found in DOS technical manuals for personal computers.

MPD by ALIS [Ref. 2], is implemented for several printers. The name stands for Multi Printer Driver. The MPD was used to drive the Epson FX 85 to display the Arabic characters in the program listings, and sample tests (Appendices H, I).

C. PROGRAM BODY

The Lexical translator is designed to be easily modified and should be done when the updated version of BCON utilizing the unified standard code set is available. The program is modular and could be rewritten in "C" or FORTRAN. The program is designed to generate a correct output file from a correct input source file. The program will not interpret the result and the programmer must exercise creativity and care as his/her programming advances, to assure correct results and clear output.

The printable output of any developed program is either a string of characters, or mathematical results. Since any string assignment is not altered, this will result in no difficulties for string output. If the result is a real or integer number, the result will be displayed based on the BCON digit mode. The program did not concern itself with numerals since all the users are familiar with the Western Hindu Numerals (Latin). Also, BCON has an option that allows the user to swap the digits in the operating system environment. So for BCON, analyzing the results of numeric calculation will be duplicating the same work. This may be a limitation under an operating system other than BCON.

1. Program Files

The program has two main files that are used for the generation of the output code. The main file and the resource file. The main file contains constant declarations, data structure declaration, variable declarations, procedures and functions, and main program body.

The resource file has the assignments of a constant array declared in the main program and is used as an include file. The resource file has a subset of the reserved words and standard function names. The resource file is a very useful modular concept since you can replace the PASCAL resource file with one for the language "C". With minimum changes in the constants and directives one could use one Translator with several resource files, one for each language, to Lexically translate from Arabic to one of many Latin compilers syntax. This program focus is on the Turbo PASCAL syntax.

2. Generated Files

The translator will generate two files:

- A Dictionary file with the same name and "DIC" extension.
- An Output file with the same file name and "PAS" extension.

The program will generate the desired output in the "PAS" file. The dictionary file will be updated each time an identifier is encountered for the first time. User-defined

Arabic identifiers are translated to identifies of the form "id_000 ... id_999."

3. Key Variables and Data Structure Declarations

The external file "Resource.Pas" is an assignment of a constant array. Each element of the array is a record. The record has two components. The first component is the Latin reserved word or function name, and the second component is the Arabic translated (matching) word.[3]

The user-defined identifiers are handled by a hashing scheme and a symbol table. The decision was to demonstrate an efficient way to store and retrieve identifiers. The lexical translator will be constantly looking up any non-reserved identifier in a symbol table to insert it or to get its Latin match if predefined. To improve efficiency, the program uses a direct chain Hashing scheme [Ref. 3:p. 45].

The identifier is passed to a function and given a key number by Function_KEY. With a hashing formula the function calculates the key number of the identifier. The key number is a location in the Hash table. The content of this specific location is pointer to a word_record which either contains the word or is where a new record should be inserted in case the word was not found. Words having the same key number will be linked together in a linked list.

---

[3]The translation is in no way a standard or professionally translated. The translation was made for demonstration purposes.

The incident of having several words with the same key number decreases the efficiency of Hashing (see Ref. 3 on how to avoid Hashing collision and when to use Hashing). The word record has the following.

Id_No	-	the identifier number in the sequence of insertion.
Length	-	number of characters of the identifier.
Lastchar	-	location of the last character in the symbol table.
Nextword	-	pointer to the next identifier with the same key number.
Latin_Id	-	the Latin identifier assigned to the identifier.

With the above word (identifier) information, we can locate the word in the symbol table. The spelling table is declared as an array of 5000 characters. The size is an estimate and can be changed as one can predict a closer estimate. The symbol table is implemented as a linked list and its size can vary dynamically so as to be as large as necessary.

The translator looks for tokens using two methods. The first method uses a pair of delimiters to identify the token. The pair define the beginning and end of a token. Token classes that can be identified by this method are comments, literal strings, and directives.

The second method recognizes a token by its first character. Examples of this class are integers, and identifiers. The second method includes tokens with one character

such as separators and terminators. Both separators and terminators will be referred to as delimiters throughout the program. The delimiters are defined in a constant set. The Hex values of the set can be interpreted with the aid of the ARCII table (Appendix D).

Errors are a user-defined data type. Types of errors are, for example, long_token, long comment, and long_literal string. All of the above errors are expected to occur as a result of failure of the programmer to end a comment or literal string.

The token types are defined to be one of the following:

- Blanks
- Reserved_word
- Identifier
- Literal_String
- Control_Code
- Comment
- Integer
- Functional_Operator
- Unclassified
- Illegal

These are the main declarations of the program. The definition of the tokens and assignments of the variables will be covered in the following sections.

4. Token Classes I and II

Class I tokens are recognized using the first method. This includes the following types of tokens:

Literal_String: This token begins with Arabic quote mark, single or double, and ends with it. The Hex values are 97 Hex and A2 Hex.

Comments : Begins with right bracket followed by asterisk and ends with an asterisk followed by left bracket.

Directives : Are strings in curly brackets. This feature is for debugging. The directives will allow the user to choose between commented Latin source, with original comments, and debugging option to display on the monitor the tokens and their types.

Class II covers the identifiers, including reserved words, and integers. The remainder of token types will be reviewed shortly.

Identifiers and Reserved Words: Begin with an Arabic letter followed by an optional number of underscore, digit, or other Arabic characters.

Integers : Begins with digit and ends with any non-digit character.

The remainder of the token types are Functional_Operator, Illegal, and Unclassified. Functional_Operator tokens are the arithmetic operators, brackets, asterisk, decimal digit, semicolon, colon, pointer '^', etc. The illegal token is the token that exceeds its defined length. This condition is used to set an error message to pass to the user about the location of an error. An Illegal token is also set if the Hex code is less than 80 Hex. The legal range is 80 ...

FF Hex. The control code is any escape code or function call within the range of Arabic characters ranging from 80 Hex ... FF Hex. The Unclassified token type is used as the value before it is determined.

D. PROGRAM MODULES

The Lexical translator will call several procedures and functions in the process to generate the desired code. The main body of the program calls several procedures and functions. The program modules and their locally declared procedures and functions are as follows:

- Open_File
- Initialize
- Fill_Buffer
- Token_and_Type
  - Blank
  - Comment
  - Literal_String
  - Integer_Token
  - Identifier_Token
  - Reserved_Token
  - Special_Char_Token
  - Control_Char_Token
- Map_Identifier_To_Latin
  - Search
    - Hash_Key
    - Insert: calls Id_No

Found

Latin_Integer

Get_Latin_Spec_Char

Print_Error_Messages

1. Open_File

The program starts by calling the Open__File procedure. The procedure will prompt the user for the name of a file to translate and verify that the file does exist. The second part is to open the input file for reading, reset the Output file for writing, and the Dictionary file for writing.

2. Initialize

Initialize procedure will set all the hash table pointers to nil. The nil values are used to indicate that there are no words with that key number yet initialized. It will also set the initial values of global variables. The module is called once at the beginning of the program.

3. Fill_Buffer

This procedure will get a line of source code, keep track of the line number of the source code, and set the line size of the source code. This module is continuously called by the main program until the end of the source file is reached.

4. Buffer_Empty

This function will test to see if the variable Next_Loc, which represents the next token location on the line,

is pointing beyond the Line_Size variable. This case will set the function to true, causing the main program to call the Fill_Buffer procedure to refill the buffer. This module is called continuously by the main program.

5. Token_And_Type

When called, this procedure is passed a line of source code and the location of the first character of the token to be fetched. The procedure gets the token and gives it a type. The procedure initially sets the type of the token to Unclassified and through several calls, tries to analyze the type of the token. The first convenient check is for Comments. It should be noted here that one would like to place the most likely type check at the beginning to reduce time of analysis of the token type. Another reason for searching for comments first is because they are the only type that requires two characters in the beginning and the end of the token. The rest can be predicted just by inspecting the first character.

If the token type is not set to Comment, then the module calls several modules with a case statement. The modules are called based on the first character after the last token read. The Next Location variable points at this character in the input line buffer called "Line." The possibilities are:

FIRST CHARACTER	LIKELY TOKEN TYPE
Arabic space	Blank(s)
Double or Single Quotes	Literal string
Arabic Digit	Integer
Arabic Letter	Identifier
Function Code	Control Char
Other Characters	Special Characters

Each possible token type above represents a module. The module will be called to set the type of the token.

Looking at each module called by Token_and_Type, they all set the token type and the length of the token. All likely token types except for Literal Strings and Comment will not set any error flags, since one character will satisfy their requirements. For example, Blanks, Integers, Identi-fiers, Control Characters, and Special Characters all could be one character long. When Literal String and Comment modules are called, they must begin and end with a predeter-mined pattern. So an open comment for longer than line length is an error, and the same for a long literal string token. Token_And_Type only examines the Line Buffer charac-ter and does not consume it. The called modules assign the character to the Token Buffer and advance the pointer of the Line Buffer one character. When a successful, token type is assigned the module sets the token length. PASCAL uses the first array location to store the length of the assigned characters in bytes.

The behavior of the modules called by Token_and Type, are summarized below:

Blanks: Will keep consuming the Line Buffer blanks (Arabic and Latin) up to a non_blank character is reached. Blanks will set Token Type and Length.

Comment: Consumes the characters within the Arabic characters range, until the comment closing mark is reached. The module will set the error set to long_comment, if any character lies in the Latin alphabet range, including the end of file and carriage return (ASCII OD, OA Hex). The error is long comment since the comment is restricted to one line long. Comment alters the opening and closing bracket of the Arabic comment token. The characters are the Arabic opening brackets, closing brackets, and the asterisk, having the Hex values A8, A9, and AA respectively.

Literal_String: The module will be called in case the next characters are single or double quotes. The module will expect to be terminated with the same character it began with. If the matching character is not reached before the end of the line it is considered an illegal token, and the error set will be assigned the type long token. Valid literal strings will not be altered. However the opening and closing will be translated to single or double quotes accordingly.

Integer_Tok: Stands for integer token, and will be called when a digit is present. The module will keep assigning the Latin digits in the token buffer, and assign the Token__Type Integer to the variable Tok_type.

Identifier_Tok: Will be called when the character is a letter. The single letter qualifies as an identifier alone, or could be followed by an optional number of Arabic underscore, digit, or letter. The module will set the Tok_type to Identifier. The module has no effects on error set, since when called it was a valid token based on the first character of the token.

Reserved_Tok: The module is called when the token found is an identifier. The module will check if

the token is in the reserved words constant array called "Res_Word." If the identifier is a reserved word the index of the table is passed back to the main program.

Control_Char_Tok: The module is called when a BCON function code is the next character in the Line_Buffer. The module assigns one character (code) to the token buffer.

Special_Char: This module assigns one character to the token. The token will always have one character.

When Token_and_Type returns the token type to the main program, a case statement will either call a procedure or do the processing with a compound statement. The blanks will be translated to Latin ASCII code blanks. The returned comment token will be written out as is. Literal strings are written out literally. Reserved words are written as is using the Match_Index in the Res_Word constant array. The identifiers are looked up in the symbol table. If predefined, the token identifier number is returned with it, or else the identifier is inserted in the table and given an identifier number. The module used is called Map_Iden_To Latin.

6. Map_Iden_To_Latin

The Identifier token is received and searched for with a procedure called Search.

7. Search

This module starts by calling the Hash_Key function.

a. Hash_Key

Hash_Key calculates the token key_no with a hash formula. The key number is used to look up the pointer of the word record in the hash table. The word record is a linked list of identifiers of the same key number. All the pointers are initialized to nil at the beginning of the program. If the key number results in a nil pointer value, that means there is no such word in the symbol table, nor any other word with the same hash key number, then Search calls Insert to insert the identifier in the symbol table.

b. Insert

Insert creates a word record at the beginning of the linked list and stores the identifier in the spelling table. Insert makes a call to IDEN_LBL_NO, which uses the global variable ID_NO (sequence of appearance), and assigns an identifier number in the word record.

If the pointer is pointing at a word record, then the first word in the linked list is checked, and so on until there are no more word records in the list or the word is found.

c. Found

The function Found checks if the resulting pointer is pointing at the exact identifier spelling.

If the word record is found then it already has been assigned a specific identifier number which is then passed back to the main program to be written out as the Latin identifier.

8. Latin_Int

The procedure maps each digit of the token to the Latin digit 0...9, and passes back the Latin integer.

9. Get_Latin_Spec_Char

The procedure is to give each Arabic special character its Latin "functionally" equivalent character.

10. Print Errors

Based on the error set, Print Errors will send the error type and the line number in the source code where it was encountered.

E. PROGRAM DIRECTIVES

The program offers two directives. One is the option to keep the source comments in the output file, or the program will omit the comments by default. Two is the option to turn on and off the debug option at any location in the code at the beginning of a line. This option will display the tokens and their types as they are scanned.

The program is demonstrated by a list of test runs to verify the translation of reserved words and special characters. Also a sample of small PASCAL programs are included with their generated files, code and dictionary tables (Appendix I).

E. LIMITATIONS

The program does not allow the user to use the 'Include' directive in TURBO PASCAL. The size of the program is

limited by TURBO PASCAL to 64k, where an additional code could be included as an 'Include' file.

The program is set to handle up to one thousand identifiers. This is a reasonable number in working with TURBO PASCAL since the program size is limited to 64k bytes.

The spelling table is 5000 characters long. That means the total length of all identifiers can not exceed 5000 characters. The programmer can avoid, when writing long programs, exceeding the limit by using short identifiers.

The program will not generate an error flag if a Latin string is found in comments or literal string. This is because both comments and literal strings are not altered.

ARCII provides two commas. The numeric comma is used with real numbers in Arabic, and the Arabic Comma is used, in this specification, as the Latin comma except for the real number case. This is a small hurdle in the case of translating the generated code back to Arabic. The appearance of the two Arabic commas is different. They are 180^{o} out of phase on the vertical axis where the numeric comma looks like the Latin comma. The decision on using both commas was to avoid overloading the use of the Arabic comma.

VII. CONCLUSION

This thesis has tried to narrow the gap between educated Arabic-speaking people and computers in general. The target ages are mid-teenage, and forty-five and above. The majority of these two classes still look at computers as magic. They believe man created them. However they have a hard time believing that man tells computers what to do. With that attitude, the only thing that can convince them is to help them to write small programs and see the results. We are convinced that the majority will get rid of their fear and have the desire to explore this machine.

In short, the topic of the interface between the rich Latin software library, and the Arabic language environment is a promising area in the sense that it will bring those who fear computers closer, and find a more efficient way to get the job or hobby done.

A. CONCEPT FUTURE

The program is simple in concept and to code, but the environment where it is expected to work is not yet standardized. The standards are not widely implemented, nor are the developers of bilingual operating systems very helpful in responding to concerns about hardware compatibility.

Once a unified environment is established, then the concept could be developed further. The goal of this work was to illustrate the feasibility and avoid specific issues of the implementation environment. The program modules were designed to be adaptable and portable for several purposes with little modification. For example:

- For several programming language translations, such as "C," FORTRAN, and BASIC, we only need several resource files and several special character sets, one for each programming language requirement.
- For several code sets, including different languages, we need the concept of a bilingual operating system that uses the upper range of the character set ranging from 80 ... FF Hex.
- The program can work in a Latin-only operating system, to translate source codes that have been edited using Arabic code set values. Also, the generated source could be compiled in the same machine. If the program is interactive, then it needs to run under a bilingual operating system.

