Table of Contents

OVERVIEW

languageONE is the both the newest and the oldest of programming languages. By the strictest of definitions it is assembler code, but you would never know that by looking at it. More broadly, it is a set of macros that provide for all the functionality required when programming. And it is more. It provides a framework to support the macros in an assembled program and systematically defines the structures required to manage data and program flow. *languageONE* provides for 64bit coding and can be used "as is" or can be used by gaining an in depth knowledge of the system so that the full power of the language can be realised.
Unlike some other systems which allow in line assembler *languageONE* is assembler, so if you want to write some assembler code, just do it.

languageONE is extensible (able to be extended). Because the language is based on macros you can code a solution, give it a macro name and it then becomes part of the language. Additionally, the existing syntax can be changed to suit your programming style. ie. The delivered syntax allows "isGreaterThan", “GT” or “>” in decision making. However, as this is just a predefined token and access-able via an %include file, your are free to modify this to whatever you prefer.

This is one of the things that makes *languageONE* not just another programming language. It makes it THE programming language.

So..What is a macro
a macro is a single instruction that expands automatically into a set of instructions to perform a particular task. And what makes macros so powerful is that the preprocessor of a macro assembler allows you to code parameters that may differ each time the macro is expanded. Consider the following assembler macro:-

```
%macro $initialise 1-2 ' '
        mov  al, %2
        mov rdi, %1
        mov rcx,[%1-8]
        cld
rep     stosb
%endmacro
```

this probably doesn't look too familiar but when coded in *languageONE* it looks like:-

- *Initialise w_FirstName*

This is how you would initialise a data item

LINUX/WINDOWS
Version 1.25 was the final version where Linux and Windows version were separate entities. The version No jumped to V2.00 with the uniting of the two into a single download. This manual assumes that it is addressing Version 2.00

When installing *languageONE* on a Windows system, unpack the compressed download (V9.99.zip), view the w.README.V2.00 text file and follow the instructions.

THE SYSTEM

languageONE is implemented with a number of (o)bject modules that provide functionality to the system. These modules must be statically linked with your assembled code. The command lines for assembling and linking look as follows:-

Assembling using nasm[1]

LINUX:(refer l.assemble in languageONE root directory)

- *nasm -f elf64 <-F dwarf> -w-macro-params $1(where $1 is your program)*

  -F dwarf:used for debugging

WINDOWS:(refer w.assemble.bat in languageONE root directory)

- *nasm -f win64 <-g> -w-macro-params $1(where $1 is your program)*

  -g :used for debugging

LINUX - Linking using ld

LINUX:(refer l.link in languageONE root directory)

- *ld -o $1 $1.o lib/languageONE.l*

WINDOWS - Linking using golink

WINDOWS:(refer w.golink.bat in languageONE root directory)

- *golink /Console /entry _start /debug coff %1.obj @w.golink.libraries*

  coff :used for debugging

WINDOWS - Linking using polink (Refer PellesC)

WINDOWS:(refer w.PellesClink.bat in languageONE root directory)

- *polink /subsystem:console /entry:_start /largeaddressaware:no %1.obj lib\ languageONE.lib @w.pellesC.libraries*

WINDOWS:(refer w.PellesClib.bat in languageONE root directory)

- *polib /machine:x64 /out:lib\languageONE.lib lib*.obj*

---

[1]The **Netwide Assembler** (**NASM**) is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit(IA-32) and 64-bit(x86-64) programs. NASM is considered to be one of the most popular assemblers for Linux.

reWriting

languageONE V1.10a introduced a reWriting process. It is a thin layer that sits between a *languageONE* program and the assembler. It's purpose is to manipulate a program by changing what is presented by the coder into something that NASM requires. It can be considered a pre-processor in the same way that NASM contains a pre-processor. More of this on the following pages. The command line for rewriting,assembling and linking look as follows:-

- *bin/languageONE $1 (where $1 is your program.)* (no extension – but must be ".ONE")

reWriting,Assemble and Link

LINUX:(refer l.makeONE in languageONE root directory)

- l.makeONE $1 *(where $1 is your program.)* (no extension – but must be ".ONE")

WINDOWS:(refer w.makeONE.bat in languageONE root directory)

- w.makeONE $1 *(where $1 is your program.)* (no extension – but must be ".ONE")

WHERE IS EVERYTHING

languageONE

l.pre/w.pre.bat	; commandline pre assemble script	
l.assemble/w.assemble.bat	; commandline assemble script	
l.link/w.golink.bat	; commandline link script	
l.makeONE/w.makeONE.bat	; commandline make utility	
ARCHIVE	: *THE SOURCE FOR ALL PREVIOUS VERSIONS*	
bin	: *LANGUAGE EXECUTABLES*	
l.languageONE	: the "reWRITER"	[Linux]
languageONE.exe	: the "reWRITER"	[Windows]
l.GUI.makeONE	: the (GUI) make utility	[Linux]
Doc	: *DOCUMENTATION*	
languageONE.odt	: this manual (*.odt)	
languageONE.pdf	: this manual (*.pdf)	
languageONE.html	: this manual (*.html)	
LICENSE	: gnu general public license	
README.MD	: github readme	
VERSION.HISTORY	: Text file of all version history	
Html	; *HTML SCREENS*	
GUI-makeONE...	; GUI make utility screens	
V1-15...	; demo program V1.15 screens	
V1-16...	; demo program V1.16 screens	
include	; *LANGUAGEONE CORE COMPONENTS*	
Lib	: *STATICALLY LINKED LANGUAGEONE LIBRARIES & ARCHIVE*	
Src	: *LANGUAGEONE SOURCE CODE*	
Src/examples	*: LANGUAGEONE DEMONSTRATION PROGRAMS*	

Please Note:- *The directory structure when installed with TinyCore Linux is as per the TinyCore's team request. For a better understanding refer to VERSION.HISTORY*

THE LANGUAGEONE "reWRITER"

Overview

HINT:- It may be more useful to skip this section and come back to it.

As mentioned above the *languageONE* "reWRITER" is a pre-processor that manipulates text. It is NOT a compiler nor is it an assembler. (A language rewriter is usually a program that translates the form of expressions without a change of language). It is in fact written in *languageONE* and can be modified and assembled as any other *languageONE* program can be.

NASM and other assemblers implement a pre-processor that presents certain facilities to make coding easier. As such a program as input to NASM may contain pre-processor statements to direct the pre-processor. ie. %macro and %endmacro are both pre-processor directives. NASM's pre-processor implements some unequalled features but is fixed in the way it works. As an example, NASM uses no brackets other than single line macros. Brackets are almost mandatory in many programming languages and such the languageONE "reWRITER" will manipulate their use.

The command line for "reWRITER",assembling and linking is as follows:-

./l.makeONE <program name> (no extension – but must be ".ONE")
This is a bash script

The command line for reWriting only is as follows:-

bin/l.languageONE <program name> (no extension – but must be ".ONE")

GUI utility, l.GUI-makeONE is a browser based utility, written in *languageONE* for reWriting/Assembling & linking.

./l.GUI-makeONE 1024 (where 1024 denotes a port no [must be >1024 and <65535])

"reWRITER" functions:-

Note the terminology used in this manual. Non reWritten code is referred to as 'raw' or 'pure' *languageONE*, while reWritten code is referred to as "cooked" *languageONE*. When describing macros both "raw" and "cooked" syntax will be listed.

Continuation Character

NASM itself recognises the \ (backslash) as a continuation character and as such is also recognised by languageONE (by simply letting it pass through)

Line Splitting

NASM expects one instruction per line. The languageONE "reWRITER" will acknowledge the
"|" character as a line splitting character and split the lines into their components.

Bracketing

All brackets found, other than those on single line macros, will be removed. This gives you a great deal of freedom in the way you wish to code ie.

All of the following may be coded

- IF A = B
- IF (A = B)
- Display “Hello World”
- Display (“Hello World”)
- WHEN ({“Hello World”,1,5} = “Hello”)

In fact because the pre-processor simply removes brackets you could even code IF (()) “Hello”)))))) = ((“Hello”)))))))) But then...would you really want to

Commas

Commas will be inserted here NASM expects to see them.
ie. IF A = B will become IF A,=,B

Synonyms

languageONE encompasses the idea of “synonyms”.
Essentially a synonym may be used at a program level to rename KEYWORDS. This allows you to use the synonym within your program and the "reWRITER" will replace it with a KEYWORD (macro Name). The synonym directives, BEGIN.SYNONYMS and END.SYNONYMS, and the code they enclose, must be coded as comments (starting the line with a semi-colon). The synonym (coded 1st) and the KEYWORD must be separated by a colon. The structure is as follows:-

```
;    BEGIN.SYNONYMS
;          $.       :       WORDS.
;          @.       :       INTEGERS
;          #.       :       NUMBERS.
;    END.SYNONYMS
```

Version 2.07 built on the idea of Synonyms by enhancing the reWriter to manage Alias’s.
A dataname may be given an alias such that the reWriter will recognise the Alias and replace it with the dataname that it refers to. A dataname may be Aliased as often as you like. The Alias is coded identically to a Synonym.

Beginning/End Constructs

BEGIN – END constructs will be counted and reported at the end of the reWriting

```
[000012] Begin Subs          [000012] End Subs
[000012] Begin Repeats       [000012] End Repeats
[000002] IF's                [000002] End Ifs
[000000] Dot Commands        [000000] Dot Ends
[000000] Begin Test's        [000000] End Tests
[000000] When's              [000000] Wend's
[000008] Open Braces         [000008] Close Braces
[000000] Open SqBrackets     [000000] Close SqBrackets
```

Decimal Places

Where NASM sees a decimal point it assigns a value based on its assumption that the program will use the Floating Point Unit. *languageONE* does not in fact use the FPU but rather performs fixed point arithmetic. To that end numbers with decimal places will be manipulated by the "reWRITER" to allow for correct processing. Ie:-

```
insertnumber Test1,-1.23456,'#,###,##9.99999-'
        will be transformed into
insertnumber Test1,-123456,'#,###,##9.99999-'

insertnumber Test2,1.234,'##9.99999-'
        will be transformed into
insertnumber Test2,123400  ,'##9.99999-'

insertnumber Test4,12345,'##9.99999-'
        will be transformed into
insertnumber Test4,1234500000,'##9.99999-'
```

In addition, where a fixed point number is used as a literal following the begin.instructions directive a picture must be supplied. The "reWRITER" will now provide that picture such that you may code only the fixed point number. So, whereas a “raw” *languageONE* program requires for example

```
numbers.add n_Destination,{12345,'99.999'}
        you may now code
numbers.add n_Destination,12.345
```

Inference

The "reWRITER" will 'infer' calculations in the following way:-

```
Code:- w_SomeWord = 'Hello World'
    and the "reWRITER" will infer:-
words.copy 'Hello World',w_SomeWord

Code:- A = B + 2 * 80
    and the "reWRITER" will infer:-
integers.calc A = B + 2 * 80

Code:- A = B + 2.34 * 67.345
    and the "reWRITER" will infer:-
numbers.calc A = B + 2.34 * 67.345
```

Precedence

languageONE uses square brackets to denote precedence.

```
Ie A = [[[B + 2 + 3] / 3] + 123] * [5 + [C * 2]]
Ie A = [[[B + 2.34 + 3.45] / 3] + 123.88] * [5 + [C * 2.45]]
```

- **Table Elements**

  languageONE extends the use of square brackets and in combination with inference can now recognise Table Elements coded in the more traditional format.

  ie n_Integer[Idx]

  Note:- The *languageONE* reWriter cannot infer indexed table elements {ie. Item[1] = 1} coded prior to the BIND command.

