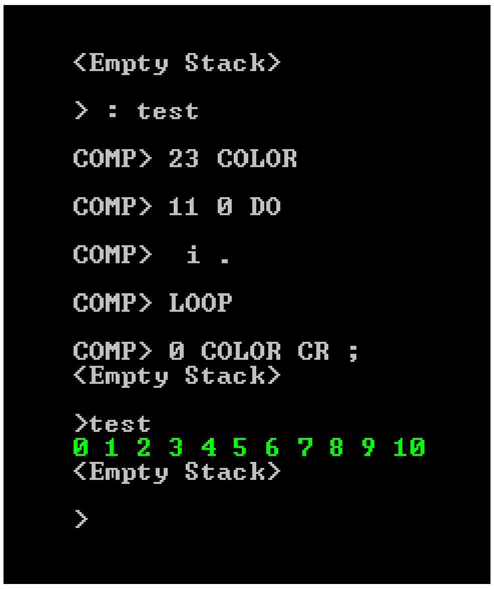
CForth Manual



```
<Empty Stack>

> : test

COMP> 23 COLOR

COMP> 11 0 DO

COMP>  i .

COMP> LOOP

COMP> 0 COLOR CR ;
<Empty Stack>

>test
0 1 2 3 4 5 6 7 8 9 10
<Empty Stack>

>
```

Version 1.0

Vicente Jiménez 20/4/2014

Index

DISCLAIMER AND LICENSE

Disclaimer

This software is provided “as is," and you use the software at your own risk. The author makes no warranties as to performance, merchantability, fitness for a particular purpose, or any other warranties whether expressed or implied. No oral or written communication from the author shall create a warranty. Under no circumstances shall the author be liable for direct, indirect, special, incidental, or consequential damages resulting from the use, misuse, or inability to use this software, even if the author has been advised of the possibility of such damages.

License

This software is provided under version 3 of the GNU General Public License (GPL3)
The details of the license be obtained from the reference below:
https://www.gnu.org/copyleft/gpl.html

1. INTRODUCTION

CForth is a language created for a purpose. I was developing a related project called F3 Gizmo. For this project I created a stack based language. As the language evolved I find a lot of similarities with the Forth programming language. At some point I saw that I was reinventing the wheel and restarted the project separating the hardware layer of the F3 Gizmo from the central software engine. This central software engine, called CForth is quite similar to Forth but it is not the same and it is not ANS Forth compliant at all.
It is important to clarify from the start of this manual that CForth is not Forth and that you cannot use CForth to learn Forth, although if you know Forth it will very easy for you to learn Forth.

Inside this manual the major differences between CForth and Forth will be indicated in gray background **COMPATIBILITY** paragraphs. There is also a full section at the end that deals with these differences.

This manual can be used as a reference manual to CForth and as a tutorial to this language. To support this second use some code examples are included. Having a working system to try the examples increases the utility and fun while reading this document.

CForth is a system independent software product. In order to use CForth it must be integrated into a **port** that provides the system dependent operations. If you have this manual it is because you have one software product that integrates CForth with a **port**.

After all this introduction, let's begin with CForth.

Vicente Jiménez

2. CFORTH BASICS

2.1. Nomenclature

This subsection describes the basic nomenclature used in this manual.

n : Any number

u : Any number equal or greater than zero

c : A character (8 bit signed number)

f : A flag, that is, a number that can be interpreted as FALSE (zero) or TRUE (not zero)

addr : A number that points to one address inside the 32 bit memory space of the system CPU

Uaddr : A number that points to the offset of one memory element inside the User Dictionary (User Memory area)

<word> : A valid word name
Any valid sequence of characters that CForth understands

<uword> : A valir user word name
Any valid sequence of characters defined to perform a user defined functionality

WORD : One specific word name. Although CForth makes no case differences standard word names will be written uppercase to make the stand out.

MFoth is a word based language. Each new word will be associated to a definition. In the first line of a definitions there can be three optional indications:

CForth	Indicates that this word is not Forth compatible
NI	Indicates that this word cannot be used in interactive mode
NC	Indicates that this word cannot be compiled inside a word definition

2.2. The console

CForth interacts with the user using a console. When it starts it writes some information and waits for the user to write orders like in the example below:

```
CForth
Version 1.0 (14/4/2014)
Released under the GNU v3 public license
  Parameter stack size: 50
  Return stack size: 50

Test version. Not to be released
Console ready...

<Empty Stack>
>
```

The exact text written depends on the **port** CForth is associated to, but in any case you will get a prompt.

The last `>` element is the prompt waiting for commands. You can write a series of tokens in the line that starts with the prompt. When you press the return key, all the tokens in the line are processed one by one from left to right.

One CForth line is composed of two kind of characters, normal characters and separators. The separators are the space and the tab, the rest of printable characters are normal ones. A token is defined as any sequence of normal characters. Inside a line two tokens can be separated by any number of separators.