B. LIMITATIONS

The bilingual operating system was not well documented as far as how some of the function codes are implemented during editing. Some of the characters have two codes (such as the Arabic multiply sign and the numeric multiply sign). To know which multiply sign is generated when I strike a key, I had to use an editing tool to display the code in Hex values and match the text file and its Hex values.

Right indentation is relative to the editor mode. If you select your screen mode to be Arabic and you read a piece of Latin code, it will be right justified.

The user must be careful reading data files. Some data is readable only in Arabic mode and some data is readable only in Latin. Also the data displayed may have been transformed by the operating system. As mentioned before, the user could use the "SWAP" option for altering ASCII digits and ARCII digits in the DOS environment, or read the digits as a string and change the values into ASCII. This is important in order to perform numerical operations with Arabic digits.

I strongly believe that, with time, standards will be developed with more care and concern for the user. This is the reason we chose not to design the program for a specific system.

It is hoped that this work will benefit other researchers and future thesis students from other countries since a similar concept could be applied to other languages, especially languages descended from Latin.

APPENDIX A

FIGURES

ا ب ت ث ج ح خ د ذ ر ز

س ش ص ض ط ظ ع غ ف ق

ك ل م ن ه و ي

Figure 1. The 28 Arabic Alphabets

ا ب ت ث ج ح خ د ذ ر ز

س ش ص ض ط ظ ع غ ف

ك ل م ن ه و ي ء ى ة

Figure 2. The 31 Alphabets (Optimum Set)

NAME	CHARACTER	NAME	CHARACTER
ALEF	ا	DAD	ض
BA'A	ب	TAH	ط
TA'A	ت	DHAH	ظ
THA	ث	AIN	ع
JEEM	ج	GHAIN	غ
HA'A	ح	FA	ف
KHA'A	خ	QAF	ق
DAL	د	KAF	ك
THAL	ذ	LAM	ل
RA	ر	MEEM	م
ZA	ز	NOON	ن
SEEN	س	HA	ه
SHEEN	ش	WAW	و
SAD	ص	YA	ي

HAMMAZAH	ء
TAAMARBOTA	ة
ALEF_MAQSURA	ى

Figure 3. Arabic Alphabet Names

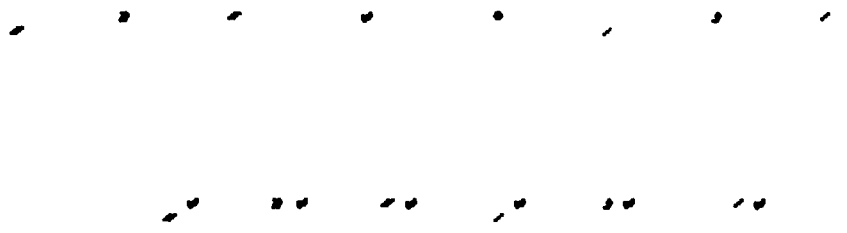


Figure 4. Arabic Diacritics (Vowelization)

৭ ৮ ৬ ৭ ৫ ৪ ৩ ২ ১ ০

Eastern Hindu numerals

9 8 7 6 5 4 3 2 1 0

Western Hindu numerals

Figure 5. Hindu Numerals

تشد الرحال وقت الحج فيقضي الحجاج بجوار المسجد زمنا يؤدون به الصلاة فإذا قدر لك أن تذهب إلى هذا المسجد لراعك أن يجمع الألوف المؤلفة مجمع العبادة حيث يقفون جميعا خاشعين أمام الله

أقبل العرب على فتح الشام واثقين من النصر

الماس نفيس

a. Without Vowels

تُشَدُّ الرِّحَالُ وقْتَ الْحَجِّ فَيَقْضِي الْحُجَّاجُ بِجِوَارِ الْمَسْجِدِ زَمَنًا يُؤَدُّونَ بِهِ الصَّلَاةَ فَإِذَا قُدِّرَ لَكَ أَنْ تَذْهَبَ إلى هَذا الْمَسْجِدِ لَرَاعَكَ أَنْ يَجْمَعَ الْأُلُوفَ الْمُؤَلَّفَةَ مَجْمَعُ الْعِبَادَةِ حَيْثُ يَقِفُونَ جَمِيعًا خَاشِعِينَ أَمَامَ اللهِ

أَقْبَلَ الْعَرَبُ عَلى فَتْحِ الشَّامِ وَاثِقِينَ مِن النَّصْرِ

الْمَاس نَفِيس

b. With Vowels

Figure 6. Arabic Text

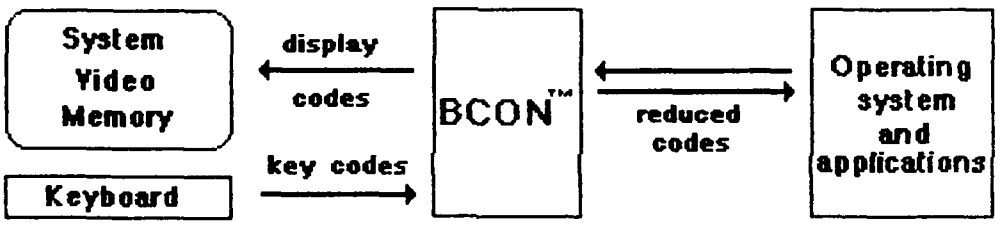


Figure 7. BCON Code Sets

APPENDIX B

TEXAS INSTRUMENTS APPROACH TO
BILINGUAL OPERATING SYSTEM

Philosophy of Bilingual Arabic Latin Implementation on Microcomputer System

Texas Instruments

ARABIC COMPUTER SYSTEMS
PHILOSOPHY

SPECIFIC CHARACTERISTICS OF THE ARABIC LANGUAGE

- ★ ARABIC IS WRITTEN FROM RIGHT TO LEFT
- ★ THERE ARE SOME VARIATIONS IN TYPES OF ARABIC CURRENTLY IN USE IN DIFFERENT COUNTRIES
- ★ THE LANGUAGE IS A FOUR LEVEL ONE. A CHARACTER CAN HAVE UP TO FOUR SHAPES DEPENDING ON ITS POSITION IN THE WORD : ISOLATED, INITIAL, MEDIAL OR FINAL
- ★ ARABIC CHARACTERS ARE JOINED WITHIN A WORD
- ★ NO UPPER CASE EXISTS IN ARABIC

I'll start my presentation by a brief mentioning of some of the characteristics of the Arabic language which have been covered in previous papers and which affect the use of the Arabic language in the computer field.

ARABIC COMPUTER SYSTEMS
PHILOSOPHY

SPECIFIC CHARACTERISTICS OF THE ARABIC LANGUAGE

★ ONLY THREE CHARACTER VOWELS EXIST IN ARABIC :
ALIF ا , OUAOU و , YAA ى .

★ VOWELISATION IN ARABIC IS ALSO PERFORMED THROUGH THE USE OF DIACRITICS. THESE ARE USED :

- IN THE CASE OF SIMILARLY WRITTEN WORDS TO AID THE READER
- IN RELIGIOUS TEXTS INCLUDING THE KORAN
- FOR SCHOOL TEACHING

★ ARABIC LANGUAGE USES INDIAN NUMERICS, WITH THE DECIMAL POINT BEING A COMMA.

★ THERE ARE ARABIC SPECIAL CHARACTERS WHICH INCLUDE THE ARABIC COMMA ، , SEMICOLON ؛ , QUESTION MARK ؟ , ETC.

ARABIC COMPUTER SYSTEMS
PHILOSOPHY

ARABIC ALPHABET

- ★ THE BASIC ARABIC ALPHABET IS COMPOSED OF 28 CHARACTERS
- ★ THE LAMALIF WHICH IS COMPOSED OF TWO CHARACTERS LAM + ALIF IS CONSIDERED AS ONE CHARACTER
- ★ THE HAMZA CAN BE WRITTEN IN MANY DIFFERENT WAYS IN ARABIC DEPENDING ON ITS USE, WITH A VOWEL OR ISOLATED
- ★ IF THESE TWO CHARACTERS ARE TAKEN INTO CONSIDERATION THE ALPHABET IS 30 CHARACTERS
- ★ THE TAMARBOUTA IS A SPECIAL CHARACTER NOT INCLUDED IN THE ALPHABET. IT IS OCCASIONALLY INCLUDED AT THE END OF WORDS DEPENDING ON GRAMMATICAL RULES

ARABIC COMPUTER SYSTEMS
BILINGUAL SYSTEM APPROACH

SOLUTION 1 : CORRESPONDANCE & DIFFERENCES

★ THIS STUDY IS BASED ON THE CORRESPONDANCE AND DIFFERENCES BETWEEN ARABIC CHARACTERS. THE ARABIC ALPHABET MAY BE CONSIDERED AS FORMED OF THREE TYPES OF CHARACTERS :

- TYPE A INCLUDES CHARACTERS HAVING 1, 2, OR 3 POINTS :
  ن ب ت ث ي
- TYPE B INCLUDES CHARACTERS WITHOUT POINTS :
  ك ل م ه و
- TYPE C INCLUDES CHARACTERS HAVING AT LEAST ONE FORM IN EACH CASE :
  ذ د / ج ح خ / ص ض / س ش / ر ز

★ IF WE ONLY CONSIDER THE FORMS WITHOUT POINTS WE CAN REDUCE THE CHARACTERS IN EACH TYPE AND THEN ADD THE POINTS AFTERWARDS

ARABIC COMPUTER SYSTEMS
BILINGUAL SYSTEM APPROACH

SOLUTION 2 : ROOTS & APPENDICES

★ A STUDY BASED ON THE USE OF APPENDICES AND ROOTS TENDS TO REDUCE THE TOTAL NUMBER OF SHAPES BY CONSIDERING A ROOT TO BE USED IN INITIAL & MEDIAL SHAPES TO WHICH AN APPENDIX IS ADDED TO FORM THE FINAL OR ISOLATED SHAPES

TYPE A	TYPE B	TYPE C
جـ + ـر = ـج	سـ + ـر = ـس	بـ + ـا = ـب
حـ + ـر = ـح	شـ + ـر = ـش	تـ + ـا = ـت
خـ + ـر = ـخ	صـ + ـر = ـص	ثـ + ـا = ـث
عـ + ـر = ـع	ضـ + ـر = ـض	فـ + ـا = ـف
غـ + ـر = ـغ		قـ + ـا = ـق
مـ + ـر = ـم		نـ + ـا = ـن

The problem with this solution is what code to give to these apprendices. if they are coded would they be considered as characters in a character count? How would high level languages interpret them? How would special s/w function interpret them? replace — insert — find string.
This is the study which resulted in the actual Arabic implementation on Texas Instruments equipment and which will be explained in this paper.

ARABIC COMPUTER SYSTEMS
BILINGUAL SYSTEM APPROACH

SOLUTION 3 : CONTEXTUAL ANALYSIS

- A STUDY BASED ON THE USE OF SHAPING ALGORITHMS. USING CONTEXTUAL ANALYSIS TO DETERMINE THE PROPER SHAPE OF THE CHARACTER, FOUR GROUPS ARE IDENTIFIED
  - GROUP 1 ONE SHAPE PER CHARACTER
  - GROUP 2 TWO SHAPES PER CHARACTER
  - GROUP 3 THREE SHAPES PER CHARACTER
  - GROUP 4 FOUR SHAPES PER CHARACTER
- POSSIBLE APPROACHES
  - ONE-KEY ONE-SHAPE SIMPLIFIES THE SOFTWARE BUT USUALLY LIMITS THE SET OF ARABIC CHARACTERS AND CREATES A COMPLEX KEYBOARD SINCE ALL THE ARABIC CHARACTER SHAPES MUST BE PRESENT ON THE KEYBOARD.
  - ONE-KEY MANY-SHAPES IMPLIES MORE SOPHISTICATED SOFTWARE BUT SIMPLIFIES KEYBOARD & USER INTERFACE

Of these 2 approaches the 2nd one has been chosen and this will be covered in the following slides.

APPENDIX C

DS9900 BILINGUAL COMPUTER
SYSTEM BY TEXAS INSTRUMENTS

ARABIC COMPUTER SYSTEMS
DS990 BILINGUAL SYSTEM

COMMERCIAL COMPUTING REQUIREMENTS FOR THE MIDDLE-EAST

- ★ BILINGUAL LATIN/ARABIC DATA INPUT & OUTPUT
- ★ COBOL DRIVEN APPLICATIONS
- ★ BILINGUAL PRINTING
- ★ BILINGUAL SORT/MERGE

SPECIAL PRODUCTS DEVELOPPED TO MEET REQUIREMENTS

- ★ BILINGUAL DATA ENTRY TERMINAL
- ★ BILINGUAL MATRIX PRINTER
- ★ BILINGUAL LINE PRINTER

SOFTWARE

These handle both in the natural manner + software simplified k/w for operators + high level languages easy handling.

ARABIC COMPUTER SYSTEMS
DS990 BILINGUAL SYSTEM

CHARACTERISTICS OF BILINGUAL DATA ENTRY TERMINAL

★ BILINGUAL VIDEO DISPLAY UNIT

– THE CHARACTER GENERATOR ROM GENERATES 7 × 8 MATRIX FOR ALL STANDARD ASCII CHARACTERS AND 128 ARABIC SHAPES

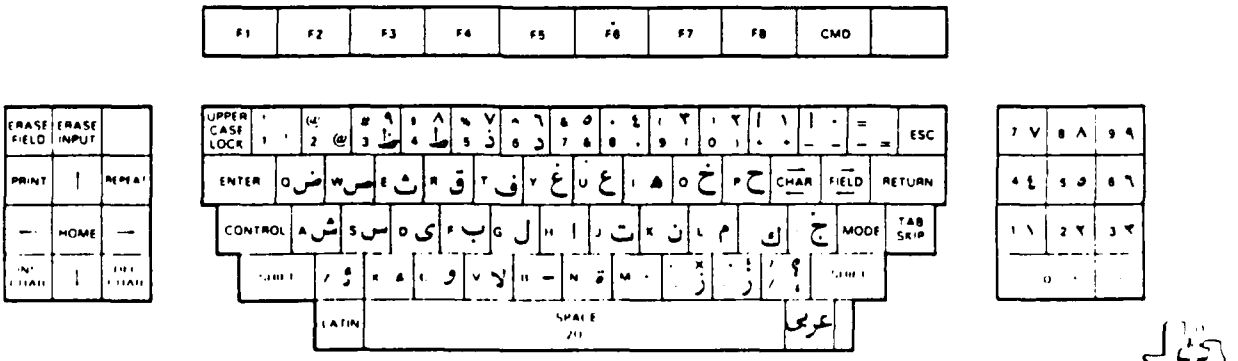
A 7 × 10 MATRIX IS USED FOR INTRICATE ARABIC CHARACTERS

Latin & Arabic can be displayed on the screen at the same time.