DATA DEFINITION

Data in *languageONE* is defined in one of two areas. For strings it is within the Dictionary and for numbers it is within the Matrix.

DICTIONARY

The 'dictionary' is the area of the program where words, phrases and sentences may be described. The *%include 'BEGIN.DICTIONARY'* directive is used to inform the system that the dictionary will begin here. The *%include 'END.DICTIONARY'* directive is used to inform the system that the Dictionary is complete. *languageONE* uses the keyword "INSERTWORD" to define strings within the dictionary. It takes the form:-

Insertword

```
Insertword w_MyFieldName,32,'My Fields Literal'
```

w_MyFieldName is the field name used within the *languageONE* program
32 defines the length of the string
'My Fields Literal' is the initial value given to w_MyFieldName

Valid characters in word names are letters, numbers, _(underscore), $(dollar sign), #(hash), @(at sign), ?(question mark and a .(period). The 1st character must be a letter, a .(period), an _(underscore) or a ?(question mark)

NOTE:-that when the given literal is shorter than the defined string length the field will be padded with spaces thus to initialise a blank field you would code:-

```
INSERTWORD w_Blanks,64,' '
```

Where the literal is longer than the defined length, NASM will give an error.

Qualifying a string

Wherever a string is used within a program it may be qualified by coding a starting position and the number of characters to reference. It takes the form:-

```
{w_MyFieldName,8,16}.
```

This says:- take 16 characters of w_MyFieldName starting at character 8. It must be surrounded by braces.

MATRIX

The 'matrix' is the area of the program where Numbers may be described. The *%include 'BEGIN.MATRIX'* directive is used to inform the system that the Matrix will begin here. The *%include 'END.MATRIX'* directive is used to inform the system that the Matrix is complete.

Each entry begins with the 'keyword' INSERTNUMBER, followed by a name you choose to refer to the item within the program. It is followed by the number and then (Optionally) by a 'picture'. *(Note:- Pictures are required for Fixed Point Numbers)*

Insertnumber

Insertnumber w_MyNumber,100,'#,##9'

w_MyNumber	is the field name used within the languageONE program
100	initialises the number field to 100
'#,##9'	defines it as an integer, displayed in 4 characters with a thousand sign. It will always display 0 if that is the value

The numbers 'picture' describes how the system handles/outputs the number. It comprises a series of characters that define the formatting of the number in a similar way to the BASIC programming language. The following symbols are represented:-

'-' (the minus sign) *[Last character]*
This defines the number as being signed. The number can represent both positive and negative numbers. It should be the last character of the picture.

'9' This defines a place value for the number.
If there is a zero in this position it will be displayed

'#' This also defines a place value for the number
However, a zero in this position will be displayed as a space

',' This defines a thousands indicator and will be displayed when appropriate.

'.' This defines the number as a Fixed Point number.

Alternatives:-

Insertnumber w_MyNumber,100

If a number has no picture, a default 26 character picture will be used. If a number without picture is accepted as input, *languageONE* will assign a picture developed from that input.

ie. accept=1,234.56- Assigned Picture= 9,999.99-

Valid characters in number names are letters, numbers, _(underscore), $(dollar sign), #(hash), @(at sign), and a ?(question mark. The 1st character must be a letter, an _, (underscore) or a ?(question mark)

Integers

Unsigned: *All numbers will be considered as signed*
Signed: From −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
can be represented as integers.

Fixed point numbers

Fixed Point numbers are used in *languageONE*. This is different from many programming languages that use floating point numbers. Floating point numbers require a FPU and are not entirely accurate. (Ever had that experience with a calculator where you enter something like 2 * 2 and get an answer of 3.9999999999

Fixed point numbers in *languageONE* are integers with an 'implied' decimal place. This is implemented via the picture clause of the insertnumber keyword. By defining a picture like '999.99', you are asking for a number that contains 2 decimal places. A picture of say '#,###,##9.9999-' defines a number with 4 decimal places with a sign *displayed* (remembering all numbers are treated as signed).

Because all numbers are actually 64 bit integers the following restrictions apply to Fixed Point numbers. Using *languageONE*'s largest integer (9,223,372,036,854,775,807) the following range is encapsulated

.9223372036854775807
This number can define the size of something smaller than an electron

92233720368547758O.7
This number can count, in seconds, from the big bank until now and be only half filled. It follows that the largest number that contains the largest no of decimal places is 9999999999.999999999 or 999999999.9999999999

Defining Fixed Point Numbers

Because fixed point numbers are held as an integer with a implied decimal place you must acknowledge the number of decimal places in a given value. That is, given a picture of '999.99' and a required value of 123 you must code 12300,'999.99'. Coding 123,'999.99' implies the number is 1.23.

Literals

If a fixed point number is required for a literal then a picture MUST be coded following that literal. ie: {123,'9.99'} equates to 1.23

(NOTE:- this may be handled by the "reWRITER". A pure Fixed Point number may be defined ie A = 1.23 – be aware though that the reWritten version of your program will of had the number converted to its implied Fixed Point value)

Setting precedence

square brackets denote precedents for fixed point number and these are handled by the "reWRITER". Ie A = [[[B + 2.34 + 3.45] / 3] + 123.88] * [5 + [C * 2.45]]

FILES

The 'FILES' section is the area of the program where Files may be described. The *%include 'BEGIN.FILES'* directive is used to inform the system that file descriptions will begin here. The *%include 'END.FILES'* directive is used to inform the system that the file section is complete.

Each entry begins with the 'keyword' INSERTFILE, followed by a file type indicator, a name you choose to refer to the file within the program, and the external name of the file.

Insertfile

Insertfile <Delimiter>,<I-Name>,<'./X-Name'>
where <Delimiter> may be c_NULL,
c_LF,
c_CSV,
'?'
c_RECORD,
c_RANDOM,
c_INDEXED,
c_DIRECTORY,
'Your Own Delimiter'

where <I-Name> is the name used within the program to reference the file
where <'./X-Name'> is the external name of the file

DELIMITERS

c_NULL
Indicates that a NULL value (0x00) is used to delimit logical records. *languageONE* will add this character when writing to the file and strip this character when reading from the file.

c_LF
Indicates that a LINEFEED value (0x0A) is used to delimit logical records. *languageONE* will add this character when writing to the file and strip this character when reading from the file.

c_CSV
Indicates a standard comma delimited file will be used. Alphanumerics will be enclosed in quotation marks, fields will be separated by commas and a LINEFEED character will delimit the logical record

'?' Your own character
Any single character enclosed in quotes

c_RECORD
Indicates that a RECORD will be used to read from and write to the file (refer next section)

c_RANDOM

Indicates that the file can be accessed by record no. RECORDS are implicit in random files.

c_INDEXED
Indicates that the file is optimised for use with the Xtables module. Its format echo's the Xtable format in memory, ie there is an 8 byte index field and a 1 byte status field that precedes each record. It is accessed by record no. RECORDS are implicit in indexed files.

c_DIRECTORY
Indicates that it is a directory that is being read.

The external name may be replaced by a dictionary word and can be allocated dynamically by simply setting it equal to name. ie.

[in the file section] Insertfile c_LF,IN_File

[in the code section] IN_File = “External File Name”

File Status

When accessing files *languageONE* provides a 'STATUS' field. It is defined as FileName_STATUS. Thus the status field for the file A01_File would be A01_File_STATUS. This field can be checked each time a file is accessed.

Present values are:-

10(EOF)	= End Of File
23(INVALIDKEY)	= Invalid Key
22(DUPLICATES)	= More than one record exists for this key
90(LOCKED)	= Record/File is locked

File Size

Additionally, when a file has been opened, *languageONE* provides a 'SIZE' field. It is defined as FileName_SIZE. Thus the size field for the file A01_File would be A01_File_SIZE.

RECORDS

A record is a collection of elements, typically in fixed number and sequence and typically indexed by serial numbers or identity numbers. The elements of records may also be called fields or members.

Records provide for structuring data in a logical way. They are usually, but not always, used to describe data in a file and thus provide clearer code when defined with the file in question. Each entry in a file is described as a record. Records are also mandatory in *languageONE* tables. (refer next section)

Each record entry starts with BEGIN.RECORD keyword. It is followed by a numeric value defining the length of the record, and then the record name. Each record terminates with the END.RECORD keyword followed by the record name. Fields can then be defined within the BEGIN and END of the record by using Insertword and insertnumber.

Record Description

```
begin.record <record_length>,A01_Record
    insertnumber    A01_No1,        00, '99999'
    insertnumber    A01_No2,        00, '99999'
    insertnumber    A01_No3,        00, '99999'
    insertnumber    A01_Word1,      10, ''
    insertnumber    A01_Word2,      10, ''
end.record A01_Record
```

The files module must be informed of the record length and it is Parameter 1 or the BEGIN.RECORD keyword that accomplishes this.

Record No

When a record is used for File access, *languageONE* provides a Record Number which may be used to access records in a random manner rather than sequentially [one after the other]. This access method is termed 'relative' or 'random'. The Record Number is the record name followed by '_NO', so in the case above the record no field will be A01_Record_NO.

TABLES

Tables(Arrays/Lists in other terminology) are defined in *languageONE* as structured internal storage. They are treated very similarly to Files but unlike files must ALWAYS have a record associated with them and thus are almost synonymous with random access files. This allows file records to be read into storage quite efficiently. You insert a table by defining its size (Record Length) times No Of Records that you require.