ARABIC COMPUTER SYSTEMS
DS990 BILINGUAL SYSTEM

★ BILINGUAL KEYBOARD

- PROVIDES 5 MODES OF OPERATION : ARABIC, LATIN, SHIFT, UPPERCASE & CONTROL. IT CONSISTS OF 91 KEYS
- PROVIDES THE USER WITH THE CAPABILITY OF ENTERING ARABIC AND/OR LATIN DATA WITHOUT CONSTRAINTS
- KEYBOARD MULTIFUNCTION CAPABILITY IS PROVIDED BY A MODE SELECTION KEY AND TWO CHARACTER SET SELECTION KEYS
- DATA IN EITHER LANGUAGE CAN BE ENTERED IN EITHER MODE
- THE KEYBOARD GENERATES 7-BIT CODES FOR LATIN AND 8-BIT CODES FOR ARABIC



* Basic placements of Arabic key like typewriter.

ARABIC COMPUTER SYSTEMS
DS990 BILINGUAL SYSTEM

ARABIC CHARACTER SHAPING

- ★ 32 BASIC ARABIC CHARACTERS ARE GENERATED BY THE KEYBOARD
- ★ A CONTEXTUAL ANALYSIS OF THE ARABIC DATA IS PERFORMED BY THE CONTROL PROGRAM TO DETERMINE THE CORRECT SHAPE OF THE CHARACTER TO BE DISPLAYED
- ★ IN TOTAL THE TERMINAL CAN DISPLAY 115 SHAPES

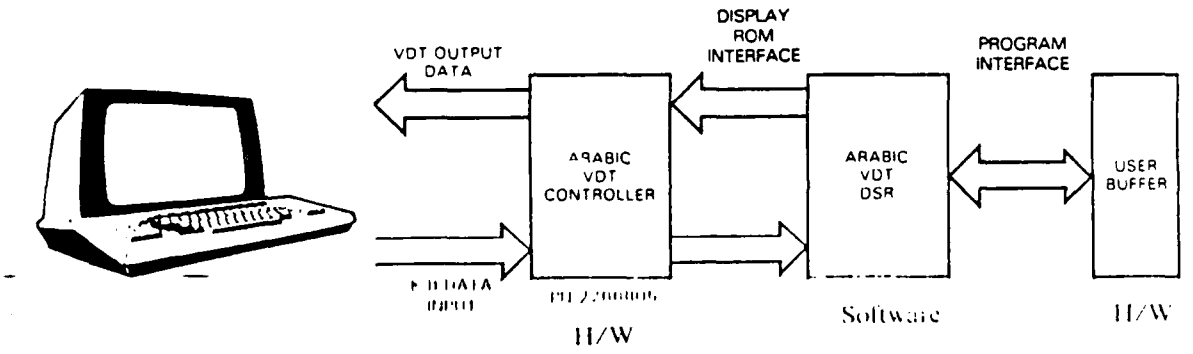
EXAMPLE OF SHAPING PROCEDURE

TYPE THE WORD	:	يتكلم
ENTER YAA	:	ي
TA	:	يت
KAF	:	يتك
LAM	:	يتكل
MIM	:	يتكلم
SPACE	:	يتكلم

ARABIC COMPUTER SYSTEMS
DS990 BILINGUAL SYSTEM

DEVICE SERVICE ROUTINE INTERFACE OVERVIEW

★ THE DEVICE SERVICE ROUTINE IS CONTROL SOFTWARE BETWEEN THE USER'S PROGRAM AND THE VIDEO DISPLAY TERMINAL (VDT)



System flexibility by Software implement.

ARABIC COMPUTER SYSTEMS
DS990 BILINGUAL SYSTEM
BILINGUAL TERMINAL PROGRAM INTERFACE

				b8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b4	b3	b2	b1		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p	ERASE FIELD	HERE IS	ENTER		ا	ظ		
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q	ERASE INPUT	BREAK			ب	ع		
0	0	1	0	2	STX	DC2	"	2	B	R	b	r	HOME	F1			ت	غ		
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s	TAB	F2		!	ث	ف		
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t	DEL CHAR	F3		)	ج	ق		
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u	SKIP	F4		(	ح	ك		
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v	INS CHAR	F5		=	خ	ل		
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w	FIELD ←	F6		+	د	م		
0	0	0	0	8	BS	CAN	(	8	H	X	h	x	CHAR ←	F7		–	ذ	ن		
0	0	0	1	9	HT	EM	)	9	I	Y	i	y	↑	F8		SP	ر	ه		
0	0	1	0	A	LF	SUB	*	:	J	Z	j	z	CHAR →	PRINT		×	ز	ة		
0	0	1	1	B	VT	ESC	+	;	K	[	k	{	↓	CMD		÷	س	و		
0	1	0	0	C	FF	FS	,	<	L	\	l	|	FIELD →	ITI		.	ش	ى		
0	1	0	1	D	CR	GS	-	=	M	]	m	}				؟	ص	ئ		
0	1	1	0	E	SO	RS	.	>	N	^	n	~				.	ض	ء		
0	1	1	1	F	SI	US	/	?	O	_	o	DEL				—	ط	ى		

* Basic Character Set.

ARABIC COMPUTER SYSTEMS
DS990 BILINGUAL SYSTEM

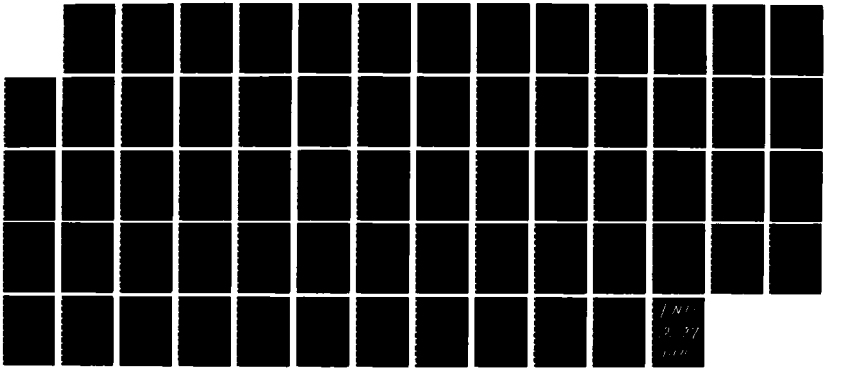
BILINGUAL TERMINAL DISPLAY ROM INTERFACE

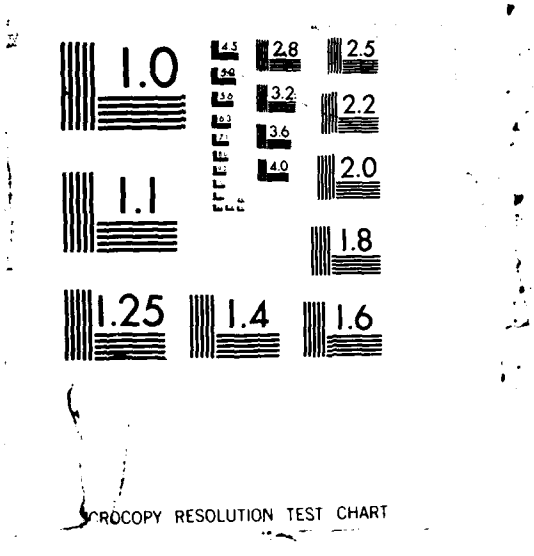
b4	b3	b2	b1		b8	0	0	0	0	0	0	0	0	١	١	١	١	١	١	١	١
					b7	0	0	0	0	١	١	١	١	0	0	0	0	١	١	١	١
					b6	0	0	١	١	0	0	١	١	0	0	١	١	0	0	١	١
					b5	0	١	0	١	0	١	0	١	0	١	0	١	0	١	0	١
					0	١	٢	٣	٤	٥	٦	٧	٨	٩	٠	A	B	C	D	E	F
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p	ع	ز	ى	ت	ـ	ش	ء	٠
0	0	0	١	١	SOH	DC1	!	١	A	Q	a	q	غ	ز	غ	ة	ز	ش	ء	١
0	0	١	0	٢	STX	DC2	"	٢	B	R	b	r	ع	ز	ا	ة	ز	ش	ء	٢
0	0	١	١	٣	ETX	DC3	#	٣	C	S	c	s	غ	ب	(	ث	ص	ض	ئ	٣
0	١	0	0	٤	EOT	DC4	$	٤	D	T	d	t	ـ	ـ	ـ	ـ	ـ	ـ	ـ	٤
0	١	0	١	٥	ENQ	NAK	%	٥	E	U	e	u	ـ	ـ	ـ	ـ	ـ	ـ	ـ	٥
0	١	١	0	٦	ACK	SYN	&	٦	F	V	f	v	ف	م	)	د	ب	ق	ة	٦
0	١	١	١	٧	BEL	ETB	'	٧	G	W	g	w	ـ	ـ	ـ	ـ	ـ	ـ	ـ	٧
١	٠	٠	٠	٨	BS	CAN	(	٨	H	X	h	x	ـ	ـ	ـ	ـ	ـ	ـ	ـ	٨
١	٠	٠	١	٩	HT	EM	)	٩	I	Y	i	y	ـ	ـ	ـ	ـ	ـ	ـ	ـ	٩
١	٠	١	0	A	LF	SUB	*	:	J	Z	j	z	لا	ا	ا	ذ	ط	ك	:	×
١	٠	١	١	B	VT	ESC	+	;	K	[	k	{	ن	لا	أ	ذ	ظ	ك	=	÷
١	١	0	0	C	FF	FS	,	<	L	\	l	|	ح	و	لا	ـ	ظ	٥	٤	٠
١	١	0	١	D	CR	GS	-	=	M	]	m	}	ر	و	ـ	ـ	ظ	٥	٤	؟
١	١	١	0	E	SO	RS	.	>	N	^	n	~	ر	و	ـ	ـ	ظ	٥	ـ	ـ
١	١	١	١	F	SI	US	/	?	O	_	o	DEL	ر	ن	ـ	ـ	ـ	/	ـ	%

Problems of Arabic Numerics must use ASCII numeric code for COBOL, FORTRAN.

AD-A175 145 LEXICAL TRANSLATOR FROM ARABIC TO LATIN IN PASCAL ENVIRONMENT(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA S S ALJUHAIMAN SEP 86 2/2

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
CROCOPY RESOLUTION TEST CHART

APPENDIX D

BCON BILINGUAL OPERATING SYSTEM BY ALIS INC.

Default Reduced Codes

00-7F	Latin characters identical to original ASCII set with the exception of the following two characters
0E	Function code Set Bilingual Screen Operating Mode (Imaged as Latin space)
0F	Function code Set Latin-Only Screen Operating Mode (Imaged as Latin space)
80	Numeric* space
81	= Arabic** number sign
82	× Numeric multiply sign
83	& Arabic ampersand sign
84	Arabic apostrophe sign
85	% Numeric percent sign
86	÷ Numeric divide sign
87	(Numeric left parenthesis
88	) Numeric right parenthesis
89	+ Numeric plus sign
8A	- Numeric minus sign
8B	Numeric less than sign
8C	= Numeric equals sign
8D	Numeric greater than sign
8E	Function code Set Arabic Screen Language Mode (Imaged as Arabic space)
8F	Function code Set Latin Screen Language Mode (Imaged as Arabic space)
90	Function code Set Arabic Line Language Mode (Imaged as Arabic space)
91	Function code Set Latin Line Language Mode (Imaged as Arabic space)
92	@ Arabic commercial at sign
93	Arabic left square bracket
94	Arabic right square bracket
95	Arabic upward arrow head
96	Arabic underline
97	Arabic reverse apostrophe
98	Arabic left curly bracket
99	Arabic vertical line
9A	Arabic right curly bracket
9B	~ Arabic tilde
9C	(reserved)
9D	(reserved)
9E	Function code Set Line Boundary (Imaged as Arabic space)
9F	(reserved)

(*): Numeric means character is Arabic but has intrinsic right spacing (i.e. will be considered part of a numeric string).

(**): Arabic means character has intrinsic left spacing.

A0		Arabic space
A1		Arabic exclamation mark
A2		Arabic quotation mark
A3	×	Arabic multiply sign
A4	-	Arabic dollar sign
A5		Arabic percent sign
A6		Arabic period
A7	÷	Arabic divide sign
A8		Arabic left parenthesis
A9	)	Arabic right parenthesis
AA	*	Arabic asterisk
AB	+	Arabic plus sign
AC	،	Arabic comma
AD	-	Arabic minus sign
AE	,	Numeric comma
AF		Arabic solidus
B0	٠	Arabic digit 0
B1	١	Arabic digit 1
B2	٢	Arabic digit 2
B3	٣	Arabic digit 3
B4	٤	Arabic digit 4
B5	٥	Arabic digit 5
B6	٦	Arabic digit 6
B7	٧	Arabic digit 7
B8	٨	Arabic digit 8
B9	٩	Arabic digit 9
BA	:	Arabic colon
BB	؛	Arabic semi-colon
BC	<	Arabic greater than sign
BD	=	Arabic equals sign
BE	>	Arabic less than sign
BF	؟	Arabic question mark

C0	ـٰ	TAIL
C1	ــ	KASHIDA
C2	ـّ	SHADDAH
C3	ـْ	SUKUN
C4	ـَ	FATHA
C5	ـّـَ	SHADDAH FATHA
C6	ـً	FATHATAN
C7	ـّـً	SHADDAH FATHATAN
C8	ـُ	DAMMAH
C9	ـّـُ	SHADDAH DAMMAH
CA	ـٌ	DAMMATAN
CB	ـّـٌ	SHADDAH DAMMATAN
CC	ـِ	KASRAH
CD	ـّـِ	SHADDAH KASRAH
CE	ـٍ	KASRATAN
CF	ـّـٍ	SHADDAH KASRATAN
D0	ء	HAMZAH
D1	ا	ALEF
D2	ٱ	WASLA ON ALEF
D3	أ	HAMZAH ON ALEF
D4	إ	HAMZAH UNDER ALEF
D5	آ	MADDAH ON ALEF
D6	ب	BA'A
D7	پ	PEH
D8	ة	TA'A MARBUTA
D9	ت	TA'A
DA	ث	THA'A
DB	ج	JEEM
DC	چ	SHEEM
DD	ح	HA'A
DE	خ	KHA'A
DF	د	DAL

E0	د	TAL
E1	ر	RA
E2	ز	ZAIN
E3	س	SEEM
E4	س	SEEN
E5	ش	SHEEN
E6	ص	SAD
E7	ض	DAD
E8	ط	TAH
E9	ظ	DHAH
EA	ع	AIN
EB	غ	GHAIN
EC	ف	FA
ED	ق	QAF
EE	ك	CAF
EF	ك	GAF
F0	ل	LAM
F1	لا	LAMALEF
F2	لا	WASLA ON LAMALEF
F3	لا	HAMZAH ON LAMALEF
F4	لا	HAMZAH UNDER LAMALEF
F5	لا	MADDAH ON LAMALEF
F6	م	MEEM
F7	ن	NOON
F8	ه	HA
F9	و	WAW
FA	ؤ	HAMZAH ON WAW
FB	ى	ALEF MAQSURA
FC	ى	YA A
FD	ئ	HAMZAH ON YA A
FE	\	Arabic reverse solidus
FF		Blank "FF" character (imaged as Arabic space)

Key Codes to Reduced Codes Table

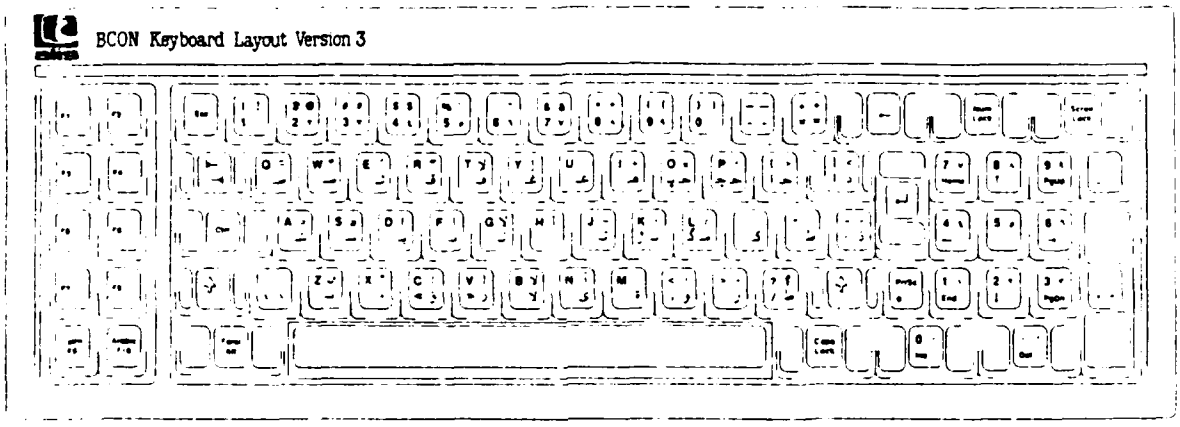
Key code* (ASCII)	English Legend	Reduced Code (ARCII)	Arabic Legend	Arabic Name
20	SPACE	A0	SPACE	Arabic space
21	!	A1	!	Arabic !
22	"	A2	"	Arabic "
23	#	81	#	Arabic #
24	$	A4	$	Arabic $
25	%	A5	٪	Arabic %
26	&	83	&	Arabic &
27	'	E8	ط	TAH
28	(	A8	(	Arabic (
29	)	A9	)	Arabic)
2A	*	AA	*	Arabic *
2B	+	AB	+	Arabic +
2C	,	F9	و	WAW
2D	-	AD	-	Arabic -
2E	.	E2	ز	ZAIN
2F	/	E9	ظ	DHAH
30	0	B0	٠	Arabic 0
31	1	B1	١	Arabic 1
32	2	B2	٢	Arabic 2
33	3	B3	٣	Arabic 3
34	4	B4	٤	Arabic 4
35	5	B5	٥	Arabic 5
36	6	B6	٦	Arabic 6
37	7	B7	٧	Arabic 7
38	8	B8	٨	Arabic 8
39	9	B9	٩	Arabic 9
3A	:	BA	:	Arabic :
3B	;	EE	ك	KAF
3C	<	AE	،	Arabic numeric comma
3D	=	BD	=	Arabic =
3E	>	A6	.	Arabic
3F	?	BF	؟	Arabic ?

(*): Character byte of key code word only (low-order byte). The scan code (high-order byte) is not modified by BCON.

40	@	92	ﷺ	Arabic @
41	A	CC	ـِ	KASRAH
42	B	F5	ﻵ	MADDAH ON LAMALEF
43	C	98	{	Arabic {
44	D	93	[	Arabic [
45	E	C8	ـُ	DAMMAH
46	F	94	]	Arabic]
47	G	F3	ﻷ	HAMZAH ON LAMALEF
48	H	D3	أ	HAMZAH ON ALEF
49	I	A7	÷	Arabic divide sign
4A	J	C1	ـ	KASHIDA
4B	K	AC	،	Arabic comma
4C	L	AF	/	Arabic /
4D	M	84	'	Arabic '
4E	N	D5	آ	MADDAH ON ALEF
4F	O	A3	×	Arabic multiply sign
50	P	BB	؛	Arabic semi-colon
51	Q	C4	ـَ	FATHA
52	R	CA	ـٌ	DAMMATAN
53	S	CE	ـٍ	KASRATAN
54	T	F4	ﻹ	HAMZAH UNDER LAMALEF
55	U	97	`	Arabic `
56	V	9A	}	Arabic }
57	W	C6	ـً	FATHATAN
58	X	C3	ـْ	SUKUN
59	Y	D4	إ	HAMZAH UNDER ALEF
5A	Z	C0	ـ	TAIL
5B	[	DB	ﷻ	JEEM
5C	\	FE	\	Arabic \
5D	]	DF	ـ	DAL
5E	^	95	^	Arabic ^
5F	_	96	_	Arabic _

60		E0	ذ	THAL
61		E5	ش	SHEEN
62		F1	لا	LAMALEF
63		EA	ؤ	HAMZAH ON WAW
64		FC	ى	YA'A
65		DA	ث	THA A
66		D6	ب	BA A
67		F0	ل	LAM
68	h	D1	ا	ALEF
69	i	F8	ه	HA
6A	j	D9	ت	TA'A
6B	k	F7	ن	NOON
6C	l	F6	م	MEEM
6D	m	D8	ة	TA'A MARBUTA
6E	n	FB	ى	ALEF MAQSURA
6F	o	DE	خ	KHA'A
70	p	DD	ح	HA'A
71	q	E7	ض	DAD
72	r	ED	ق	QAF
73	s	E4	س	SEEN
74	t	EC	ف	FA
75	u	EA	ع	AIN
76	v	E1	ر	RA
77	w	E6	ص	SAD
78	x	D0	ء	HAMZAH
79	y	EB	غ	GHAIN
7A	z	FD	ئ	HAMZAH ON YA'A
7B		BE	>	Arabic >
7C	¦	99	:	Arabic |
7D		BC	<	Arabic <
7E	~	C2	ـ	SHADDAH

Key code (scan - char)	English Legend	Reduced Code (ARCII)	Arabic Legend	Arabic Name
1800	A1	D7	ڤ	PEH
1800	A1	D7	ڤ	PEH
1900	A1	DC	ڜ	SHEEM
1900	A1	DC	ڜ	SHEEM
2500	A1	E3	ڛ	SEEM
2500	A1	E3	ڛ	SEEM
2600	A1	EF	ڪ	GAF
2600	A1	EF	ڪ	GAF



Keyboard Layout and Keycodes to Reduced Codes Tables

Display Codes

Display code	Reduced code	Name (shape) (*)
0 to FF	0 to FF	Latin characters, identical to original ASCII set, with the exception of the following two characters
0E	0E	Function code 0E
0F	0F	Function code 0F
100	C2	SHADDAH
101	C3	SUKUN
102	C4	FATHA
103	C5	SHADDAH FATHA
104	C6	FATHATAN
105	C7	SHADDAH FATHATAN
106	C8	DAMMAH
107	C9	SHADDAH DAMMAH
108	CA	DAMMATAN
109	CB	SHADDAH DAMMATAN
10A	CC	KASRAH
10B	CD	SHADDAH KASRAH
10C	CE	KASRATAN
10D	CF	SHADDAH KASRATAN
10E	A0	Arabic visible space
10F	9E	Arabic visible boundary
110	8E	Function code 8E
111	8F	Function code 8f
112	90	Function code 90
113	91	Function code 91
114	9E	Function code 9E
115	00(**)	(Reserved)
116	00	(Reserved)
117	00	(Reserved)
118	00	(Reserved)
119	00	(Reserved)
11A	00	(Reserved)
11B	00	(Reserved)
11C	00	(Reserved)
11D	00	(Reserved)
11E	00	(Reserved)
11F	00	(Reserved)

(*) A means Alone, F means Final, I means Initial and M means Medial

(**) 00 means that this display code is reserved and that no reduced code is associated to it by default

Display code	Reduced code	Name (shape) (*)
120	A0	Arabic space
121	A1	Arabic !
122	A2	Arabic "
123	A3	Arabic =
124	A4	Arabic $
125	A5	Arabic %
126	A6	Arabic &
127	A7	Arabic '
128	A8	Arabic (
129	A9	Arabic)
12A	AA	Arabic *
12B	AB	Arabic +
12C	AE	Arabic , (numeric comma)
12D	AD	Arabic -
12E	A6	Arabic .
12F	AF	Arabic /
130	B0	Arabic 0
131	B1	Arabic 1
132	B2	Arabic 2
133	B3	Arabic 3
134	B4	Arabic 4
135	B5	Arabic 5
136	B6	Arabic 6
137	B7	Arabic 7
138	B8	Arabic 8
139	B9	Arabic 9
13A	BA	Arabic :
13B	BB	Arabic ;
13C	BC	Arabic <
13D	BD	Arabic =
13E	BE	Arabic >
13F	BF	Arabic ?

(*) A means Alone, F means Final, I means Initial and M means Medial.

(**) 00 means that this display code is reserved and that no reduced code is associated to it by default

Display code	Reduced code	Name	(shape) (*)
140	92	Arabic @	
141	D0	HAMZAH	
142	D1	ALEF	AI
143	D2	WASLA ON ALEF	AI
144	D4	HAMZAH UNDER ALEF	AI
145	D7	PEH	A
146	D7	PEH	I
147	D8	TA A MARBUTA	AI
148	D9	TA A	A
149	D9	TA A	I
14A	DA	THA A	A
14B	DA	THA A	I
14C	DB	JEEM	A
14D	DB	JEEM	I
14E	DC	SHEEM	A
14F	DC	SHEEM	I
150	DD	HA A	A
151	DD	HA A	I
152	DE	KHA A	A
153	DE	KHA A	I
154	DF	DAL	AI
155	E1	LAMALEF	A
156	E2	WASLA ON LAMALEF	A
157	E3	HAMZAH ON LAMALEF	A
158	E4	HAMZAH UNDER LAMALEF	A
159	E5	MADDAH ON LAMALEF	A
15A	E6	MEEM	A
15B	93	Arabic [	
15C	FE	Arabic	
15D	94	Arabic]	
15E	95	Arabic	
15F	96	Arabic _	

(*) A means Alone F means Final, I means Initial and M means Medial

(**) 00 means that this display code is reserved and that no reduced code is associated to it by default

Display code	Reduced code	Name	(shape) (*)
160	97	Arabic	
161	F6	MEEM	I
162	F7	NOON	A
163	F7	NOON	I
164	F8	HA	A
165	AC	Arabic text comma	
166	A3	Arabic x (multiply sign)	
167	A7	Arabic divide sign	
168	D3	HAMZAH ON ALEF	AI
169	E0	THAL	AI
16A	00	Arabic > >	
16B	00	Arabic < <	
16C	E4	SEEN with compressed tail	A
16D	E5	SHEEN with compressed tail	A
16E	E6	SAD with compressed tail	A
16F	E7	DAD with compressed tail	A
170	80	Numeric space	
171	82	Numeric x (multiply sign)	
172	85	Numeric %	
173	86	Numeric divide sign	
174	87	Numeric (	
175	88	Numeric)	
176	89	Numeric +	
177	8A	Numeric -	
178	8B	Numeric (	
179	8C	Numeric =	
17A	8D	Numeric)	
17B	98	Arabic	
17C	99	Arabic	
17D	9A	Arabic	
17E	9B	Arabic –	
17F	FF	Arabic (DELETE sign)	

(*) A means Alone, F means Final, I means Initial and M means Medial

(**) 00 means that this display code is reserved and that no reduced code is associated to it by default

Display code	Reduced code	Name	(shape) (*)
180	C2	SHADDAH	(linking)
181	C3	SUKUN	(linking)
182	C4	FATHA	(linking)
183	C5	SHADDAH FATHA	(linking)
184	C6	FATHATAN	(linking)
185	C7	SHADDAH FATHATAN	(linking)
186	C8	DAMMAH	(linking)
187	C9	SHADDAH DAMMAH	(linking)
188	CA	DAMMATAN	(linking)
189	CB	SHADDAH DAMMATAN	(linking)
18A	CC	KASRAH	(linking)
18B	CD	SHADDAH KASRAH	(linking)
18C	CE	KASRATAN	(linking)
18D	CF	SHADDAH KASRATAN	(linking)
18E	C0	TAIL	
18F	C1	KASHIDA	
190	D1	ALEF	F
191	D2	WASLA ON ALEF	MF
192	E4	SEEN with compressed tail	F
193	D3	HAMZAH ON ALEF	MF
194	D4	HAMZAH UNDER ALEF	MF
195	D5	MADDAH ON ALEF	AI
196	D5	MADDAH ON ALEF	MF
197	D6	BA'A	A
198	D6	BA'A	F
199	D6	BA'A	I
19A	D6	BA'A	M
19B	D7	PEH	F
19C	D7	PEH	M
19D	D8	TA'A MARBUTA	MF
19E	D9	TA'A	F
19F	D9	TA'A	M

(*) A means Alone, F means Final, I means Initial and M means Medial.