The 'TABLES' section is the area of the program where tables may be described. The *%include 'BEGIN.TABLES*' directive is used to inform the system that Table descriptions will begin here. The *%include 'END.TABLES*' directive is used to inform the system that the definitions are complete.

Insert Table

Inserttable WorkTable,c_RecordLength*c_NoOfRecords

Inserting a table defines the table name and the overall size of the table, defined as the record length times the number of records. Before a table can be used it must be bound.

Bind Table

Tables.bind (WorkTable,A01_Record,c_NoOfRecords)

While the INSERTTABLE keyword tells *languageONE* about the overall size of the table, TABLES.bind, binds a record to the table and defines the dimensions. It is coded within the body of the program (following the *%include BEGIN.PROGRAM*)

Of course Tables/Arrays can have more than 2 dimensions. *languageONE* allows up to 9 dimensions (you count the record as the 1st dimension.). They are defined by first INSERTing a TABLE with the dimensions detailing the size and then by binding a record to the table. ie TABLES.bind,WorkTable2,A01_Record,3,4

If you use an analogy like pages in a book you could say that the record defines the words across the page, the 3 defines the lines down the page and the 4 defines the pages. You can expand this analogy with an extra dimension defining books. So the index 3,4,5 would define 3 books, 4 pages, and 5 lines leaving the record to define the words across the line.

```
Inserttable WorkTable2,RecLength*3*4*5
Tables.Bind WorkTable,A01_Record,3,4,5
```

Tables are read or written by providing the required index(s) numbers in the get and put operations. Refer further ahead.

Note:- The *languageONE* reWriter cannot infer indexed table elements {ie. Item[1] = 1} coded prior to the BIND command.

XTABLES

XTables(Arrays/Lists in other terminology) are large tables that have been optimised for use with indexed files, they may however be used for any other purpose. They are single dimension tables with memory being allocated at runtime.

InsertXTable

InsertXTable LargeTable

Inserting an XTable defines the table name. Before a table can be used it must be bound.

Bind XTable

XTables.bind LargeTable,Record,Size

While the INSERTXTABLE keyword tells *languageONE* the xtables name, XTABLES.bind, binds a record to the table and defines the tables size. It is coded within the body of the program (following the *%include BEGIN.PROGRAM*)

NOTE:- An XTABLE contains an extra 9 bytes at the front of each record to hold a Delete Flag and an Index and must be taken into account when defining the xTable size.

Size = NoOfRecords*(RecordLength + 9)

Note:- The *languageONE* reWriter cannot infer indexed table elements {ie. Item[1] = 1} coded prior to the BIND command.

ARRAYS (Integer)

An Array – introduced in Version V2.10 - is a contiguous no of integers (all numbers are represented as qwords in languageONE) with memory being allocated at runtime. They do not have the control block normally associated with a languageONE data structure and therefore must be accessed via the ARRAYS.LIB package. Unless manually manipulated indexes have been logically coded, Arrays are always 1 dimensional and a new Quick Sort algorithm has been coded for Arrays in order to optimise the sort speed of integers. The 'TABLES' section is the area of the program where Arrays may be described.

InsertArray

InsertArray ArrayName,NoOfIntegers

SIMPLE INTEGER ARITHMETIC

There are 6 simple arithmetic commands in *languageONE* that are not part of the NUMBERS.o object package but available to every *languageONE* program. It must be noted though that they have an almost 1 to 1 relationship to their underlying assembler equivalents and are not error checked in anyway.

If you add 1 to 9,223,372,036,854,775,807 you will get −9,223,372,036,854,775,808. However, if you are careful, they are the best way to use integers

Integers.eq
(Raw) integers.eq(w_MyNumber,123)
(Cooked) w_MyNumber = 123
will set the value of w_MyNumber to 123

Integers.add
(Raw) integers.add(w_MyNumber,234)
(Cooked) w_MyNumber = w_MyNumber + 123
will add 234 to w_MyNumber

Integers.sub
(Raw) integers.sub(w_MyNumber,345)
(Cooked) w_MyNumber = w_MyNumber - 345
will subtract 345 from w_MyNumber

Integers.mul
(Raw) integers.mul(w_MyNumber,456)
(Cooked) w_MyNumber = w_MyNumber * 456
will multiply w_MyNumber by 456

Integers.div
(Raw) integers.div(w_MyNumber,567)
(Cooked) w_MyNumber = w_MyNumber / 567
will divide w_MyNumber by 567. Note that for this integer function, the remainder will be lost.

Integers.calc
(Raw) integers.calc Result = No1 + No2 – No3 * No4 / No5 .....
(Cooked) Result = No1 + No2 – No3 * No4 / No5 .....
Applies each operation to the destination field working from left to right of the source integers.

Setting precedents

The order of an integer operation my be directed by way of square brackets. As an example:- A = [[8 + B] * 3] + 5 will direct languageONE to evaluate the [8 + B] 1st,multiply the result by 3 2nd and finally add 5 to the answer. The "reWRITER" will enable this by developing the operation one by one within a macro and apply those values to the final calculation. The original line of code will be commented and replaced by the system developed macro name.

In essence square brackets say to languageONE, do this 1st.

LOGICAL EXPRESSIONS

There are 3 logical expressions in *languageONE* that are not part of the NUMBERS.o object package but available to every *languageONE* program. It must be noted though that they have an almost 1 to 1 relationship to their underlying assembler equivalents and are not error checked in anyway.

Integers.and

(Raw) integers.and(w_MyNumber,<Number/literal>)

(Cooked) N/A

will perform a logical 'AND' of w_MyNumber with Number/Literal> placing the result in w_MyNumber

Integers.or

(Raw) integers.or(w_MyNumber,<Number/literal>)

(Cooked) N/A

will perform a logical 'OR' of w_MyNumber with Number/Literal> placing the result in w_MyNumber

Integers.xor

(Raw) integers.xor(w_MyNumber,<Number/literal>)

(Cooked) N/A

will perform a logical 'XOR' of w_MyNumber with Number/Literal> placing the result in w_MyNumber

CONSTANTS

Using %define

are a way of giving a name to a value. It is good programming practise and will save you a lot of headaches further on. As an example a record length may be used in several places and so by replacing the value with the constant c_RecordLength the record length can be changed in one spot (at its definition) and be correct in every use of the constant. Constants are defined by the use of the %define directive.

- %define c_RecordLength 47
- %define c_NoOfRecords 1024
- %define c_Size 47*1024
- %define c_Size (47 * 1024) + 88
- %define c_Size c_RecordLength*c_NoOfRecords

LANGUAGEONE PACKAGES

Introduction

languageONE delivers functionality via keywords. Each keyword is an assembler macro that is used to manage the parameters and make a call to one of the l*anguageONE* object modules. (In this way, if a particular object module is not required you will not have to link it in). Keywords are case-Insensitive such that CURSOR, cursor or even CuRsOr are acceptable forms of coding. With the introduction of the *languageONE* "reWRITER" in V1.10a brackets may be used to surround passed parameters if you choose to do so.

Inline Assembler

There is no package required to include assembler code within a *languageONE* program. As noted *languageONE* IS assembler, but in order to manage structures within the system, the "BEGIN.ASSEMBLER" and "END.ASSEMBLER" keywords are used such that:-

```
begin.assembler
        push rax
        pop rbx
end.assembler
```

may be coded anywhere following the %include 'BEGIN.INSTRUCTIONS' directive. Data may be coded within the data dictionary or the matrix. Uninitialised data may be reserved following the begin.tables directive. In fact data may be coded anywhere within the program as per a normal assembler program. Ie

```
[section .data]
        insertnumber n_Integer,0
[section .text]
```

COMMON.o

The COMMON.o object module provides generalised routines for *languageONE.* It is always linked to all *languageONE* program. In this present release it handles the date routines along with the 'RUN' and the 'WAIT' keywords

date.get

(Raw) date.get (noofdays,datestring)
(Cooked) *No equivalent as yet*

Returns the current number of days and date

- noofdays is an integer value defined as the number of days from the linux epoch (1970/01/01).
- datestring is a 10 character word containing a date formatted as CCYY/MM/DD

date.datefromdays

(Raw) date.datefromdays (noofdays,datestring)
(Cooked) *No equivalent as yet*

Returns the date associated with the No Of Days passed.

date.daysfromdate

(Raw) date.daysfromdate (noofdays,datestring)
(Cooked) *No equivalent as yet*

Returns the No Of Days associated with the Date passed.
This function should be used when validating a date. If the date is invalid then the system field ERROR_CODE will be set to 1

date.seconds

(Raw) date.seconds (integer)
(Cooked) *No equivalent as yet*

Returns the no of seconds from the linux epoch (1970/01/01).

Number of Days is useful in the processing of past or future dates. A date.get will return the no of days associated with the current date. By adding/subtracting a numeric value to the no of days and then performing a date.datefromdays a future or past date can be calculated.

date.seconds is useful in performing “stop-watch” functions. Store the no of seconds prior to starting a process and subtract it from the no of seconds at the process end to produce an elapsed time.

Run

(Raw) run command,ResponseFileHandle
(Cooked) *No equivalent planned*

This keywords allows a program to run another program. Filename must be either a binary executable, or a script. Note that in this release a fully qualified name ie “path/program” must be provided. Ie “/usr/bin/nasm”

- The macro returns a system variable, CHILD_PID defining the new running process's 'Process Identification Number'
- The macro returns a FileHandle that duplicates the child process's STDOUT and STDERR. This file should not be opened by the calling process but must be closed by it

wait

(Raw) wait CHILD_PID
(Cooked) *No equivalent planned*

This keyword, followed by a Process Identification Number,will suspend your program until the program identified by integer has completed

[WINDOWS]
When running a program on a windows system, it has been found that buffers for StdOut are NOT cleared when StdOut is closed or the sub-ordinate program terminates. In the absence of a setbuf or fflush call being available the following work around may be used:-

- Define a file
  insertfile c_LF,STDOUT

- Set the handle to StdOut (Note that the file does not have to be opened)
  push qword[StdOutHandle]
  pop qword[STDOUT_HANDLE]

- Write to this file rather than using display statements
  files.write STDOUT,'Your text goes here'

STDIO.o

The STDIO.o object module provides input/output routines for the Linux terminal. Fields used in a STDIO.o call may be literals, words from the dictionary and numbers from the matrix. The following keywords are supported:-

cursor

(Raw) Cursor (row,column)
(Cooked) *No equivalent planned*

positions the cursor on the screen detailed by the row and column parameters. ie. Cursor 03,05 Note:- cursor sets the values of the system variable c_Cursor. You must perform a “display c_Cursor” for the cursor to be positioned.