(**) 00 means that this display code is reserved and that no reduced code is associated to it by default

Display code	Reduced code	Name	(shape) (*)
[illegible]	DA	THA A	I
[illegible]	DA	THA A	M
[illegible]	DB	HEM	F
[illegible]	DB	HEM	M
1A4	DC	SHEEM	I
1A5	DC	SHEEM	M
1A6	DD	HA A	I
1A7	DD	HA A	M
1A8	DE	KHA A	F
1A9	DE	KHA A	M
1AA	DF	DAL	MF
1AB	E5	SHEEN with compressed tail	F
1AC	E0	THAL	MF
1AD	E1	RA	AI
1AE	E1	RA	MF
1AF	E2	ZAIN	AI
1B0	E2	ZAIN	FM
1B1	E3	SEEM	AI
1B2	EE	SEEM	FM
1B3	E4	SEEN	A
1B4	E4	SEEN	F
1B5	E4	SEEN	I
1B6	E4	SEEN	M
1B7	E5	SHEEN	A
1B8	E5	SHEEN	F
1B9	E5	SHEEN	I
1BA	E5	SHEEN	M
1BB	E6	SAD	A
1BC	E6	SAD	F
1BD	E7	DAD	A
1BE	E7	DAD	F
1BF	E8	TAH	AI

(*) A means Alone, F means Final, I means Initial and M means Medial

(**) 00 means that this display code is reserved and that no reduced code is associated to it by default

Display code	Reduced code	Name	(shape) (*)
1C0	E8	TAH	MI
1C1	E9	DHAH	AI
1C2	E9	DHAH	MI
1C3	EA	AIN	A
1C4	EA	AIN	F
1C5	EA	AIN	I
1C6	EA	AIN	M
1C7	EB	GHAIN	A
1C8	EB	GHAIN	F
1C9	EB	GHAIN	I
1CA	EB	GHAIN	M
1CB	EC	FA	A
1CC	EC	FA	F
1CD	EC	FA	I
1CE	EC	FA	M
1CF	ED	QAF	A
1D0	ED	QAF	F
1D1	ED	QAF	I
1D2	ED	QAF	M
1D3	EE	CAF	A
1D4	EE	CAF	F
1D5	EE	CAF	AI
1D6	EE	CAF	MF
1D7	EF	GAF	A
1D8	EF	GAF	F
1D9	EF	GAF	I
1DA	EF	GAF	M
1DB	F0	LAM	A
1DC	F0	LAM	F
1DD	F0	LAM	I
1DE	F0	LAM	M
1DF	F1	LAMALEF	F

(*) A means Alone, F means Final, I means Initial and M means Medial

(**) 00 means that this display code is reserved and that no reduced code is associated to it by default

Display code	Reduced code	Name	(shape) (*)
1E0	F2	WASLA ON LAMALEF	I
1E1	F3	HAMZAH ON LAMALEF	I
1E2	F4	HAMZAH UNDER LAMALEF	I
1E3	F5	MADDAH ON LAMALEF	I
1E4	F6	MEEM	F
1E5	F6	MEEM	M
1E6	F7	NOON	F
1E7	F7	NOON	M
1E8	F8	HA	F
1E9	F8	HA	I
1EA	F8	HA	M
1EB	F9	WAW	A
1EC	F9	WAW	F
1ED	FA	HAMZAH ON WAW	A
1EE	FA	HAMZAH ON WAW	F
1EF	FB	ALEF MAQSURA	AI
1F0	FB	ALEF MAQSURA	MF
1F1	FC	YA'A	A
1F2	FC	YA'A	F
1F3	FC	YA'A	I
1F4	FC	YA'A	M
1F5	FD	HAMZAH ON YA'A	A
1F6	FD	HAMZAH ON YA'A	F
1F7	FD	HAMZAH ON YA'A	I
1F8	FD	HAMZAH ON YA'A	M
1F9	00	ALEF (for LAMALEF)	MF
1FA	00	WASLA ON ALEF (for WASLA ON LAMALEF)	M
1FB	00	HAMZAH ON ALEF (for HAMZAH ON LAMALEF)	MF
1FC	00	HAMZAH UNDER ALEF (for HAMZAH UNDER LAMALEF)	MF
1FD	00	MADDAH ON ALEF (for MADDAH ON LAMALEF)	MF
1FE	E6	SAD with compressed tail	F
1FF	E7	DAD with compressed tail	F

(*) A means Alone, F means Final, I means Initial and M means Medial

(**) 00 means that this display code is reserved and that no reduced code is associated to it by default

APPENDIX E

CODAR I, II, U CODE SETS

Seven bit CODAR II

7 bit CODAR II					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
					⓪	①	②	③	④	⑤	⑥	⑦
0	0	0	0	⓪	NUL	DLE	ESP	U	ⓐ ـًّ	ـً	ب	ظ
0	0	0	1	①	SOH	DC	!	1	ـً	ـً	ت	ع
0	0	1	0	②	STX	DC	"	2	ـٌ	ـٌ	ة	غ
0	0	1	1	③	ET	DC	#	3	ـٍ	ـٍ	ث	ف
0	1	0	0	④	EOT	DC	$	4	ـُ	ـُ	ج	ق
0	1	0	1	⑤	ENO	NAK	%	5	ـّ	ـْ	ح	ك
0	1	1	0	⑥	ACK	SYN	&	6	ـِ	أ	خ	ل
0	1	1	1	⑦	BEL	ETB	'	7	ـَ	إ	د	م
1	0	0	0	⑧	BS	CAN	(	8	ـَ	آ	ذ	ن
1	0	0	1	⑨	HT	EM	)	9	ـَ	ؤ	ر	ه
1	0	1	0	Ⓐ	LF	SUB	*	:	ـَ	ئ	ز	و
1	0	1	1	Ⓑ	VT	ESC	+	;	ـَ	[پ	س	ء
1	1	0	0	Ⓒ	FF	FS	,	<	ـُ	] ة	ش	ا ه
1	1	0	1	Ⓓ	CR	GS	-	=	ـُ	] ك	ص	ى
1	1	1	0	Ⓔ	SO	RS	.	>	ـً	ء	ض	ئ ـ
1	1	1	1	Ⓕ	SI	US	/	?	ـً	ا	ط	ي

CODAR II coding compatible with CCITT Nr. 5. The set coded is the sub-system ASV-CODAR/1 comprising 64 characters for informatics and data transmission. It was presented at the UNESCO/IBI Conference at Bizerte, 1976. The ASV-CODAR/2 sub-system can be obtained by eliminating the characters framed in heavy lines.

Seven bit CODAR U

ش.ع.م CODAR-U COARIM 15-6-1977					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
					0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	ESP	0	◌	ـْ	ب	ظ
0	0	0	1	1	SOH	DC	!	1	ـً	ـٰ	ت	ع
0	0	1	0	2	STX	DC	"	2	ـٌ	ـٱ	ة	غ
0	0	1	1	3	ET	DC	#	3	ـٍ	ـٓ	ث	ف
0	1	0	0	4	EOT	DC	$	4	ـُ	أ	ج	ق
0	1	0	1	5	ENQ	NAK	%	5	ـِ	آ	ح	ك
0	1	1	0	6	ACK	SYN	&	6	ـّ	ؤ	خ	ل
0	1	1	1	7	BEL	ETB	'	7	ـَ	ئ	د	م
1	0	0	0	8	BS	CAN	(	8	ـ	ئ	ذ	ن
1	0	0	1	9	HT	EM	)	9	ـٔ	إ	ر	ه
1	0	1	0	A	LF	SUB	*	:	ـُ	ي	ز	و
1	0	1	1	B	VT	ESC	+	;	ـٗ	پ	س	ء
1	1	0	0	C	FF	FS	,	<	ـٖ	ة	ش	ـ
1	1	0	1	D	CR	GS	-	=	ـٕ	گ	ص	ى
1	1	1	0	E	SO	RS	.	>	ـٙ	ء	ض	ـ
1	1	1	1	F	SI	US	/	?	ـٚ	ا	ط	DEL

APPENDIX F

FINAL CODE U-F.D.

FINAL CODE
CODAR U-F.D.

Recommendation of the final Meeting Held In Rabat (Morocco) In 22-24 April 1982

FOREWORD

The importance of the role of the information channels in the Arabic world is becoming increasingly obvious in all sectors. All Arabic countries are dealing with various types of information in the fields of administration organization, planning, science and technology.

The simple concept of cooperation between the Arab countries, and the positive results of standardization make it necessary to introduce a unified cipher for the Arabic characters used in the field of information exchange.

In this connection the concerned Arabic organization have taken considerable measures such as the two meetings which were held in Rabat (Morocco); the first meeting was the (Arab experts conference for the unified Arabic cipher in the field of information). It was held with the cooperation of the (Arabic Institute for Researches and Arabization) during the period between 25th-29th Sept., 1980. The second meeting concerned with the regulation of the Arabic cipher in its final shape and was held on April 22-24, 1982. In this meeting the technical committee did achieve the projected corrections, and the Arabic cipher which is known as (CODAR U.F.D.) was ready.

Attached are the reasons for modification of the COAR-UF.D., the recommendations adopted at the meetings and the final shape of the unified Arabic cipher which will be formed in an Arabic standard. This standard will be distributed to the ASMO member bodies for further studying and approval as a prelude to the actual experimentation and application.

RECOMMENDATION

In the final session and with a group agreement of the conferees on the final shape of the unified Arabic cipher, the following recommendations have been adopted:

(1) The conference requests the Arab League Education Culture and Science Organization (ALECSO) and Arab Organization for Standards and Metrology (ASMO) to adopt the Arabic cipher which has been agreed upon, and take all necessary measures for its adoption and enforcement in all Arabic countries.

(2) The conferees recommend to the information organization that use Arabic language to experiment the new cipher before enforcement.

These recommendations shall be submitted in particular to the (Institute for Research and Studies for Arabization) in Morocco, the Saudi Arabian Standards Organization and the National Center for Information in Tunisia for the purpose of testing the new cipher before the next (ASMO) meeting.

(3) It is recommended that the Arabic cipher in its new and final share be adopted by the Arabic association for telecommunications.

(4) It is also recommended that ALECSO, the ASMO and the Arab association for telecommunication shall make necessary coordination to use Arabic language in the field of information between them and other international organizations bodies and the UNESCO.

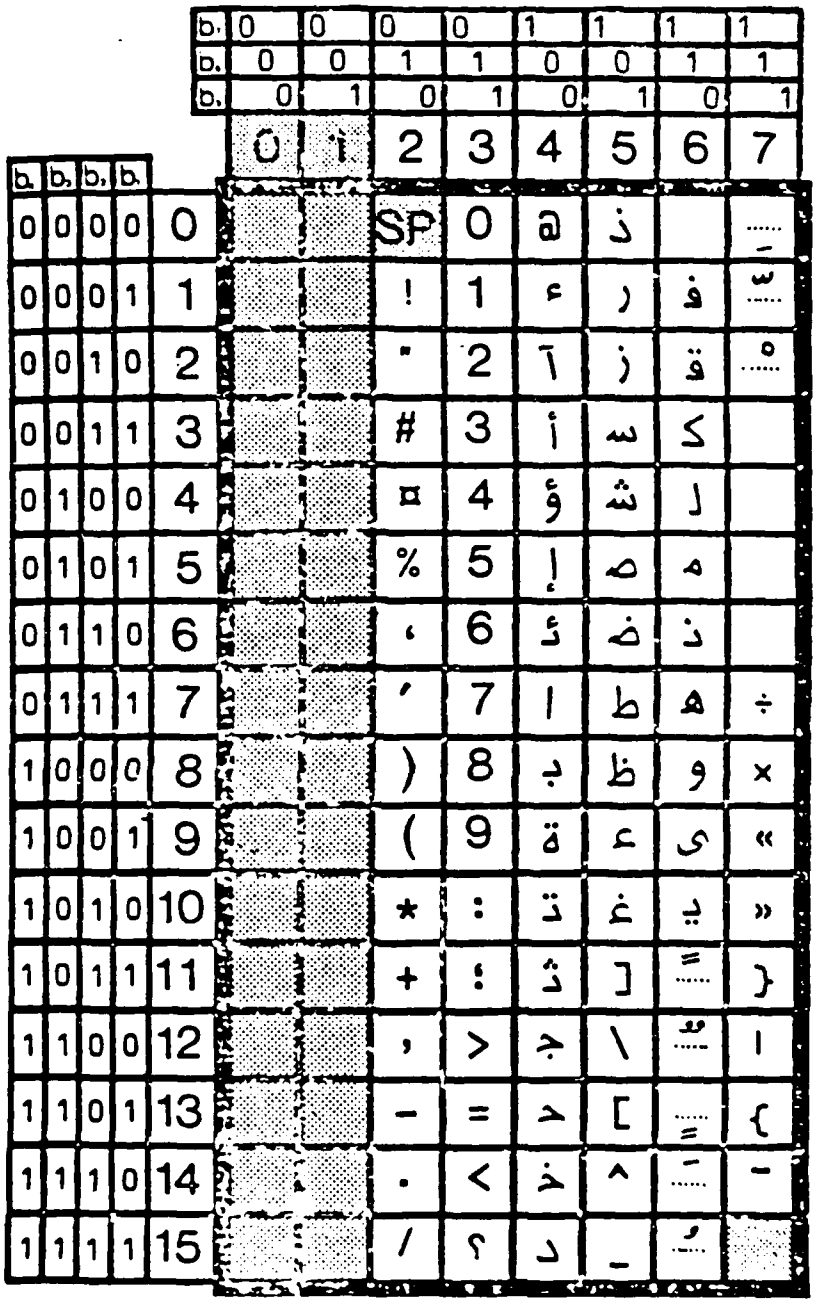
(5) The meeting recommends an emergency session ALECSO and ASMO to regulate the specifications of the devices, the printing letters and their forms and to find the best way of utilizing computers.

(6) The meeting also recommends the continuous contact between ALECSO and ASMO to see to the best execution of these recommendation.

CODAR - U/F D

Codage arabe unifié forme definitive
(RABAT 22 - 24 Avril 1982)

الشفرة العربية الموحدة في صورتها النهائية
(الرباط 22 ـ 24 أبريل 1982)



b.	b.	b.	b.		0	1	2	3	4	5	6	7
0	0	0	0	0			SP	0	@	ذ		ـ
0	0	0	1	1			!	1	ء	ر	ف	ـّ
0	0	1	0	2			"	2	آ	ز	ق	ـٌ
0	0	1	1	3			#	3	أ	س	ك	
0	1	0	0	4			¤	4	ؤ	ش	ل	
0	1	0	1	5			%	5	إ	ص	م	
0	1	1	0	6			،	6	ئ	ض	ن	
0	1	1	1	7			'	7	ا	ط	ه	÷
1	0	0	0	8			)	8	ب	ظ	و	×
1	0	0	1	9			(	9	ة	ع	ى	«
1	0	1	0	10			*	:	ت	غ	ـِ	»
1	0	1	1	11			+	؛	ث	]	ـّ	}
1	1	0	0	12			،	>	ج	\	ـُ	ا
1	1	0	1	13			-	=	ح	[	ـْ	{
1	1	1	0	14			.	<	خ	^	ـً	ـ
1	1	1	1	15			/	؟	د	_	ـٍ	

Réunion Alecso - Asmo

sur la mise au point et la normalisation
du Codar - U.

اجتماع اليكسو ـ آسمو

حول ضبط الشفرة العربية الموحدة
وتنميطها في الاعلاميات

APPENDIX G

ASMO'S APPROVED ARAB STANDARD SPECIFICATIONS

ARAB STANDARD SPECIFICATIONS

449

Data processing - 7 - bit coded Arabic Character set for Information Interchange

ARAB LEAGUE
ARAB ORGANIZATION FOR STANDARDIZATION
AND METROLOGY (ASMO)

Preface

This Arabic Standard was prepared by technical committee No. 8 (Arabic characters in informatics). Among the parties who participated in its preparation are the Arab League Educational, Cultural, and Scientific Organization (ALECSO), and the Institute of Studies and Research for Arabization in Morocco.

In accordance with the 1982 Directives for the Technical Work of the Arab Organization for Standardization and Metrology - Part I: Procedure and Working Methods - this Arabic Standard was adopted by the resolution of the General Assembly of ASMO No:

(R 342 / G.A. / S 15 - October 21, 1982).

DATA PROCESSING: 7-BIT CODED ARABIC CHARACTER SET FOR INFORMATION INTERCHANGE

0. INTRODUCTION

This Arabic Standard specifies the properties of a coded character set using 7-bit binary codes for information interchange among different types of data processing equipments using the Arabic characters. It also specifies a set of control and graphic characters, in addition to its coded representation inspired from ISO 646. The set of specific graphic characters in this standard enable us under all circumstances to represent Arabic text whether it is totally vowelized, partially vowelized, or unvowelized. This standard provides the possibilities for information interchange for special applications, as well as the possibilities for expansion in case of insufficiency of the coded character set. This Arabic Standard was made in accordance with ISO 646, and the following points were modified so that the standard ISO 646 is convenient for Arabic usage:

— Table 1.

— Comments on this table.