Display

(Raw) Display (field,field,field….)
(Cooked) *No equivalent planned*

Displays the listed fields on the screen. Any number of fields may be passed. A useful field to use here could be the predefined field LF. This is the linefeed character and when used as the last field of the display keyword will trigger a linefeed following the last field listed.

display.line

(Raw) Display.line (field,field,field…)
(Cooked) *No equivalent planned*

Displays the listed fields on the screen and follows each one with the linefeed character. It is the equivalent of coding “display field,LF,field,LF,field,LF…”

display.at

(Raw) Display.at (row,column,field)
(Cooked) *No equivalent planned*

Displays the listed field on the screen at the specified row and column position. It is equivalent of coding: “cursor 03,02” followed by “display c_Cursor,field”. Note it makes no sense to display more than one field

acceptline

(Raw) Acceptline (field)
(Cooked) *No equivalent planned*

Accepts the listed field on the screen from the current cursor position. Only a single field at a time can be accepted. The field is terminated when the <ENTER> key is pressed.

acceptline.at

(Raw) Acceptline.at (row,column,field)
(Cooked) *No equivalent planned*

Accepts the listed field on the screen at the specified row and column position. It is equivalent of coding: “cursor 03,02” followed by “acceptline c_Cursor ,field”

accept.at

(Raw) Accept.at (row,column,field)
(Cooked) *No equivalent planned*

Accepts the listed field on the screen at the specified row and column position. This differs from the other accept keywords in that it restricts the number of characters entered to the size of the receiving field. It also reports the escape key being pressed and returns a value of 27 in RETURN_CODE. It is useful when full screen applications are being developed.

Screen Attributes

The following screen attributes may be “display”ed to modify the linux terminal.

- **Terminal Colours**
  - **Foreground**

c_BlackFG	c_WhiteFG
c_DarkGreyFG	c_LightGreyFG
c_RedFG	c_LightRedFG
c_GreenFG	c_LightGreenFG
c_YellowFG	c_LightYellowFG
c_BlueFG	c_LightBlueFG
c_MagentaFG	c_LightMagentaFG
c_CyanFG	c_LightCyanFG
c_DefaultFG	

- **Background**

c_BlackBG	c_WhiteBG
c_DarkGreyBG	c_LightGreyBG
c_RedBG	c_LightRedBG
c_GreenBG	c_LightGreenBG
c_YellowBG	c_LightYellowBG
c_BlueBG	c_LightBlueBG
c_MagentaBG	c_LightMagentaBG
c_CyanBG	c_LightCyanBG
c_DefaultBG	

- **Attributes**

c_Bold	c_ResetBold
c_Dim	c_ResetDim
c_Underlined	c_ResetUnderlined
c_Blink	c_ResetBlink
c_Reverse	c_ResetReverse
c_Hidden	c_ResetHidden
c_ResetAll	c_ClearScreen

- **Graphics Characters**

c_BottomRight	c_TopRight
c_BottomLeft	c_TopLeft
c_LeftMiddle	c_RightMiddle
c_BottomMiddle	c_TopMiddle
c_Cross	c_Line
c_Bar	

WORDS.o

The WORDS.o object module provides string handling routines. They function from left (source) to right (destination). Although there is ".copy" keyword, a copy is always done when a second field is coded, such that you may code- "words.uppercase w_Name" to change the characters in w_Name to upper case and you may also code "words.uppercase w_Name1,w_Name2 to copy the w_Name1 to w_Name2 and change w_Name2 to upper case. In this example w_Name1 would be left unchanged. The following keywords are supported:-

words.copy

(Raw) words.copy (Sourcefield Destinationfield)
(Cooked) *No equivalent planned*

This will copy the number of characters held in Sourcefield to Destinationfield. Reminder: If the Sourcefield is shorter than the Destination field, NO padding will take place. Note that although *languageONE* will accept "words.copy w_Name", it does not make sense to do so.

The sourcefield may be a numeric field and it is this routine that may be used to convert numbers to alphanumerics.

words.pad

(Raw) words.pad (Sourcefield Destinationfield)
(Cooked) *Destinationfield = Sourcefield*

Performs the same function as COPY but will pad a longer Destinationfield with spaces.

words.uppercase

(Raw) words.uppercase (Sourcefield Destinationfield[optional])
(Cooked) *No equivalent as yet*

Will copy the Sourcefield to the Destinationfield and convert the Destinationfield to all upper case characters. Coding "words.uppercase Sourcefield" will convert the Sourcefield to all upper case characters

words.lowercase

(Raw) words.lowercase (Sourcefield Destinationfield[optional])
(Cooked) *No equivalent as yet*

Will copy the Sourcefield to the Destinationfield and convert the Destinationfield to all lower case characters. Coding "words.lowercase Sourcefield" will convert the Sourcefield to all lower case characters

words.insert

(Raw) words.insert (Sourcefield,Destinationfield)
(Cooked) *No equivalent as yet*

Inserts the Sourcefield into the Destinationfield by moving Destination characters to the right. This works well when qualified. ie- words.insert (“RET”,{Destinationfield,5}) says insert the word “RET” into the Destination field starting at the 5th character

words.find

(Raw) words.find ({'RET',counter1},word1)
(Cooked) *No equivalent as yet*

Locates the phrase “RET” in word1 and returns the character position in counter1

words.replace

(Raw) words.replace ({'RET','ROG'},word1)
(Cooked) *No equivalent as yet*

Locates all occurrences of “RET” in word1 and replaces them with “ROG”

words.environment

(Raw) words.environment (Destinationfield)
(Cooked) *No equivalent as yet*

A special WORDS function that will return the command line parameters and store them in Destinationfield

words.length

(Raw) words.length (Word,length)
(Cooked) *No equivalent as yet*

A special WORDS function that will return the length of a word

words.stringtorecord

(Raw) words.stringtorecord (string,record)
(Cooked) *No equivalent*

A WORDS function that will populate a record from a string. This is required because fields are manage by a control structure and records being a collection of fields are managed by multiple control structures

words.recordtostring

(Raw) words.recordtostring (record,string)
(Cooked) *No equivalent*

A WORDS function that will populate a string from a record. This is required because fields are manage by a control structure and records being a collection of fields are managed by multiple control structures

NUMBERS.o

The NUMBERS.o object module provides number handling routines. They function from right (source) to left (destination). Note that although NUMBERS.o can handle integers, the 6 inbuilt integer functions (integer.eq, integer.add, integer.sub, integer.mul, integer.div, integer.calc) are more efficient. Remembering the integer functions have no error correction it is best to use NUMBERS.o for integers when range boundaries are expected to be crossed.

- Unsigned: All numbers will be considered as signed
- Signed: From −9,223,372,036,854,775,807 to 9,223,372,036,854,775,807

numbers.eq

(Raw) numbers.eq (Destinationfield,Sourcefield)
(Cooked) *Nofield1 = Nofield2*

Sets the value of Destinationfield to Sourcefield. The sourcefield in this case may be an alphanumeric field and it is this routine that may be used to covert alphanumerics to numbers

numbers.add

(Raw) numbers.add (Destinationfield,Sourcefield)
(Cooked) *Nofield1 = Nofield? + Nofield?*

Adds the value of Sourcefield to Destinationfield

numbers.sub

(Raw) numbers.sub (Destinationfield,Sourcefield)
(Cooked) *Nofield1 = Nofield? - Nofield?*

Subtracts the value of Sourcefield from Destinationfield

numbers.mul

(Raw) numbers.mul (Destinationfield,Sourcefield)
(Cooked) *Nofield1 = Nofield? * Nofield?*

Multiplies the Destinationfield by the Sourcefield

numbers.div

(Raw) numbers.div (Destinationfield,Sourcefield)
(Cooked) *Nofield1 = Nofield? / Nofield?*

Divides the Destinationfield by the Sourcefield

Overloading

A feature of the NUMBERS.o package is that these macros may be “overloaded”, meaning that more than 1 Sourcefield may be coded. ie:-

Numbers.add Destinationfield,2,4,16,32
This would add 2 then 4 then 16 and then 32 (Total of 54) to the Destinationfield

This can be a useful in many ways. Take for example cubing a number.

With the following defined in the matrix
Insertnumber w_no1, 3
Insertnumber w_no2, 3
You can overload the multiplication to effect a cubing of the number
numbers.mul (w_no1,w_no2,w_No2)

numbers.calc

(Raw) numbers.calc Destinationfield,=,Nofield1,+,Nofield2,*,Nofield3 .....
(Cooked) *Nofield1 = Nofield? + Nofield? * Nofield?*

Applies each operation to the destination field working from left to right of the source numbers (Number precedence will be followed if square brackets are used).

numbers.random

(Raw) numbers.random Receivingfield
(Cooked) *N/A*

Returns a random number in the range 0 thru 65,536.

Note: that this function is not intended to return a random number that can be relied upon for robust applications. It merely takes the number of nano-seconds, splits it into 2 16bit words, xor's the 2 and delivers the result. By definition it has the range of 0 thru 65,536. You may restrict its range by ANDing it with a mask ie: to obtain a number between 0 and 6 (ie a throw of a dice) ADD the returned value by 0110B

If a more “industrial strength” random no is required it can be obtained in:
Linux - investigate the /dev/<u>random file system
Windows - refer CryptoAPI

DECISIONS.o

The DECISIONS.o object module provides for decision making in *languageONE*. In reality it performs one function, that of a compare. The many manifestations of that are handled by the macros themselves.

The following table describes operation equivalences. All are allowable

Equals	EQ	=
isNOTEqualTo	N_EQ	!=
isLessThan	LT	<
isNOTLessThan	N_LT	!<
IsGreaterThanOREqualTo		
isGreaterThan	GT	>
isNOTGreaterThan	N_GT	!>
isLessThanOREqualTo		
_IN		
_NIN		

You are free to add to this table by editing DECISIONS.INC and describing your own operations name

Note: for the decisions module:-

(Raw) ***A shown***

(Cooked) *Do not require commas*

if statements

```
IF w_no1,=,w_no2
        Do Something
ELSE
        Do Something Else
END.IF
```

(Cooked) *Do not require commas*

This is a fairly standard If statement and is no different from most languages. There is no elseif statement in languageONE because it is easily handled. ie.

```
IF w_no_1,=,w_no2
        Do Something
ELSE
        IF w_Alpha,=,“A”
                Do Something
        ELSE
                Do Something Else
        END.IF
END.IF
```

Equally, you could use the line splitting character

```
IF w_no_1 = w_no2
  Do Something
ELSE | IF  w_Alpha = "A"
          Do Something
  ELSE
          Do Something Else
  END.IF
END.IF
```

.IF .OR .AND .END

Compound Ifs *are defined as those that contain '.or' or '.and' statements. They must begin with a '.if' and be terminated with a '.end'*

In order to understand the .AND and the .OR statement you must consider how *languageONE* functions. An IF statement will set a flag as either TRUE of FALSE. Subsequently an .OR statement will proceed if the flag is so to FALSE (no need to do anything if the previous IF returned a TRUE) and set the same flag to either TRUE or FALSE. Similarly an .AND statement will proceed if the flag is set to TRUE (no need to do anything if the previous IF returned a FALSE) and set the same flag to either TRUE or FALSE.