Table 1 was modified in such a way which permits the usage of the coded character set as a separate group from the Latin character set described in ISO 646 for information interchange, and the usage of basic programs in Arabic Language for the purpose of complete Arabization when using computers. This table also allows the usage of the coded character set together with the Latin character set as in the *International Standard* ISO 646 because of the correspondence between these two standards.

Applying this standard requires several application standards to be implemented on a carrier (magnetic carrier, transmission network, etc.), and these applications are specified in other standards.

1. SCOPE AND FIELD OF APPLICATION

1.1 This Arabic Standard contains a set of 128 characters (control characters and graphic characters such as letters, digits and symbols) with their coded representation. Most of these characters are mandatory and unchangeable, but provision is made for some flexibility to accommodate special national and other requirements.

1.2 The need for graphics and controls in data processing and in data transmission has been taken into account in determining this character set.

1.3 This Arabic Standard consists of a general table with a number of options, notes, a legend and explanatory notes.

1.4 This character set is primarily intended for the interchange of information among data processing systems and associated equipment, and within message transmission systems.

1.5 This character set is applicable to all Arabic alphabets.

1.6 This character set includes facilities for extension where its 128 characters are insufficient for particular applications.

1.7 The definitions of some control characters in this Arabic Standard assume that data associated with them is to be processed serially in a forward direction. Their effect when included in strings of data which are processed other than serially in a forward direction or included in data formatted for fixed record processing may have undesirable effects or may require additional special treatment to ensure that the control characters have their desired effect.

2. IMPLEMENTATION

2.1 This character set should be regarded as a basic alphabet in abstract sense. Its practical use requires definitions of its implementation in various media. For example, this could include punched tapes, punched cards, magnetic tapes and transmission channels, thus permitting interchange of data to take place either indirectly by means of an intermediate recording in a physical medium, or by local electrical connection of various units (such as input and output devices and computers) or by means of data transmission equipment.

2.2 The implementation of this coded character set in physical media and for transmission, taking into account the need for error checking, is the subject of other ISO publications.

Table (1)

b7	0	0	0	0	1	1	1	1
b6	0	0	1	1	0	0	1	1
b5	0	1	0	1	0	1	0	1
b4 b3 b2 b1 \ column	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	TC. (DLE)	SP	0	﴿	ذ	—	ـَـ
0 0 0 1	1	TC. (SOH)	DC.	!	1	٠	ر	ف	ـُـ
0 0 1 0	2	TC. (STX)	DC.	"	2	٢	ز	ق	ـِـ
0 0 1 1	3	TC. (ETX)	DC.	# (2)	3	١	س	ك	(3)
0 1 0 0	4	TC. (EOT)	DC.	¤ (2)	4	ف	ش	ل	(3)
0 1 0 1	5	TC. (ENQ)	TC. (NAK)	%	5	!	ص	م	(3)
0 1 1 0	6	TC. (ACK)	TC. (SYN)	&	6	ز	ض	ن	(3)
0 1 1 1	7	BEL	TC. (ETB)	'	7	ا	ط	ه	(3)
1 0 0 0	8	FE. (BS)	CAN	) (4)	8	ب	ظ	و	(3)
1 0 0 1	9	FE. (HT)	EM	((4)	9	ة	ع	ى	(3)
1 0 1 0	10	FE. (1) (LF)	SUB	*	:	ت	غ	ي	(3)
1 0 1 1	11	FE. (1) (VT)	ESC	+	; (4)	ث	] (4)	ـً	} (4)
1 1 0 0	12	FE. (1) (FF)	IS. (FS)	, (4)	> (4)	ج	\	ـٌ	|
1 1 0 1	13	FE. (1) (CR)	IS. (GS)	-	=	ح	[(4)	ـٍ	{ (4)
1 1 1 0	14	SO	IS. (RS)	.	< (4)	خ	^	ـَ	~
1 1 1 1	15	SI	IS. (US)	/	؟ (4)	د	_	ـُ	DEL

(1) See Note (1)
(2) See Note (2)
(3) See Note (3)
(4) See Note (4)

NOTES ABOUT TABLE 1:

1) The format effectors are intended for equipment in which horizontal and vertical movements are effected separately. If equipment requires the action of CARRIAGE RETURN to be combined with a vertical movement, the format effector for that vertical movement may be used to effect the combined movement. For example, if NEW LINE (symbol NL, equivalent to CR+LF) is required, FE2 shall be used to represent it. This substitution requires agreement between the sender and the recipient of the data.

   The use of these combined functions may be restricted for international transmission on general switched telecommunication networks (telegraph and telephone networks).

2) The symbols # and locations 2/3 and 2/4 are used respectively to denote NUMBER SIGN and CURRENCY SIGN. Note that the character do not designate the currency of a specific country unless otherwise agreed upon between the sender and the recipient of data.

3) These positions are intended for national use or for alphabet extension. If not used for such purposes, they may be used for representing symbols which do not have specific functions. This requires agreement between the sender and the recipient of the data.

   For the general case of information interchange among computers, these positions shall not be used.

4) Positions and names of special signs which have specific functions in the code table is the same as in ISO 646. However, such signs should be imaged and printed according to text as shown in the following Table.