You must be careful with this as *languageONE* has no way (at present) of bracketing OR's and AND's (remember brackets are simply removed) and cannot evaluate something like:-
IF (A=B AND C=D) OR (A=C AND (B=D OR E=F))

The following will achieve the same thing. It's just a little bit cumbersome at present. This will be addressed in a later release.

```
      .IF A = B
    .AND C = D
    .END
    Do Something
  END.IF

      .IF B = D
     .OR E = F
   .AND A = C
    .END
    Do the same Thing
  END.IF
```

the _IN operator

The _IN operator can be used to interrogate lists. Although it is simply a compound “if” it allows for perhaps cleaner coding. Note though that fixed point numbers cannot be coded with the _IN operator as this would result in a double set of braces.

```
IF w_No1 _IN {1,2,4,8,16,31,64}                [YES]
IF w_No1 _IN {{123,'9.99},{456,'9.99}}         [NO]
```

the _NIN operator

The _NIN operator can also be used to interrogate lists. It does in fact invoke the _IN decision and then negates the answer. ie.

```
IF w_No1 _NIN {1,2,4,8,16,31,64}
```

TEST statements

This is a fairly standard statement seen in most languages.
In COBOL it is EVALUATE,
In PASCAL it is CASE OF,
In C it is SWITCH/CASE,
In BASIC it is SELECT CASE etc etc.
It takes the following forms in *languageONE*.

```
BEGIN.TEST w_No1
   WHEN < 1
        Do Something
   WEND

   WHEN < 2
        Do Something
   WEND

   WHEN < 3
        Do Something
   WEND

   OTHERWISE
        Do Something
END.TEST
```

```
BEGIN.TEST c_TRUE
   WHEN A = B
        Do Something
   WEND

   WHEN C > D
        Do Something
   WEND

   WHEN E !> D
        Do Something
   WEND

END.TEST
```

.WHEN .OR .AND .END

Compound Whens *are defined as those that contain '.or' or '.and' statements. They must begin with a '.when' and be terminated with a '.wend'*

```
BEGIN.TEST c_TRUE
   .WHEN A = B
     .AND X = Y
     .END
        Do Something
   WEND

   .WHEN C > D
    .AND J = K
    .END
        Do Something
   WEND

   .WHEN E !> D
      .OR Z = W
    .END
        Do Something
   WEND

END.TEST
```

OTHERWISE may be coded within a TEST statement.

```
BEGIN.TEST c_TRUE
   WHEN A = B
        Do Something
   WEND
   OTHERWISE
        Do Something else
```

```
END.TEST
```

_IN and **_NIN** may be used when testing for TRUE

```
BEGIN.TEST c_TRUE
   WHEN _IN {1,2,3,4,5}
      Do Something
   WEND

   WHEN _NIN {1,2,3,4,5}
      Do Something Else
   WEND

END.TEST
```

LOOPS

The is no object module providing functionality for looping. It is built into the macros themselves as they are only a comparative jump statement. They take 3 forms:-

Note: for languageONE loops:-
(Raw) ***As shown***
(Cooked) *As shown*

repeat <name>

```
Repeat <name>
    Do Something
        exit.repeat <name>
end.repeat <name>
```

A named loop is an endless loop like that shown above. It requires and exit condition coded within the loop such that something like the following would be required

```
repeat _001
    if A = 1
        exit.repeat _001
    end.if
end.repeat _001
```

repeat.if
repeat.while

It should be noted that repeat.if and repeat.while are functionally equivalent. They are both provided to suit programming style.

```
Repeat.if <condition>
    Do Something
end.repeat
```

A repeat.if statement has a built in condition which is evaluated *at the top of the loop*.

When multiple conditions are needed you may use the system supplied exitRepeat field. This flag is always tested at the top of the loop and an exit performed if it is found to be true. Note that any code following the exit will still be executed so it is advisable to code an else to allow your instructions to execute correctly. *Note that .**OR** & .**AND** are not suitable for loops. (This will be revisited in a later version)*

```
repeat.if A = 1
    if B = 1
        exitRepeat = c_TRUE
    else
        Do Something
    end.repeat
end.repeat
```

repeat.for

```
Repeat.for <counter>,<start>,<stop>,<step>
    Do Something
end.repeat
```

A repeat.for statement has a built in count function which is evaluated at the top of the loop. You provide the counter, the starting value, the ending value and the step (if other than 1)

When multiple conditions are needed you may use the system supplied exitRepeat field. This flag is always tested at the top of the loop and an exit performed if it is found to be true. Note that any code following the exit will still be executed so it is advisable to code an else to allow your instructions to execute correctly. *Note that .**OR** & .**AND** are not suitable for loops. (This will be revisited in a later version)*

```
Repeat.for Ctr,1,128,2
    if A = 1
        exitRepeat = c_TRUE
    else
        Do Something
    end.if
end,.repeat
```

Note:-

- when the start value is less than the stop value then the loop will subtract the step value from the counter. You will be counting backwards.
- When the start value is equal to the stop value the loop will add to the step value unless a -ve step value is coded

SUB ROUTINES

Overview

Using Sub Routines is the common programming practice of breaking up programs into smaller and more manageable tasks. They start with the 'keyword' BEGIN.SUB followed by a unique name that you choose. They end with the 'keyword' END.SUB followed by the same unique name you have chosen. They may also contain an EXIT.SUB (followed by the same unique name) to exit the sub routine depending on your coding decisions. To invoke a Sub Routine you use the 'keyword' $CALL followed by the Sub Routines name. Sub Routines may invoke other Sub Routines.

A Sub-Routine name may be followed by a , (comma) and a character (any, but I use a ?) during debugging. It directs *languageONE* to output a message to STDOUT when entering or exiting a subroutine. Ie

- BEGIN.SUB A_Initial,?
- END.SUB A_Initial,?

FUNCTIONS

Overview

Boolean Functions are available in *languageONE.* They start with the 'keyword' BEGIN.FUNCTION followed by a unique name that you choose. They end with the 'keyword' END.FUNCTION followed by the same unique name you have chosen. They may also contain an EXIT.FUNCTION (followed by the same unique name) to exit the function depending on your coded decisions.

Boolean functions are automatically set to return FALSE while a TRUE result may be indicated by setting RETURN_CODE to TRUE.

They maybe coded after the following keywords:-
IF : .OR : .AND : REPEAT.IF (REPEAT.WHILE) : WHEN

ie:-

```
if valid_date
    display “A valid date has been entered”
end.if
```

A Function name may be followed by a , (comma) and a character (any, but I use a ?) during debugging. It directs *languageONE* to output a message to STDOUT when entering or exiting a function. Ie

- BEGIN.FUNCTION ValidNumber,?
- END.FUNCTION ValidNumber,?

EXTENDED FUNCTIONS

Overview

As of version 2.09, the reWriter has been enhanced to offer a user defined return value (as opposed to simply populating RETURN_CODE) and to allow functions to be “square bracketed”, ie Do this 1st.

Details

A return value can be coded, along with its initial value and picture, on the BEGIN.FUNCTION instruction. It must be enclosed by braces and begin with a description of the fields type (trailed by a colon).

```
BEGIN.FUNCTION CompoundInterest,{Number:FutureValue,0,’9999.9999’}
```

Field Type may be Integer, Number or Word.

The reWriter will take this line and create an entry in the Dictionary or Matrix such that the field will be available for use throughout the program. ie..

```
BEGIN.FUNCTION Concatenate,{word:NewText,30,''}

  words.pad Hello,NewText
  words.copy World,{NewText,7}

END.FUNCTION Concatenate
```

After coding the function it may be square bracketed such that the reWriter will modify the code in the following manner:-

display.line [$Call Concatenate] will be transposed into:-

```
$Call  Concatenate
display.line NewText
```

Parameters

Of course the function is less useful without the ability to pass parameters so the following is an example of how to code something of more use:-

- Create the following 2 fields in the Dictionary.

```
Insertword Hello,      6,’’
Insertword World,      5,’’
```

- Create a macro with he same name as the function to load the parameters into the receiving fields.

```
%imacro Concatenate 2
        words.pad %1,Hello
        words.pad %2,World
        Call Concatenate
%endmacro
```

- Use the macro in your program.

```
display.line [ Concatenate (“Hello”,”World”)]
```

FILES.o

The FILES.o object module provides file access routines for languageONE. It manages both sequential, random, indexed and directory access. The following keywords are supported:-

files.open

A file must be opened before it is able to be accessed. (and closed after it is not needed). This is performed by the keyword

files.open <l_Name>,<Action> is used to perform this action, where Actions consist of:-

- **read** The file will be available for reading only
- **write** The file will be available for writing to only
- **readwrite** The file will be available for both reading and writing

When a file is opened write or readwrite you may select starting the write operations from the beginning of the file (thereby overwriting the file) or at the end of the file (appending records to the file). This is achieved by adding (with a + sign) the keywords "$beginning" or "$end". This would look like:-

files.open WorkFile1,write+$beginning

A file may be opened for exclusive use. This is done by adding (with a + sign) the keyword "$lock". This would look like

files.open WorkFile1,write+$beginning+$lock

files.read

There are two forms of the read keyword depending on if you are reading into a record or a series of fields. They are:-

- files.read <l_Name>,record<optional record no(for files of records)>

  ie: files.read (File,Record,Record_No) will yield the same result
  as: integers.eq(Record_No,1)
  files.read (File,Record)

- files $read,<l_Name>,field,field,field.....>

files.write

There are two forms of the read keyword depending on if you are writing a record or writing a series of fields. They are:-

- files.write <l_Name>,record,<optional record no(for files of records)>

  ie: files.write (File,Record,Record_No) will yield the same result
  as: integers.eq(Record_No,1)

files.write (File,Record)

- files $write,<I_Name>,field,field,field.....>

files.delete

Deletion may only take place when files are accessed randomly (with a record). The command takes the form:-

Files.delete <I_Name>,<record>,<optional record no(for files of records)>
and is managed by virtue of the Record No associated with the record. **I_Name_NO**
Although *languageONE* will delete the record regardless, it is always good practise to perform a read operation first. This make it easier to assist any user who's intention it is to delete the record.

ie: files.delete (File,Record,Record_No) will yield the same result
as: integers.eq(Record_No,1)
files.delete (File,Record)

**NOTE:-Records are never physically deleted from a file. *languageONE* reserves the 1st character in a randomly organised file and maintains its value as either:-

- 0x01 represents a “live” record
- 0xFF represents a deleted record
  These records may be restored by editing the file and altering this value to 0x01

**NOTE:-Xtables contains an LOAD and UNLOAD function which makes simple the process of purging deleted records.

files.close

Files must be closed when your program terminates. The command takes the form:

files.close<I_Name>

files.copy

The files module provides a very thin wrapper that allows in kernel copying of files. This is a more efficient method of copying entire files as it is all done within the kernel (as opposed to user space)

files.copy<IN_Name>,<OUT_Name>

Note:- Use Dictionary words. (Null terminated on Linux systems)
insertword INNAME, 08,{'IN_NAME',0x00}
insertword OUTNAME,09,{'OUT_NAME',0x00}

files.rename

The files module provides a very thin wrapper that allows for file renaming.

files.rename<OLD_Name>,<NEW_Name>

Note:- Use Dictionary words. (Null terminated on Linux systems)
insertword OLDNAME, 09,{'OLD_NAME',0x00}
insertword NEWNAME,09,{'NEW_NAME',0x00}

files.remove

The files module provides a very thin wrapper that allows for file deletion.

files.remove<Name>

Note:- Use Dictionary words. (Null terminated on Linux systems)
insertword NAME, 05,{'NAME',0x00}

files.chdir

The files module provides a very thin wrapper that allows directory manipulation.

files.chdir DirectoryName
This changes the current directory to that of DirectoryName. Relative or Absolute.

files.getcwd

files.getcwd DirectoryName
Returns an absolute path of the current work directory

RANDOM ACCESS FILES

The following are commands that are available to random and indexed access files only.

files.start

Because files may contains deleted records and holes a start command will return the 1st valid record and its record number. By initially providing the record number you will determine the starting position. The command takes the form:

files.start<I_Name>,<record>,<optional record no(for files of records)>

**NOTE:-If you create a random access file with 2 records (1 & 3) then a “Hole” will exist in the file. That is a vacant spot in the file containing a record of nulls.

files.next

Following a start, the next routine will return the next valid record (excluding deleted records and holes). The command takes the form:

files.next<I_Name>,<record>

DIRECTORIES

The following are commands that are available when reading a directory.

files.open

A directory is given in the insertfile macro or may be dynamically assigned during program procedures.

files.read

Directories are read sequentially from start to end. Note that the order of files/directories returned may not be consistent or as you may expect. The following RETURN_CODE may be interrogated following this call

- when = 00 : [Linux only] Unknown Entry
- when = 01 : [Linux only] FIFO Entry
- when = 02 : [Linux only] Character Device
- when = 04 : Directory
- when = 06 : [Linux only] Block Device
- when = 08 : File
- when = 10 : [Linux only] Symbolic Link
- when = 12 : [Linux only] Socket
- when = 14 : [Linux only] WhiteOut

files.close

RECORD LOCKING

Record locking is performed slightly differently in the Windows version of *languageONE* as opposed to the Linux version.

Linux : Advisory file and record locking is used to coordinate independent processes. Files and records are not actually locked but there is an agreement between processes that each process will adhere to the protocol. When a process wishes to write a record it is obliged to read the record, lock it , do the write and then unlock it. A second process that attempts to lock the record will receive a warning from the lock function, however reads and writes will continue to function. It is the application logic that supports the locking function.

Windows : Mandatory locking is where the I/O subsystem enforces the locking protocol. A locked record will return a file status of LOCKED whenever an application attempts to read or write that record.

It would have been possible to replicate mandatory locking within *languageONE* for Linux but advisory locking is the preferred protocol. It allows read access where required while leaving the write access to the application. The result of the implementation of locking within *languageONE* means that a Windows program would need to be coded slightly differently from a Linux version, however it is possible to do the coding such that the Windows version would perform as expected on a Linux system (but not visa-versa).

Only files or records can be locked.

files.lock

Files.lock <I_Name>,<record>,<optional record no>
is managed by virtue of the Record No associated with the record. **I_Name_NO**

ie: files.lock (File,Record,Record_No) will yield the same result
as: integers.eq(Record_No,1)

files.lock (File,Record)

files.unlock

Files.unlock <I_Name>,<record>,<optional record no>
is managed by virtue of the Record No associated with the record. **I_Name_NO**

ie: files.lock (File,Record,Record_No) will yield the same result
as: integers.eq(Record_No,1)

files.lock (File,Record)

TABLES.o

Tables, as described in previous parts of this manual, could be visualised as internal representations of random files (although they are more than that). They are however, apart from a need to open or close files, similar in the fact that they must be constructed of records and these records are read (rget) and written (rput). The commands take the following form:

tables.bind

tables.bind (WorkTable,A01_Record,Index1,Index2,Index3....)

this keyword has been described previously in this manual

tables.rget : Read a Table Record

tables.rget (WorkTable,Record_No/Index)

loads the record defined by the record no/index from the table

tables.rput : Write a Table Record

tables.put (WorkTable,Record_No/Index)

places the record defined by the record no/index into the table

tables.sort

tables.sort (WorkTable,StartPosition,NoOfCharacters)

NOTE: *Only single dimension tables may be sorted.*
NOTE: *A sorted table that is not entirely populated may not return the desired result when searched.*

tables.search

tables.search (WorkTable,StartOfKey,EndOfKey,KEY,Index)

returns the record associated with the provided key along with the Index used to locate the record. If more than one record exists, the 1st record will be returned and the STATUS field will be set to DUPLICATES

NOTE: *Only single dimension tables may be searched*
NOTE: *A sorted table that is not entirely populated may not return the desired result when searched.*

tables.fget : Read a Table Field

(Raw) *tables.fget (WorkTable,FieldNo,FieldName,Index,Index,Index...)*
(Cooked) *A = FieldName[Index]*

loads the field defined by the field No from the table, where

- WorkTable is the name of the table
- FieldNo is the field position in the record
- FieldName is the name of the recipient field. It does NOT have to be the corresponding field in the record but must have identical properties.
- Index,Index,Index...the indexes that define the records (containing the field) location in the table

tables.fput : Write a Table Field

(Raw) *tables.fput (WorkTable,FieldNo,FieldName,Index,Index,Index...)*
(Cooked) *FieldName[Index] = A*

stores the field defined by the field No from the table, where

- WorkTable is the name of the table
- FieldNo is the field position in the record
- FieldName is the name of the sending field. It does NOT have to be the corresponding field in the record but must have identical properties. Literals are acceptable as long as they also have identical properties.
  - Numeric literals. You need to provide a picture over-ride to match the receiving fields size. ie. {123,'999'}
  - Alpha fields. Field must be the same size as that defined by the record.
- Index,Index,Index...the indexes that define the records (containing the field) location in the table

Like files, when accessing tables *languageONE* provides a 'STATUS' field. It is defined as TableName_STATUS. Thus the status field for the table A01_Table would be A01_Table_STATUS. This field can be checked each time a file is accessed.

Present values are:-

22(DUPLCATES)	= More than one table element exists for this key
23(INVALIDKEY)	= Invalid Key

languageONE maintains a field named TableName_UBOUND which stores the location of the highest slot within a table. It may be accessed but it must be remembered that UBOUND works in only a single dimension. That is, if you have a table indexed as 2,2,2 then if the table is full, UBOUND would contain a value of 8.

XTABLES.o

Xtables, are tables on steroids, or more precisely, a optimised facility to work with very large files. On a functional level, they may be used in place of an ISAM (Indexed Sequential Access Manager). By sorting on a particular key and searching for values on that key, they are fast enough to enable applications to be written without an ISAM.

A new file type has been defined, that of c_INDEXED, to host xtables on disk. The Indexed file is a direct representation of an XTABLE in memory.

NOTE:- 1] That XTABLES, being bent more toward file access, are only single dimension tables. This does not mean, however, that they cannot be used for other purposes.
2] While Tables are defined at the time of assembly, Xtables are dynamic, that is, memory is allocated at runtime.

xtables.bind

xtables.bind (LargeTable,LargeRecord,Size)

must be used prior to any other xtable operation. It allocates memory for the table as defined by the Size parameter and associates a record with the table.

NOTE: Remember that an XTABLE contains an extra 9 bytes at the front of each record to hold a Delete Flag and an Index and must be taken into account when defining the xTable size.

Size = NoOfRecords(RecordLength + 9)*

xtables.rget

xtables.rget (LargeTable,Record_No)

loads the record defined by the record no from the table

NOTE: If a sort has been performed on the table, obtaining a record based solely on its record number will most times not get the record you are expecting. In order to identify your records location in the table, a search must be performed to return the correct record no.

xtables.rput

xtables.rput (LargeTable,Record_No)

places the record defined by the record no into the table.

NOTE: If the table is being loaded manually, it is your responsibility to ensure that the tables upper boundary (tablename_UBOUND) is large enough to encompass the record no being added.

NOTE: If a sort has been performed on the table, the record no used to replace the table record must be the no returned from a search operation.

xtables.delete

xtables.delete (LargeTable,Record_No,Initialise)

marks the record associated with the record number for deletion. Setting the Initialise field to TRUE will fill the record with NULLS while FALSE will the leave the record intact.

It is dependent on the application to handle theses deleted records. ie. set the IncludeDeletedRecords to c_FALSE on any unload or handle what is written to file within your code.

NOTE:- If a deleted record is initialised you will not be able successfully search the xtable being that it will contain search keys of 0x00

xtables.load

xtables.load (LargeTable,LargeFile,IncludeDeletedRecords)

Although load may be used with random access files, it has been optimised to perform best with indexed files. As indexed files are a direct representation of an xtable, *languageONE* can “swallow” the entire file with a single read. This is opposed to the reading of each record when coupled with a random file. Note though that when loading an Indexed file, the IncludeDeletedRecords indicator is ignored.

xtables.unload

xtables.unload (LargeTable,LargeFile,IncludeDeletedRecords)

Unlike the load function, unload must write each record individually in order to maintain any sort that may have been preformed during processing of the table. IncludeDeletedRecords functions as would be expected for unload.

xtables.sort

xtables.sort (LargeTable,StartPosition,NoOfCharacters,Divisor)

xtables uses an optimised Shell Sort. It does not in fact sort the data in the table, but builds an index to each table entry. A Shell Sort works by dividing the table length by a fixed value giving a span. The sort works through to the top of the table, then divides the previously derived span by that same fixed value. It continues this until the span is equal to 1.