APPENDIX H

PROGRAM CODE

```
Program Lexical_Translator(input,output);
(* ************************************************ *)
{
File Name     : Lexical.pas
Module name   : Lexical_Translator
Author        : Sadek Saleh AL-Juhaiman
Date created  : April 4, 1986
Last change   : Aug 4, 1986
Calls
  Open_File    = Gets the source file name, and
                 initialize the Output files.
  Initialize   = To initialize the hash table and global
                 variables.
  Fill_Buffer  = Fill the line buffer and increment the
                 line no.
  Buffer_Empty= Check if the line buffer was consumed.
  Token_And_Type    = Get the next token and its type.
  Map_Iden_To_Latin= Search for the identifier in the
                     symbol table. If not predefined
                     then insert it .
  Latin_Integer = Map integer tokens to Latin integers.
  Special_Character= Map special characters to Latin
                     equivalent character.
  Control_Char      = Notifies the presence of escape
                     codes.

Called by      : None
Include files  : Resource.pas

Variables      :
  Line      =Input line buffer.
  Next_Loc  =Points at the first char of next token.
  Token     =Buffer of 255 character.
  Tok_Type  =Types of the token present in token buffer.
  Tok_Len   =The length of the token in token buffer.
  Line_No   =Source code line number.
  Debug_On  =Boolean variable, debugging feature, set
             by Arabic directive in the source code.
  Comment_On= Directive, to include the comments in
               the generated output.
  Res_Word   = Array of records for the reserved words.
               contains the Arabic and its English match.
  Match_Ind  = Index in Res_Word array to token location.
  Int_Str    = Integer string of size 10 characters.
  Line       = Input line buffer.
  Next_Loc   = The first character of the next token in
               the line buffer
  Token      = Token buffer.
  Latin_Id   = The mapped identifier ( in Latin form ).
  Hash       = HashTable;
  ArabicSpell = Spelling string array of 5000 chars.
```

```
  characters   = Number of chars in spelling table.
  Line_No      = Counts the read source lines.
  Line_Size    = Line buffer upper limit.
  Match_Ind    = Index of reserved word found in the
                 constant array.
 Iden_No       = The number of the identifier in the
                 sequence of arrival.
 Latin_Char    =  One character buffer for special
                 characters.
 Lat_Int       =  The integer translation to Latin.
 Error_Set     =  Token error set.                    *)
{
Comment :
The program will ask for input source file with or
without extension . IF the name is valid it will open
the file and initialize tow out put files. The two
files will have same file name and the extensions DIC
and PAS. The DIC file has all userdefined identifiers
with their assigned Id_Numbers. The PAS file will have
the generated PASCAL code.
    After initialization the program will take one
line  and break it to tokens. The token is given a
type, then based on the type a translation module will
be called.
   The above will continue for each line of code until
a major error is encountered. Major error will result
from long tokens when using comments or literal string.
                                                      }
(* ************************************************  *)
```

```
CONST

Max_Arb_Word =12;              { size of Arabic word    }
Max_Lat_Word =12;              { size of Latin  word    }
Max_Len      =255;             { line & literal size    }
Res_Words    =59;              { reserved words size    }
MaxKey  = 6310;                { Prime number, hashing  }
MaxChar = 5000;                { Size of spelling table}

TYPE

Line_Range    = 0..Max_Len;
Arab_Word_Str = string[Max_Arb_Word ];
                              { max char per Latin word }

Latn_Word_Str = string[ Max_Lat_Word];

Word_Rec =RECORD               { constant array record     }
                               { of reserved words         }
     English: Latn_Word_Str;
     Arabic : Arab_Word_Str;
   END;

Reserved_Index= 1 .. Res_Words;
Words        = array  [ Reserved_Index ] OF Word_Rec;
Latin_Token = string [6];{ string in the form id_000  }
WordPointer = ^WordRecord;{ Pointer to user defined id}
WordRecord  =  RECORD      { for user defined iden.    }
                  Index,   { identifier number sequence}
                  Lenth,   { Length of the word .      }

                           { Location of the word last-}
                           { character in symbol table.}
                  LastChar: integer;
                           { link pointer to next word }
                  NextWord: WordPointer;
                           { assigned identifier number}
                  Latin_Id: Latin_Token;
               END;
  HashTable     = array [1 .. MaxKey ] OF WordPointer;
  SpellingTable = array [1 .. Maxchar] OF char;
  Ln_Str        = string[Max_Len];
  Token_Str     = string [Max_Len] ;
  Errors        = ( Long_Token,Long_Comment,
                    Long_Literal_Str, Illegal_Char);
  Types_Of_Token= ( Blanks,Illegal,Reserved_Word,
                    Literal_Str, Contrl_Cod,Unclsfd,
                    Identifier,Coment,Integer1,
                    Funct_Operator );

                         { Arabic characters range     }
```

```
                              { from  80 Hex to  FF Hex     }
Arbic_Alph      = set of  $80 .. $FF ;
Str10           = string[10];
```

```
CONST                          { resource file contains the }
 res_word:words =
      ( (english:'absolute'        ;arabic:'مطلق'),
        (english:'and'             ;arabic:'إضافة_إلى'),
        (english:'array'           ;arabic:'صف'),
        (english:'begin'           ;arabic:'بداية'),
        (english:'case'            ;arabic:'حالة'),
        (english:'const'           ;arabic:'ثابت'),
        (english:'div'             ;arabic:'قسمة'),
        (english:'do'              ;arabic:'إفعل'),
        (english:'downto'          ;arabic:'أسفل_إلى'),
        (english:'else'            ;arabic:'وإلا'),
        (english:'end'             ;arabic:'نهاية'),
        (english:'external'        ;arabic:'خارجي'),
        (english:'text'            ;arabic:'ملف'),
        (english:'forward'         ;arabic:'لاحق'),
        (english:'for'             ;arabic:'لأجل'),
        (english:'function'        ;arabic:'وظيفة'),
        (english:'goto'            ;arabic:'إذهب_إلى'),
        (english:'concat'          ;arabic:'وصل'),
        (english:'inline'          ;arabic:'بالسطر'),
        (english:'if'              ;arabic:'إذا'),
        (english:'in'              ;arabic:'بداخل'),
        (english:'label'           ;arabic:'رقعة'),
        (english:'mod'             ;arabic:'باقي'),
        (english:'nil'             ;arabic:'لاشئ'),
        (english:'not'             ;arabic:'خطأ'),
        (english:'overlay'         ;arabic:'غطاء'),
        (english:'of'              ;arabic:'آل'),
        (english:'or'              ;arabic:'أو'),
        (english:'packed'          ;arabic:'مضغوط'),
        (english:'procedure'       ;arabic:'طريقة'),
        (english:'program'         ;arabic:'برنامج'),
        (english:'record'          ;arabic:'كرت'),
        (english:'repeat'          ;arabic:'أعد'),
        (english:'set'             ;arabic:'مجموعة'),
        (english:'begin'           ;arabic:'بداية'),
        (english:'shl'             ;arabic:'ح.يسار'),
        (english:'real'            ;arabic:'عشري'),
        (english:'integer'         ;arabic:'صحيح'),
        (english:'boolean'         ;arabic:'منطقي'),
        (english:'read'            ;arabic:'إقرأ'),
        (english:'readln'          ;arabic:'إقرأسطر'),
        (english:'write'           ;arabic:'إكتب'),
        (english:'writeln'         ;arabic:'إكتب_سطر'),
        (english:'end'             ;arabic:'نهاية'),
        (english:'shr'             ;arabic:'ح_يمين'),
        (english:'string'          ;arabic:'سلسلة'),
        (english:'then'            ;arabic:'عندئذ'),
        (english:'type'            ;arabic:'طراز'),
        (english:'to'              ;arabic:'إلى'),
```

```
      (english:'until'          ;arabic:'حتى'),
      (english:'var'            ;arabic:'متغير'),
      (english:'str'            ;arabic:'ح_رقم'),
      (english:'chr'            ;arabic:'ق_حرف'),
      (english:'ord'            ;arabic:'ر_حرف'),
      (english:'while'          ;arabic:'بينما'),
      (english:'input'          ;arabic:'دخول'),
      (english:'output'          ;arabic:'خروج'),
      (english:'with'           ;arabic:'مع'),
      (english:'xor'            ;arabic:'أو_حث'));

Arabic_Alph : Arbic_Alph =
  [ $B0 .. $B9,          { Arabic digit                }
    $D0 .. $FD,          { Arabic letters              }
    $96,                 { under score                 }
    $C0         ];       { tail genration              }

Delimiters : SET OF char = {const set,  delimiters }
[ #$80,                  { Space                       }
  #$8E,                  { BCON function code          }
  #$8F,                  { BCON function code          }
  #$90,                  { BCON function code          }
  #$91,                  { BCON function code          }
  #$93,                  { Array left square bracket   }
  #$20,                  { Latin space                 }
  #$94,                  { Array right square bracket  }
  #$95,                  { Arabic up arrow "pointer"   }
  #$97,                  { Arabic reverse apostrophe   }
  #$A0,                  { Arabic Space                }
  #$A3,                  { Arabic multiply             }
  #$A6,                  { Arabic period               }
  #$A7,                  { Arabic divide               }
  #$A8,                  { Arabic left parenthesis     }
  #$A9,                  { Arabic right parenthesis    }
  #$AB,                  { Arabic plus sign            }
  #$AC,                  { ARABIC comma                }
  #$AD,                  { Arabic minus                }
  #$AE,                  { numeric comma used as       }
                         { the Latin decimal dgt       }
  #$BA,                  { Arabic colon                }
  #$BC,                  { Arabic greater than         }
  #$BD,                  { Arabic equal   sign         }
  #$BE ];                { Arabic less    than         }
```

```
VAR
  Debug_On     : boolean;
  Comment_On   : boolean;
  Tok_Type     : Types_Of_Token;
  Tok_Len      : Line_Range;
  Int_Str      : string[10];
  I            : integer;
  Line         : In_Str;
  Next_Loc     : Line_Range;
  token        : token_Str;
  Latin_Id     : Latin_Token;
  Hash         : HashTable;
  ArabicSpell  : SpellingTable;
  Characters   : integer ;
  Line_No      : integer;
  Line_Size    : Line_Range;
  Iden_No      : 000 .. 999;
  Match_Ind    : Reserved_Index;
  Latin_Char   : char;
  Lat_Int      : str10;
  Error_Set    : SET OF errors;
  OutFile      : text;
  InFile       : text;
  Dictionary   : text;

Procedure
OPEN_FILE;
VAR
valid  :boolean;          { for I/O error W/ file name }
F_Name,                   { file name with no extension}
File_Name : string[12];{ file name from key board.}
ind       : integer;
BEGIN
  valid := false;
  WRITELN ('Input_File name:');
  REPEAT                      { until valid file name  }
    READln  (File_Name);
    ASSIGN (InFile,File_Name);
    {$I-}                     { if no error opening  file}
    RESET(Infile);            { then file exist          }
    {$I+}                     { if no I/O error, its valid}
    valid := ( IOresult = 0);
    ClrScr;
    if not (valid) THEN
       BEGIN
         WRITELN(' ** FAILURE TO OPEN FILE === ',
                   File_Name );
         WRITELN('    Please RE_ENTER Input_File name ');
       END;
  UNTIL VALID;

  ind:= 1;
```

```
  REPEAT                    { get the name W/O extension }
    F_Name(.ind.) := File_Name(.ind.);
    ind :=ind + 1;
  UNTIL (File_Name(.ind.)=' ') OR
        (File_Name(.ind.)='.') OR
        ( ind > LENGTH (File_Name) );
  F_Name(.0.) := CHR(ind-1);
  ASSIGN (outfile,F_name+'.pas'); { translator output }
  ASSIGN (dictionary,F_Name+'.dic');{ dictionary file }
  RESET  (infile);
  REWRITE(outfile);
  REWRITE(dictionary);    { file contains  identifiers }
END;                        { and their translations     }
```

```
Procedure
INITIALIZE;                    { Initialize the hash keys    }
VAR                            { and the global variables    }
   KeyNo : integer;
BEGIN
  Debug_On  := false;
  Comment_On:= false;
  Error_Set:=[];
  Line_No := 0;
  Iden_No   := 0 ;
  KeyNo     := 1 ;
  WHILE KeyNo <= MaxKey DO
    BEGIN
      hash(. KeyNo .) := nil;
      KeyNo := KeyNo + 1 ;
    END;
  characters := 0 ;        { count of chars in spell tbl }
END; { initializ }

PROCEDURE
FILL_BUFFER
(  VAR line     : ln_Str;     { input line buffer         }
   VAR where    : line_range;{ location in buffer        }
   VAR line_no : integer;
   VAR Ln_Size : line_range
                              );
BEGIN
  READLN(infile,line);
  Line_No := Line_No + 1;
  IF Debug_On THEN WRITELN(line);
  IF (line= '{+Comment}' )THEN { set comment directive   }
  BEGIN
    Comment_On:= true;
    READLN(infile,line);
    line_No := Line_No +1 ;
  END;

  IF line ='{-Comment}'  THEN
  BEGIN                      { reset comment directive    }
    Comment_On:= false;
    READLN(infile,line);
    line_No := Line_No +1 ;
  END;
  IF line ='{+Debug}'  THEN  { set debug directive        }
  BEGIN
    Debug_On  := true ;
    READLN(infile,line);
    line_No := Line_No +1 ;
  END;

  IF line ='{-Debug}'  THEN  { reset debug directive      }
  BEGIN
```

```
    Debug_On  := false;
    READLN(infile,line);
    line_No := Line_No +1 ;
  END;
  where   := 1 ;                { initialize line pointer }
  Ln_Size := length(line);      { line size               }
END;
```

```
FUNCTION
BUFFER_EMPTY
( Next_Loc : line_range;
  Ln_Size  : line_range
                          ):BOOLEAN;

BEGIN                       { check if buffer is empty    }
  BUFFER_EMPTY := ( next_loc > Ln_Size);
END;

FUNCTION
EMPTY_ERROR_SET: BOOLEAN;
{ ********************************************** }
{ If error set is empty then no errors are found
  yet. translation will continue                  }
{ ********************************************** }

BEGIN
  EMPTY_ERROR_SET := (ERROR_SET = []);
END;

Procedure
TOKEN_AND_TYPE
( VAR where     : line_range;{ location  of next token }
  VAR token     :token_Str;
  VAR Tok_Len   :line_range; { length of resulted token}
  VAR Tok_Type :Types_Of_Token; { Token type          }
  VAR Match_Ind:reserved_index{ index of res. words   }
);

{ ************************************************** }
{
    module name   : TOKEN_AND_TYPE
    date created  : April 7, 1986
    calls         : Blanks, Comments, Literal_String,
                    Integer_Tok, Identifier_Tok,
                    Reserved_Tok, Special_Char,
                    Control_Char

    called by     : MAIN
    variables     :
    last change   : Aug 5, 1986
    Comment       :
    procedure collects the tokens  and assigned
    Token Type  names to them.
{ ************************************************ }

VAR     index      :integer;   { For token indexing      }
        ch         : char;     { special characters token }
CONST   digits     :SET OF char = [##30  .. ##39 ];
```

```
Procedure
BLANK;                                  { collects blank(s) token    }
VAR index:integer;

BEGIN
  index:=0;
                                        { A0 Arabic space 'blanks'   }
                                        { 20 Latin space 'blanks'    }
  WHILE ( ORD( line [where]) = $A0 ) OR
        ( ORD( line [where]) = $20 ) DO
  BEGIN
      index:=index + 1;
      token(. index.) := line(.where.);
      where := where+1;
  END;
  Tok_Type := blanks;
  Tok_Len  := index;
  token (.0.) := CHR(index);
END;
```

```
Procedure
COMMENT;
{ ************************************************** }
{  Procedure comment will assign the matching Latin
brackets and the body of the comment to the token.
The token type then set to Comment.                  }
{ ************************************************** }

BEGIN
  token[1] := '(';           { assign the opening bracket }
  token[2] := '*';           { and asterisk to token      }
  index    :=  2 ;           { start of comment body      }
  where    := where + 2;

  REPEAT                     { assign body of comment     }
    index:= index+1;         { pointer of token buffer    }
    token [index]:=line [where];
    where := where +1;       { pointer of line buffer     }
  UNTIL (  (ORD (line[where]  ) = $AA ) AND
           (ORD (line[where+1]) = $A8 )) OR
           (where  >=  Line_Size  );

  IF (where >= Line_Size) THEN
  BEGIN                      { The end of line is reached }
    Tok_Type := Illegal ;    { before closing the comment }
    Error_Set:= Error_Set + [Long_Comment];
  END
  ELSE                       { the comment is valid       }
    BEGIN
      token[index+1] := '*';{ assign the closing bracket }
      token[index+2] := ')';
      Tok_Type := coment;
      where := where + 2;    { advance line pointer       }
      Tok_Len  := index+2 ; { advance token pointer       }
      token[0] := chr(Tok_Len); { set token length        }
    END;
END; { COMMENT }
```

```
PROCEDURE
LITERAL_STRING;
{ ********************************************************* }
{
   Literal string will look for single and double quotes.
   Matching the quote character at the beginning and the
   end of the string. Then assigning the Latin quotation
   marks.                                                    }
{ ********************************************************* }

BEGIN
  index:= 0;
  CASE ORD(line[where]) of    { if buffer points at ;      }

    $97 : REPEAT              { single quotes              }
            index := index +1;
            token[index] := line[where] ;
            where := where + 1;
          UNTIL (ORD(line[where]) = $97 ) OR
                (where > line_size) ;

    $A2 : REPEAT              { double quotes              }
            index := index +1;
            token[index] := line[where] ;
            where := where + 1;
          UNTIL (ORD(line[where]) = $A2 ) OR
                (where > line_size);
  END;{ CASE }
                              { if literal ended with      }
                              { the right quote mark;      }
  IF (ORD(line(.where.))= $A2) OR
     (ORD(line(.where.))= $97) THEN
  BEGIN
    index     := index + 1; { advance pointer for the }
    Tok_Len   := index;     { quote mark. Set length. }
    Tok_Type  := Literal_Str;
    token[0]  := chr(Tok_Len);

                              { for single quote literal}
    IF (ORD(token [1]) = $97)  THEN
    BEGIN                     { assign single quotes     }
      token [1]     :=chr($27);
      token [index] := chr($27);
    END;

    IF (ORD(line [where])= $A2 )THEN
    BEGIN                     { assign double quotes     }
      token [1]     :=chr($22);
      token [index] := chr($22);
    END;
    where := where + 1 ;      { point to the next token }
  END
```

```
 ELSE                          { if line pointer did not see}
 BEGIN                         { single/double qoute= error }
   Error_Set :=Error_Set + [Long_Literal_Str];
   Tok_Type  := illegal;
   Tok_Len   := index;
   token[0]  := chr(index); { set length of token      }
 END;
END;
```

```
PROCEDURE
INTEGER_TOK;
{ ************************************************** }
{ The procedure will return the Digits ranging
   from B0 .. B9 Hex.
                                                      }
{ ************************************************** }

BEGIN
  index := 0;
  WHILE ( line(. where.) in digits ) DO
    BEGIN;
      index := index + 1 ;
      token(.index.) := line(.where.);
      where := where +1;
    END;
  Tok_Type := integer1;
  Tok_Len  := index;
  token[0] := chr(index);
END;

Procedure
IDENTIFIER_TOK;

{ ********************************************************** }
{ The procedure will look for any number of digits and
   underscore characters following the first letter.
                                                            }
{ ********************************************************** }

VAR valid: boolean;
BEGIN
  index:= 0;
  REPEAT
    index:= index + 1;
    token(.index.) := line(.where.);
    where:= where+1;
  UNTIL not(  ORD(line(.where.)) in Arabic_alph );
  Tok_Type:= Identifier;
  Tok_Len := index;
  token[0]:= chr(index);
END;
```

```
Procedure
RESERVED_TOK
(     VAR match_index: reserved_index);

{ ************************************************** }
{  If the TOKEN is reserved word. The procedure will
   set the  token type to Reserved_Tok and pass the
   index of the word.  In the constant array.
                                                      }
{ ************************************************** }

VAR    index: integer;
       hit  : boolean;          { when a match is found   }

BEGIN
  hit   := false;
  index := 1;
  WHILE (index <= res_words ) AND ( not(hit)) DO
  BEGIN
    IF( token = res_word(.index.).Arabic)  THEN
    BEGIN                      { the token match with    }
      hit := true;             { reserved word           }
      match_index :=index;
    END;
    index:= index + 1 ;
  END; { while no hit }
  IF hit  THEN                 {if token is reserved word}
     Tok_Type := Reserved_word; { set the token type   }
END;
```

```
Procedure
SPECIAL_CHAR_TOK;
{ ******************************************************}
{ The procedure gets all the the tokens of one char
   other than the escape codes.                         }
{ ****************************************************** }
var Illegal_Chars :set of $21..$FF;

BEGIN
  Illegal_Chars:= [$21 ..$7E,{ Latin  chars              }
                   $81..$8D,{ numeric characters, Arabic }
                   $92,     { Arabic @ character         }
                   $97,$99,
                   $9B..$9F,{ non used characters        }
                   $A1..$A2,
                   $A4..$A5,
                   $AA,$AF,
                   $BF,
                   $C0..$CF ];  { Arabic diacritics      }
  IF  ord(line (.where.)) in Illegal_chars  THEN
  BEGIN                        { Latin characters        }
    Tok_Type:= illegal;
    error_set:=error_set + [Illegal_Char];
  END
  ELSE
  BEGIN
    token[1]:=line [where]; { one character special char }
    where     := where + 1; { advance line pointer       }
    Tok_Len   := 1;
    token(.0.):= chr(1);    { set token length to one    }
    Tok_Type  := Funct_Operator { set tokne type          }
  END;
END;

Procedure
CONTROL_CHARS;
{ ********************************************************** }
{ control characters are used by BCON and will be omitted.
                                                             }
{ ********************************************************** }
```

```
BEGIN
  token[1] := line[where];
  Tok_Type := contrl_cod;
  Tok_Len  := 1;
  if Debug_On THEN
  BEGIN
    WRITELN(' Control character (',ORD(line[where]),
            ') in source code');
    WRITELN(' IN Line Number ', Line_No,
            ', Location =  ',where );
  END;
  where := where + 1;
END;

BEGIN;                          { TOKEN_AND_TYPE          *)

{ Based on the first character of the token call an
   appropriate module to collect the token and set the type.}


Tok_Type := unclsfd ;           { initialize token type  }
IF(ORD(line[where]) = $A9)AND   { $A9 openings bracket   }
  (ORD(line[where+1])=$AA)THEN { $AA is asterisk         }
       COMMENT;                 { call procedure Comment }

  IF Tok_Type <> coment THEN{ if not comment THEN based  }
  CASE  ORD(line[where]) OF { on first char get the type }

    $A0,$20  : BLANK;        { leading space(s)           }
    $A2,$97  : LITERAL_STRING;
    $B0..$B9 : INTEGER_TOK; { get integer token           }
    $D0..$FD : BEGIN         { leading letter             }
                IDENTIFIER_TOK; { is it user defined/
                                  reserved}
                RESERVED_TOK(match_ind);
               END;
    $80,$8E,
    $8F,$90,
    $91      : CONTROL_CHARS;    { control characters      }
    ELSE       SPECIAL_CHAR_TOK;

 END; { case }
END;
```

```
Procedure
MAP_IDEN_TO_LATIN
(     token     : Token_Str;
      lenth     : integer;
 VAR Latin_Id : Latin_Token );


{ ************************************************** }
{
   module name   : Map_Iden_To_Latin
   date created  : April 30,1986
   calls         : SEARCH
   called by     : MAIN
   variables     :
      token     = scanned identifier token.
      lenth     = length of scanned identifier
      Latin_Id  = the translated identifier in Latin form

  last change   : Aug 2, 1986

  Comment
     The Procedure will look up an Arabic identifier if not
     in the list it will insert the Arabic token in the list.
     The token will be assigned a Latin label for the use of
     the PASCAL compiler. The meaningless label will have the
     form of Id_### . Where the '#' is an integer.

  Note: code segments of this module is taken from
        ------------------------------------------
        "PRINCH HANSEN ON PASCAL COMPILERS" 1985
         see thesis references
                                                        }
{ ************************************************** }
```

```
Function Hash_Key              { return  the hash key of }
(         token:token_Str;      { the identifiers.       }
          lenth:line_range
):integer;

CONST   W = 32513;           { 32768 - 255, overflow chek }
        N = MaxKey;          { Prime number for words size}
VAR     sum,i :integer;      { sum is the token ord. value}

BEGIN
  sum := 0;
  i   := 1;
  WHILE  i <= lenth DO
  BEGIN
    sum := (sum +ORD(token(.I.) )) MOD W;
    i:= i + 1;
 END;
 Hash_Key:= (sum MOD N ) + 1;
END;


Procedure INSERT
(    token :token_Str;
     lenth:line_range;
     index :integer;
     KeyNo :integer

);

VAR     m,n      : integer;
        pointer  : wordpointer;
        temp     : Latin_token;

    PROCEDURE
    ID_NO( VAR Latin_id : Latin_token);
      VAR
      TEMP : string [3];
    BEGIN
      CASE IDEN_NO OF
        0..9     : BEGIN
                     STR(Iden_no:1,TEMP);
                     Latin_id := CONCAT('id_', TEMP);
                   END;
        10..99   : BEGIN
                     STR(Iden_No:2,TEMP);
                     Latin_id := CONCAT('id_',TEMP);
                   END;
        100..999 : BEGIN
                     STR(Iden_No:3,TEMP);

       Latin_id:= CONCAT('id_',TEMP);
                   END;
```

```
BEGIN                              { insert Identifier in    }
                                   { spelling table          }
  characters := characters + lenth;
  m := lenth;
  n := characters - m;
  WHILE  (m > 0) DO
  BEGIN
    ArabicSpell(. m+n .):= token(.m.);
    m:= m - 1;
  END;
  ID_NO( temp);
  NEW(pointer);                    { Insert word record info}
   pointer^.Latin_Id := temp;
   pointer^.NextWord  := Hash(.KeyNo.);
   pointer^.Index     := index;
   pointer^.lenth     := lenth;
   pointer^.lastchar  := characters ;
   WRITELN(dictionary,'                    ',
           pointer^.Latin_Id,'          ',token);
   Hash(.KeyNo.)      := pointer;

END;


FUNCTION
FOUND
( token   : token_Str;
  lenth   : integer;
  pointer: WordPointer
      ): boolean;

VAR    same : boolean;
       m,n  : integer;

BEGIN
  IF Pointer^.lenth <> lenth THEN
     same := false
  ELSE
  BEGIN
    same := true;
    m    := lenth;
    n    := pointer^.lastchar - m;
    WHILE same AND (m > 0 ) DO
    BEGIN
      same := token(.m.) = ArabicSpell(.m+n.);
      m    := m - 1 ;
    END;
  END;
  FOUND := same;
END;
```

```
Procedure
Search
(       token      : token_Str;
        lenth      : integer; { token length            }
  VAR   Latin_Id : Latin_token    { returned Latin token}
);

{  ************************************************** }
{  Comment:
    The module will call function Hash_Key to get the
token key and then look the key up in a hash table.
The  hash  table content is pointers, pointing at
word records. The records has the length of token ,
location  in symbol table, Latin  Identifier number,
the next word in the linked list.
    IF the pointer resulted from the Key number is nil,
that means the word is not in the table. That means the
word must be inserted if there is room in the spelling
table. Insertion is made by procedure INSERT. If the
pointer is pointing at a record, or linked list of
records, function  .
    FOUND is called to verify the spelling.
                                                       }
{ ************************************************** }


VAR   KeyNo     : integer; { global variables for SEARCH}
      done      : boolean;
      Pointer   : wordpointer;



BEGIN                              {            SEARCH            }
  KeyNo    := Hash_Key(token,lenth);
  pointer  := hash(.KeyNo.);
  done     := false;
  WHILE not(done) DO
                                   { insert new id. if size and }
    IF ( pointer = nil ) THEN{ and number within limits  }
    BEGIN                  {          add identifier         }
         Iden_No := Iden_No + 1 ;
         INSERT (token,lenth,Iden_No,KeyNo);
         Latin_Id := hash(.keyNo.)^.Latin_Id;
         done := true;
   END

   ELSE IF FOUND(token,Tok_Len,pointer) THEN
      BEGIN
        Latin_Id := pointer^.Latin_Id;
        done      :=true;
      END
```

```
        ELSE
          pointer := pointer^.nextword
    END;
```

```
BEGIN;                          {       Map_Iden_To_Latin       }
SEARCH (Token, Tok_Len, Latin_Id);
END; { MAP-IDENTIFIER-TO-LATIN }


PROCEDURE
GET_LATIN_SPEC_CHAR
            (     token       :token_Str ;
              VAR Latin_char  :char
            );
VAR Arb_char:string[1];
BEGIN
Arb_Char:=token(.1.);
  CASE ORD ( Arb_Char ) OF
    $BC : Latin_char := '>';{ Arabic greater than          }
    $BE : Latin_char := '<';{ Aragbic less than            }
    $93 : Latin_char := ']';{ Arabic square bracket        }
    $94 : Latin_char := '[';{                              }
    $A8 : Latin_char := ')';{ Arabic RIGHT parenthesis     }
    $A9 : Latin_char := '(';{ ====    LEFT     ====        }
    $AB : Latin_char := '+';{ Arabic Plus                  }
    $AD : Latin_char := '-';{         Minus                }
    $A7 : Latin_char := '/';{         Divide               }
    $96 : Latin_char := '_';{         Under_Score          }
    $A3 : Latin_char := '*';{         Multiply             }
    $BA : Latin_char := ':';{         Colon                }
    $BD : Latin_char := '=';{         Equal                }
    $AE : Latin_char := '.';{         Numeric comma        }
    $95 : Latin_char := '^';{         Hat                  }
    $A6 : Latin_char := '.';{         Period               }
    $BB : Latin_char := ';';{         Semicolon            }
    $AC : Latin_char := ',';{         Comma                }
 END;
```

```
END;
Procedure
LATIN_INT
(     token  :token_Str;
      Tok_Len:line_range;
  VAR Lat_Int:Str10       );

VAR ind : integer;
BEGIN                          { for each digit map to      }
                               { Latin digit                }
  for ind:= 1 to Tok_Len DO
    CASE ORD(token(.ind.)) of
      $B0 : Lat_Int(.ind.) := '0';
      $B1 : Lat_Int(.ind.) := '1';
      $B2 : Lat_Int(.ind.) := '2';
      $B3 : Lat_Int(.ind.) := '3';
      $B4 : Lat_Int(.ind.) := '4';
      $B5 : Lat_Int(.ind.) := '5';
      $B6 : Lat_Int(.ind.) := '6';
      $B7 : Lat_Int(.ind.) := '7';
      $B8 : Lat_Int(.ind.) := '8';
      $B9 : Lat_Int(.ind.) := '9';
    END;
  Lat_Int(.0.) := token(.0.);    { set length of token     }

END;

PROCEDURE
PRINT_ERROR_MESSAGES;
var ind : integer;

BEGIN
  WRITELN  ('*** ERROR ON LINE NO. ',line_no);
  for ind := 1 to line_size do write ( line(.ind.) );
  WRITELN;

  IF long_token   IN error_set THEN
     WRITELN(' has long token ***', token);
{  IF long_comment IN error_set THEN
     WRITELN(' has long comment***',token); }

  IF long_literal_Str  in error_set THEN
     WRITELN(' UNCLOSED QUOTES ');
  IF Illegal_Char IN error_set THEN

       WRITELN ('========== Character number  ',Next_Loc .
                '  is out of range=========');

END;
```

```
                                { main }

BEGIN

OPEN_FILE;
INITIALIZE;
While not(eof(infile)) AND( error_set = []) DO
BEGIN                            {  Line process                  }
  FILL_BUFFER(line,next_loc,line_no,line_size);
  WHILE not ( BUFFER_EMPTY ( next_loc,line_size) ) AND
                                ( error_set = []) DO
  BEGIN                          { Token process                  }
    TOKEN_AND_TYPE(next_loc,token,Tok_Len,
                   Tok_Type,Match_Ind);
    IF Debug_On THEN
       WRITELN('token = ',token,'lenght= ',
                Tok_Len,'  Next_Loc = ',Next_Loc);

    CASE Tok_Type of
     blanks         : FOR i := 1 to Tok_Len DO
                       write(outfile,' ');
     coment         : IF Comment_On THEN
                         write(outfile,token);
     literal_Str    : write(outfile,token);
     reserved_word: write(OUTFILE,
                         res_word(.match_ind.).English);
     identifier     : IF (Iden_No < 1000) AND
                         (characters < MaxChar ) THEN
                      BEGIN
                        MAP_IDEN_TO_LATIN(token,Tok_Len,Latin_Id);
                        write(outfile,Latin_Id);
                      END;
     integer1       : BEGIN
                        LATIN_INT(token,Tok_Len,Lat_Int);
                        write(outfile,Lat_Int);
                      END;
     funct_operator: BEGIN
                      GET_LATIN_SPEC_CHAR (token,Latin_char);
                      write(outfile,Latin_char);
                      END;
     contrl_cod     : WRITELN('line ',line_no:4 ,
                              ' Control code was ignored ') ;
     illegal        : BEGIN
                        PRINT_ERROR_MESSAGES ;
                      END;
    END;  { CASE          }
  END;    { WHILE TOKEN }
  WRITELN(outfile);
END;
IF error_set <> [] THEN WRITELN('error on token type');
CLOSE( outfile);
```

```
CLOSE( infile);
CLOSE( dictionary );
END.
```