The divisor is critical to the speed of the sort but unfortunately varies from dataset to dataset. There is no 'one' value that can provide the best sort speed for all datasets. *languageONE* allows you to pass the divisor. It must be an integer not less than 11 and not greater than 19. If your application performs the same sort on the same data at regular intervals, you may tune the sort through experimentation (picking the most appropriate value can increase the sort speed by almost 100%). If though, your sort is more like a one off you may need to, through trial and error, determine the best value for 'most' datasets.

xtables.search

xtables.search (LargeTable,StartOfKey,EndOfKey,KEY,Idx)

returns the record associated with the provided key along with the Index used to locate the record. If more than one record exists, the 1st record will be returned and the STATUS field will be set to DUPLICATES

xtables.fget : Read a xTable Field

(Raw) *xtables.fget (LargeTable,FieldNo,FieldName,Index)*
(Cooked) *A = FieldName[Index]*

loads the field defined by the field No from the table, where

- LargeTable is the name of the table
- FieldNo is the field position in the record
- FieldName is the name of the recipient field. It does NOT have to be the corresponding field in the record but must have identical properties.
- Index is the index that defines the records (containing the field) location in the table

xtables.fput : Write a xTable Field

(Raw) *xtables.fput (LargeTable,FieldNo,FieldName,Index)*
(Cooked) *FieldName[Index] = A*

stores the field defined by the field No from the table, where

- LargeTable is the name of the table
- FieldNo is the field position in the record
- FieldName is the name of the sending field. It does NOT have to be the corresponding field in the record but must have identical properties. Literals are acceptable as long as they also have identical properties.
  - Numeric literals. You may need to provide a picture over-ride to match the receiving fields size. ie. {123,'999'}
  - Alpha fields. Field must be the same size as that defined by the record.
- Index is the index that defines the records (containing the field) location in the table

General

Like files, when accessing xtables *languageONE* provides a 'STATUS' field. It is defined as TableName_STATUS. Thus the status field for the table A01_Table would be A01_Table_STATUS. This field can be checked each time a file is accessed.

Present values are:-
22(DUPLCATES) = More than one table element exists for this key
23(INVALIDKEY) = Invalid Key

languageONE maintains a field named TableName_UBOUND which stores the location of the highest slot within a table. It may be accessed directly.

ARRAYS.o

Arrays, as described in previous parts of this manual, are a contiguous run of qwords (8 bytes). They do not have the normal control block that other languageONE data items have and therefore data must be transferred to and from the Array. They are always 1 dimensional unless treated other wise by a logical construction (ie, transforming the index to repesent an array of more than one dimension) . Arrays are useful when speed is required and to this end an optimised Quick Sort has been implimented.

Arrays.Get : Returns an Array Element

Arrays.Get ArrayName,(Value,Index)

returns the array element - located by Index - to Value

Arrays.Put : Stores a value in an Array

Arrays.Put ArrayName,(Value,Index)

stores the data - value - in the Array located by Index

Arrays.Swap : Swaps 2 Array elements

Arrays.Swap ArrayName,(Index1,Index2)

Swaps the Array elements located by Index 1 and Index 2

Arrays.Sort

Arrays.sort ArrayName

An optimised Quick Sort of the Array.

www.o

languageONE manages a graphical user interface by, dare I say, “lever-ageing” web technology. In fact a *languageONE* program supplying a graphical experience to a user is a simple HTML server. In this way it may be accessed locally thru a browser (URL being localhost:portNo) or across either a local or remote network (URL being IPAddress:portno).

Socket technology is also used for (I)nter (P)rocess (C)ommunication. This will be described in the following section.

The program may be defined in 3 simple statements.

- www.open (PortNo)
- www.process (screen,responsefield,GET_Subroutine,POST_Subroutine)
- www.close (PortNo)

and that is it, in its entirety. In fact a complete web site may be managed by the above 3 statements. Of course a great deal of the work is done by the front end that is described by the HTML and perhaps Javascript that may or may not be coded.

V1.15.ONE and V1.16.ONE demonstrate the way that *languageONE* can implement a graphical user interface and V1.16.ONE is replicated in GUI.makeONE.ONE, being the *languageONE* GUI make utility (reWrite/assemble/link).

V2.02 demonstrates a full blown application, while V2.03 demonstrates that same application with record locking.

The *languageONE* program takes a slightly different take on CGI.
Wiki describes CGI as:-

> *In computing, Common Gateway Interface (CGI) offers a standard protocol for web servers to interface with executable programs running on a server that generate web pages dynamically. Such programs are known as CGI scripts or simply as CGIs; though usually written in a scripting language, they can be written in any programming language.*

In regard to the above statement *languageONE* nominates the executable program as taking the lead and supplies the HTML server as a secondary process. By supplying the POST call back address's in the www.process call the *languageONE* program can manage the interface from the back end.

It is important to understand the *languageONE* demonstration programs, V1.15.ONE and V1.16.ONE in addition to the HTML and Javascript associated with these programs, to fully appreciate the end to end process that has been adopted by *languageONE*. It is also taken for granted that everyone nowadays understands web technologies.

NOTE:- WWW.LIB provides a *"passthru"* where a ResponseField populated by your application will be returned directly to a browser. This mechanism is controlled by the RETURN_CODE field. Initialising it with the length of the ResponseField tells WWW.LIB to do the passthru as opposed to the more likely returning of a html document.

V2.02.ONE and V2.03.ONE demonstrate this process.

www.open

www.open (portno)

This is the 1st call required in a *languageONE* GUI program. This call establishes a connection between a *languageONE* program and the outside world (be it a local connection via a "localhost:portno" URL or a "IPaddress:portno" URL across a network (local or otherwise). It is the first call that a *languageONE* GUI program makes in order to interface to a user in a graphical environment.

www.process

www.process (Screen,ResponseField,GET_SubRoutineName,POST_SubRoutineName)

This is the 2nd call required in a *languageONE* GUI program. It defines the screen name that begins a conversation, a response field that will be passed back to a calling program, the name of a sub routine that will be called by WWW.o when a GET is received from the front end and the name of a sub routine that will be called by WWW.o when a POST is received from the front end.

It is the *languageONE* application program that interprets the response field and manages the process.

NOTE:- When a POST only is expected then the value passed for the GET callback address should be set to 0x00

www.process (Screen,ResponseField,0x00,POST_SubRoutineName)

In the same way when a program requires only GET processing then the POST callback address should be set to 0x00

www.process (0x00,ResponseField,GET_SubRoutineName,0x00)

www.close

www.close (portno)

This is the 3rd and final call required in a *languageONE* GUI program. This call closes the previously established connection and terminates the process.

(I)nter (P)rocess (C)ommunication

Inter-process communication(IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests. Many applications are both clients and servers, as commonly seen in distributed computing.

This is managed in *languageONE* by the WWW.o module. Messages can be sent between processes running on a single machine or multiple machines running on a network (including the internet).

Client

www.sendmsg

www.SendMsg w_IP_Address,w_PortNo,w_Msg
www.SendMsg "127.0.0.1","1024",”Hello World”

This command sends a message to the machine described by the socket <IP/Port>.

If your application requires a response it is the applications responsibility to enable that. One way of doing this may be to define a response socket to the server within you message. Ie www.SendMsg w_IP_Address,w_PortNo,”192.168.0.4 1025 Hello World”

Server

www.recvmsg

www.RecvMsg w_IP_Address,w_PortNo,w_Msg
www.RecvMsg "127.0.0.1","1024",w_Msg

The receive message call sets the server into a waiting state where it will listen for any incoming messages. Once it has received one it will return the message to the application program. Note that the server listens for ALL messages on the described port, NOT just one from your application.

APPENDIX A.

How to implement a list in languageONE.

Lists are not particularly part of the language but can be implemented as follows.

In the DICTIONARY, insert a word:-

```
insertword wordlist,17*9        ,{'FIRST LIST ITEM   '    \
                                 ,'SECOND LIST ITEM '  \
                                 ,'THIRD LIST ITEM  '    \
                                 ,'FORTH LIST ITEM  '    \
                                 ,'FIFTH LIST ITEM  '    \
                                 ,'SIXTH LIST ITEM  '    \
                                 ,'SEVENTH LIST ITEM'  \
                                 ,'EIGHTH LIST ITEM '    \
                                 ,'NINTH LIST ITEM  '    \
                                 }
```

- The length [17*9] defines the entire word (ie:- Length of each item times the number of items.
- The list must be enclosed in braces “{}” with each item separated by a comma.
- Each item in the list must be of equal size. (you must define trailing spaces.)
- The list may split across lines by using the “\” (continuation character)

wordlist may then be addressed in the program body in the following manner:-

```
repeat.for (I,1,17*8,17)
   words.pad {wordlist,I,17},wordlistItem
   …
   …
end.repeat
```

APPENDIX B.

KEYWORDS (macros)

Integers

INTEGERS.EQ	INTEGERS.ADD	INTEGERS.SUB
INTEGERS.MUL	INTEGERS.DIV	INTEGERS.CALC
INTEGERS.AND	INTEGERS.OR	INTEGERS.XOR

Sub Routines

BEGIN.SUB	END.SUB	EXIT.SUB

Functions

BEGIN.FUNCTION	END.FUNCTION	EXIT.FUNCTION

Screen IO

CURSOR	ACCEPTLINE	ACCEPTLINE.AT
ACCEPT.AT	DISPLAY	DISPLAYLINE
DISPLAY.AT		

Strings

INSERTWORD	WORDS.COPY	WORDS.PAD
WORDS.UPPERCASE	WORDS.LOWERCASE	WORDS.INSERT
WORDS.FIND	WORDS.REPLACE	WORDS.ENVIRONMENT
WORDS.LENGTH	WORDS.STRINGTORECORD	WORDS.RECORDTOSTRING

Numbers

INSERTNUMBER	NUMBERS.EQ	NUMBERS.ADD
NUMBERS.SUB	NUMBERS.MUL	NUMBERS.DIV
NUMBERS.CALC		

Decisions

IF	.IF	.OR
.AND	.END	
ELSE	END.IF	BEGIN.TEST
WHEN	.WHEN	WEND
END.TEST		

Loops

REPEAT <name>	REPEAT.IF	REPEAT.WHILE
REPEAT.FOR	EXIT.REPEAT	END.REPEAT

Files

INSERTFILE	BEGIN.RECORD	END.RECORD
FILES.OPEN	FILES.READ	FILES.WRITE
FILES.START	FILES.NEXT	FILES.DELETE
FILES.CLOSE	FILES.CHDIR	FILES.GETCWD
FILES.COPY	FILES.RENAME	FILES.REMOVE

Tables

INSERTTABLE	TABLES.BIND	TABLES.RPUT
TABLES.RGET	TABLES.FPUT	TABLES.FGET
TABLES.SORT	TABLES.SEARCH	

Xtables

INSERTXTABLE	XTABLES.BIND	XTABLES.RPUT
XTABLES.RGET	XTABLES.FPUT	XTABLES.FGET
XTABLES.SORT	TABLES.SEARCH	XTABLES.SEARCH
XTABLES.LOAD	XTABLES.UNLOAD	

Arrays

INSERTARRAY	ARRAYS.GET	ARRAYS.PUT
ARRAYS.SWAP	ARRAYS.SORT	

Date

DATE.SECONDS	DATE.GET	DATE.DATEFROMDAYS
DATE.DAYSFROMDATE		

WWW

WWW.OPEN WWW.SENDMSG	WWW.PROCESS WWW.RECVMSG	WWW.CLOSE

Miscellaneous

INTEGERS.TOGGLE	RUN	WAIT (CHILD_PID)
TERMINATE (ERROR_CODE)		

APPENDIX C.