APPENDIX I

TEST RUNS

Test Run 1

```
برنامج اسماء_الفصل_٢!؛
ثابت   عدد_الطلبه  = ٣٢ ؛
طراز الطالب = كرت
        الاسم  : سلسله [٣٠] ؛
        العمر  : صحيح ؛
        الجنس  : منطقي ؛
نهايه ؛
متغير   طلبه : صف [ ١.. عدد_الطلبه] ال الطالب ؛
        مؤشر : صحيح؛
بدايه
بينما ( مؤشر < ٣٢) !فعل
بدايه
مؤشر :=  مؤشر + ١ ؛
!قرأ ( طلبه[مؤشر].الاسم،طلبه[مؤشر].العمر)؛
!كتب ( طلبه[مؤشر].الاسم،طلبه[مؤشر].العمر)؛
نهايه؛
نهايه.
```

Source Code

id_1	اسماء_الفصل_٢
id_2	عدد_الطلبه
id_3	الطالب
id_4	الاسم
id_5	العمر
id_6	الجنس
id_7	طلبه
id_8	مؤشر

Test Run 1 Dictionary Table

```
program id_1;
    const   id_2  = 32 ;
    type id_3 = record
                  id_4  : string [30] ;
                  id_5 :  integer ;
                id_6 : boolean ;
              end ;
var   id_7 : array [ 1.. id_2] of id_3 ;
        id_8 : integer;
begin
  while( id_8 < 32) do
  begin
    id_8 := id_8 + 1 ;
    read ( id_7[id_8].id_4,id_7[id_8].id_5);
    write ( id_7[id_8].id_4,id_7[id_8].id_5);
  end;
end.
```

Generated Code

Test Run 2

```
برنامج اسماء_الفصل_١٢؛
ثابت   عدد_الطلبه  = ٣٢ ؛
طراز الطالب = كرت
        الاسم  : سلسله [٣٠] ؛
        العمر  : صحيح ؛
        الجنس  : منطقي ؛
نهايه ؛
متغير   طلبه : صف [ ١.. عدد_الطلبه] من الطالب ؛
        مؤشر : صحيح؛
بدايه
 بينما ( مؤشر < ٣٢) افعل
  بدايه
  مؤشر :=  مؤشر + ١ ؛
  اقرأ ( طلبه[مؤشر].الاسم،طلبه[مؤشر].العمر)؛
  اكتب ( طلبه[مؤشر].الاسم،طلبه[مؤشر].العمر)؛
  نهايه؛
 نهايه.
```

Source Code

id_1	اسماء_الفصل_١٢
id_2	عدد_الطلبه
id_3	الطالب
id_4	الاسم
id_5	العمر
id_6	الجنس
id_7	طلبه
id_8	مؤشر

Dictionary Table

Test Run 2

```
program id_1;
    const   id_2  = 32 ;
    type id_3 = record
                  id_4  : string [30] ;
                  id_5 :  integer ;
                id_6 : boolean ;
              end ;
var   id_7 : array [ 1.. id_2] of id_3 ;
        id_8 : integer;
begin
  while( id_8 < 32) do
  begin
    id_8 := id_8 + 1 ;
    read ( id_7[id_8].id_4,id_7[id_8].id_5);
    write ( id_7[id_8].id_4,id_7[id_8].id_5);
  end;
end.
```

Generated Code

Test Run 3

```
{تحري+}

برنامج جديد (دخول،خروج)؛
ثابت عدد_الطلبه_٦ = ٥؛
متغير الاسم : سلسله[٥٠]؛
الهاتف: سلسله[١٢]؛
الدخل : عشري؛

بدايه
الدخل:= ١٢٢,٧ x الهاتف ؛
الاسم:= 'صادق صالح عبدالعزيز الجهيمان' ؛
الهاتف:= '٠٥٣٦٢٤٧٦' ؛
وصل ( الاسم،الهاتف)؛
إكتب_سطر (الاسم،الهاتف)؛

نهايه.
```

Source Code

Test Run 3

```
program id_1(input,output);
  const id_2 = 15;
  var id_3  : string[50];
        id_4: string[12];
        id_5 : real;

begin
  id_5:= 122.7 * id_4 ;
   id_3:= 'صادق صالح عبدالعزيز الجهيمان' ;
   id_4:='٤٧٦٢٠٥٣ ' ;
    concat ( id_3,id_4);
   writeln (id_3,id_4);

end.
```

Generated Code

id_1	جديد
id_2	عدد_الطلبه_ل
id_3	الاسم
id_4	الهاتف
id_5	الدخل

Dictionary Table

LIST OF REFERENCES

1. Proceedings of the International Symposium for Standardization of Codes, Character Sets and Keyboards for the Arab Language in Computers, 1-4 June 1980 in Riyadh, Saudi Arabia, Saudi Arabian Standard Organization, 1984.

2. BCON Programmer's Manual, Arabic-Latin Information Systems, Inc., Montreal, Canada, 1985.

3. Hansen, Per B., Brinch Hansen on PASCAL Compilers, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

BIBLIOGRAPHY

Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey, D., Compilers: Principles, Technique and Tools. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

Schlidt, Herbert, Advanced Turbo PASCAL: Programming and Technique. McGraw-Hill Book Company, Berkeley, California, 1986.

Tremblay, Jean-Paul and Sorenson, Paul G., The Theory and Practice of Compile Writing. McGraw-Hill Book Company, New York, New York, 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
2.	Prof. Daniel Davis, Code 52Vv Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
3.	Cdr. Ron Rautenberg, Code 52Rt Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4.	Prof. Kamil Said, Code 56Si Department of National Security Affairs Naval Postgraduate School Monterey, California 93943-5000	1
5.	Major Abdul-Latif Alzayani Department of Operations Research Naval Postgraduate School Monterey, California 93943-5000	1
6.	Major Hamad Al_Yosefi 20907 East Borough Drive Fort Collins, Colorado 80525	1
7.	Major Abdullaziz I. Al-Hudaithi 20907 East Borough Drive Fort Collins, Colorado 80525	1
8.	CPT Abdulkareem Al-Juhaiman 1596 W. Straford Drive Chandler, Arizona 85224	1
9.	Royal Saudi Air Defense Forces Training Riyadh, Saudi Arabia	1
10.	Prof. Ahmed Lakhdar Gazal Director De L'Iera P.O. Box 430 Rabat, Morocco	1

11. Director General of Saudi Arabian Standards Organization
Riyadh, Saudi Arabia — 1

12. CPT Sadek S. Alju-aiman
P.O. Box 5233
Riyadh 11422
Saudi Arabia — 10

13. Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22304-6145 — 2

END

2-87

DTIC