System Supplied Fields

LIBRARY	NAME	STRUCTURE	VALUE	ACCESS
STDIO.o	LF	*languageONE*	0x0A/0x0D0A	Read
STDIO.o	c_Cursor	*languageONE*	0,0	
COMMON.o	c_FALSE	Literal	0	
COMMON.o	c_TRUE	Literal	1	
COMMON.o	ERROR_CODE	*languageONE*	0	Read/Write
COMMON.o	RETURN_CODE	*languageONE*	0	Read/Write
COMMON.o	CHILD_PID	*languageONE*	0	Read Only
COMMON.o	exitRepeat	*languageONE*	c_TRUE/C_FALSE	Read/Write
FILES.o	c_NULL	Literal	1	
FILES.o	c_LF	Literal	2	
FILES.o	c_CSV	Literal	4	
FILES.o	c_RECORD	Literal	8	
FILES.o	c_RANDOM	Literal	16	
FILES.o	c_INDEXED	Literal	32	
FILES.o	c_DIRECTORY	Literal	64	
FILES.o	<filename>_READLENGTH	*languageONE*	0	Read Only
FILES.o	<filename>_STATUS	*languageONE*	0	Read Only
FILES.o	<filename>_SIZE	*languageONE*	0	Read Only
FILES.o	<filename>_HANDLE	*languageONE*	0	Read/Write
FILES.o	EOF	*languageONE*	10	
FILES.o	INVALIDKEY	*languageONE*	23	
FILES.o	<recordname>_NO	*languageONE*	0	
TABLES.o	<tablename>_STATUS	*languageONE*	0	Read Only
XTABLES.o	<tablename>_STATUS	*languageONE*	0	Read Only
XTABLES.o	<tablename>_UBOUND	*languageONE*	0	
WORDS.o	w_CommandLine	256 bytes	Program parameters	Read Only
WORDS.o	w_Spaces	128 bytes	spaces(nn)	

APPENDIX D.

languageONE Structures

```
INSERTWORD
    dq 0                    ; Not Used
    db 'X'                  ; Define it as a string
    dq 0                    ; Define the length of the string
    db ' '                  ; Create the string

INSERTNUMBER
    dq 1                    ; Scaling for Fixed Point
    db '9'                  ; Define it as a number
    dq 0                    ; Defines the length
    dq 0                    ; Using 64 bits with an initial value
    db ' '                  ; the editing picture
```

APPENDIX E.

Debugging a languageONE programming

Basic debugging involves the coding of a 2nd parameter to the BEGIN.SUB and the END.SUB directives. By following the subroutine name by a comma and any character (I use ?) languageONE will display an Entering/Exiting message on STDIO.o. This allows a fairly quick way of determining where bugs may be.

- **LanguageONE in debug mode**
  The difficulty in debugging a *languageONE* program lies in its make up. Because it is really only a collection of assembler macros, most debuggers have problems in aligning the source with the running code. Consider the following simple program:-

```
001         display('Hello World')
```

This code in fact expands to many lines of actual assembler code and so it seems that debuggers then have a problem with finding lines in the source.

The solution to the problem is to produce a source file such that the debugger is able to follow the code line by line. The *languageONE* "reWRITER" has been modified for the purpose and now includes a -d option. 'makeONE -d V1.12a' will produce a file that is formatted as follows.

- Single line Comments are removed from the code following the 'BEGIN.INSTRUCTIONS' directive.
- The keyword 'DEBUG' will be inserted prior to every languageONE keyword. Ie

```
display('Hello World')
                                    DEBUG
display('What is your name')
                                    DEBUG
acceptline w_UserName
```

DEBUG itself is a single line macro that contains the code

```
mov  qword[STOP],1
```

in this way, a watch point may be established in the debugger and set to halt the program whenever STOP = 1. When entering a debug session, issue the following command:-

```
watch STOP if STOP==1
```

All going well, the debugger will then stop at each occurrence of the DEBUG macro, or more to the point, before each line of code to be debugged.

Debuggers

Linux system generally come with GDB, the GNU Debugger. It is a command line debugger and the back end to several GUI front ends.

languageONE debugging works well with the 'ddd', 'Insight' and ‘XXGDB’ GUI front ends and it is advantageous to become familiar with the way they work. As they all function fairly similarly there is no reason you should not use all three.

**WINDOWS. Having not paid for the use of any Windows debugger, I have used the free X64dbg. I have not managed to use this with source level debugging and so it is a little bit of a slog. The issue of debugging a *languageONE* program will be addressed in the future.

DDD Tip:-

copy ~/.ddd/init to init.save and replace the original version with the copy
each time 'ddd' seems to 'act up'.

INSIGHT Tip:-

- Edit the file usr/share/insight[VersionNo]/memwin.itb
- locate the routine itcl::body MemWin::build_win
- immediately preceding the line that reads $itk_interior.f.cntl
  insert the lines

```
set tmp [string first "0" $addr_exp 0]
if {$tmp == "-1"} {set addr_exp &$addr_exp}
```

This ensures that an '&'(Ampersand) precedes you variable name when opening a memory window

- Insight crashes can be reduced by viewing Registers prior to starting your program

APPENDIX F.

Macros

languageONE was built initially on the back of NASM's pre-processor and its ability to provide macros - (a single instruction that expands automatically into a set of instructions)

As such Macro's are native to languageONE and are passed through to NASM's pre-processor. It would be of great advantage to anyone using *languageONE* to become familiar with the way NASM's pre-processor handles these macros.

As an example *languageONE's* “INTEGERS.ADD” keyword names the following macro (stripped down for ease of understanding)

```
%imacro INTEGERS.ADD 2

        %ifnum %2
                add qword[%1],%2
                mov rax,%2
                add qword[%1],rax
        %else
                mov rax,qword[%2]
                add qword[%1],rax
        %endif

%endmacro
```

where the 2 in the %imacro command represents the 2 parameters that may be passed to the macro and the %1 and %2 in he macro body indicate how the parameters are used. This allows *languageONE* to “hide” the assembler code and underpins everything that *languageONE* is.

languageONE's '"reWRITER"' program enhances NASM's macro ability even further and helps give languageONE the more familiar look and feel of a traditional imperative programming language. As mentioned earlier you are encouraged to develop an understanding of the macro process and the following example is provided.

Example

languageONE makes no attempt to pass variables to a subroutine on the stack. 64bit convention states that registers should be used. In addition, all variables (words & numbers) are global in scope. There is no such thing as a local variable in *languageONE*. The general rule here is *"if you don't want to access variables outside of a subroutine then don't access variables outside of a subroutine”*.

The following macro would be used as a simple wrapper that provides the appearance of passing variables. It is in essence the same as what most compilers would provide ie. Move the parameters to an area of memory (in their case the stack) and call the sub routine. In actuality, its not such a bad way to code.

So, where as you would code,

```
n_Test = 1
w_TestDescription = 'passing a string'
$Call TestInteger
```

you may code,

```
TestInteger (1,'passing a string')
```

with the macro being written as,

```
%imacro TestInteger 2

    n_Test = %1
    w_TestDescription = %2
    $Call TestInteger

%endmacro
```

Note that:- Macros can be coded anywhere in the program 'PRECEDING' their use. There location however, demands certain conventions. A macro coded ***before*** the BEGIN.INSTRUCTIONS directive must be coded in native languageONE.

```
ie. 'integers.eq A,B
```

Whereas a macro coded ***after*** the BEGIN.INSTRUCTIONS directive may make full use of the facilities made available by the *languageONE* "reWRITER"

```
ie. A = B
```

APPENDIX G.

Assembler Directives

languageONE is built initial on the back of NASM's pre-processor and as such all the pre-processor directives are available to a *languageONE* program.

I would advise anyone interested in delving deeper into *languageONE* to spend a bit of time going over the NASM documentation. A specific example, from demonstration program V1.16.ONE is detailed here:-

At the top of the program:-

```
%define c_Standard 0                    Define a Constant
%define c_TinyCore 1                    Define a Constant
```

This is the normal way to define a constant in *languageONE* but is in fact passed thru directly to NASM. %define is a NASM pre-processor directive.

```
%assign DIRS c_Standard                 Assign a value to the DIRS token
```

This then assigns a value to the a NASM pre-processor token. %assign is a NASM pre-processor directive.

And then further in the code:-

```
%if DIRS = c_Standard
        w_Command = './bin/l.languageONE'
%endif
%if DIRS = c_TinyCore
        w_Command = 'l.languageONE'
%endif
```

The above will now assemble

- w_Command = './bin/l.languageONE'" if DIRS is set to c_Standard
  or
- w_Command = 'l.languageONE' if DIRS is set to c_TinyCore

In this example provides for conditional assembling where the installed directory structure for *languageONE* is slightly different in the TinyCore version.

languageONE Directives

%FP_DefaultPicture '9999.999999'

This directive gives *languageONE* a default picture size when creating work fields. It may be used more the once and maintains its value until the next directive is coded.

An example of a default picture size is the use of fixed point precedence such that:-

A = [[B * 1.23] / 2.34] directs *languageONE* to

create work fields as it works thru the precedence levels. Note that a divide can create any number of decimal place (ie 22 / 7) so it may help to set the no of decimal places with this directive.