For instance, you can enter the sequence `1 2 +` (followed by the return key) as in the example below. This line includes three tokens, two numbers and a word operator.

```
>1 2 +
<1> Top: 3

>
```

When the return key is pressed CForth processes the two numbers and the operator and returns the control to the console writing a new prompt `>`. The details of what happened in the above example will be explained later in this manual.

CForth basically understands two main kinds of tokens: numbers and words. It also understands character strings but they deserve some special explanations and we won't talk about them yet.

Any token that CForth don't understand as a number is considered as a **word**. Every word that CForth understands is associated to some CForth action. If you introduce a word that CForth don't understand it will complain like in the example below:

```
>unknown
ERROR: Token >>>unknown<<< not recognized in interactive mode
<1> Top: 3
```

You can get the list of words that CForth understands using the word ***WORDS*** , try yourself:

```
> WORDS
```

The definition of WORDS is:

WORDS Show the word list in search order for each dictionary.

You can obtain help about the use of one CForth word using the ***WH*** word.

WH <word> CForth NC Gives help about the use of the word <word> use.

The above two lines are the definition of the ***WH*** word. In this manual we will give a word definition for each word CForth understands. In the definition above it is shown that ***WH*** reaquires a word to follow and that it gives information about it.
The CForth in the first line indicates that this word is not standard in Forth and you probably will only see it in CForth. As ***WORDS*** is standard in any Forth system, no special indication is made.
The NC in the first line indicates (Not for Compile) that this word can only be used in interactive mode, that is, it cannot be compiled inside a new word definition.
Some words cannot be used interactively and can only be used as part of the compilation a new word definition; those words feature the NI identifier.

For instance we can ask about the **+** operator word:

```
>WH +
WH +
Found in Base dictionary:

   +
   Add
   (a)(b) -- (a+b)
```

You can get a list of all words that you can use when CForth starts by calling the ***BASEWORDS*** word.

BASEWORDS	CForth NC
Gives a list of all know base words with their usage.	

You are not limited to use the base words. As you will see you can define your own user words using the base words as construction elements. The ***WORDS*** word lists not only the base words but also the user defined words.

The fact that MFort interacts with the user only with the console means that CForth by itself provides no file operations. Any file operation is system dependent and will be a **port** extension so they won't be covered in this manual.

2.3. Character Case

CForth don't make difference in character case. That way the words DROP, Drop and drop mean the same to CForth.

2.4. Numbers

CForth is designed to work with integer numbers. Floating point numbers are not supported at all.

The numbers can be introduced using three different formats:

Decimal numbers
These are normal numbers you learn at school. They can be positive or negative.
Examples: 67   -453

Hexadecimal numbers
These are 16 base numbers. They can only be positive in CForth. You use them by adding a **0x** prefix.
Examples: 0x4F   0xB0D40

Characters:
These are numbers associated with an ASCII code. You use them writing a single character between single quotes.
Examples: 'A'   '&'

CForth will always show numbers in decimal notation unless you instruct it to use hexadecimal or character notations.

COMPATIBILITY
ANS compliant systems don't use the 0x notation for hexadecimal numbers. They use a special BASE word to set the base of input and output numbers. That makes difficult to use together different base numbers. Compliant systems also don't feature the single quote character notation.

Try yourself
Try to write numbers in the console. Don't forget to hit the return key after each one. The line before the prompt will show the last entered values and the number of values you have entered. For instance, if you write:

```
>1 2 3
<4> Top: 3
```

The second line shows that you have entered 4 numbers and that the last of them is 3. Three of the numbers `1`, `2` and `3` have been entered in the last line.

You can also enter numbers as characters or in hexadecimal notation:

```
> 'A'
<5> Top: 65

> 0xFF
<6> Top: 255
```

2.5. Number storage

CForth uses three storage options for numbers. As the main target for this language is 32 bit systems, the base number storage, called **Cell**, is a 32 bit number.

A part from 32bit numbers, CForth can also use 16 bit **Half Cell** numbers and 8 bit **char** numbers.

All numbers are always stored as signed with 2s complement representation. If you try, for instance, to store 255 in a char space, it will show as -1 on recall.

NOTE: *MFort provides functions to recover the 255 value from the -1 stored value if you know that it was unsigned before storage.*

The limits of the three number formats are:

	Minimum	Maximum
Cell (32 bit)	−2 147 483 648	2 147 483 647
Half Cell (16 bit)	−32 768	32 767
Char (8 bit)	-128	127

This limits and others can be checked anytime using the ***LIMITS*** word.

NOTE: *Cell numbers are special in the sense that for some operations they are treated as unsigned. For instance, when working with 32 bit memory pointers we have no use for negative numbers and we need a full 32 bit unsigned range to access the top half of the memory map. This manual will indicate any operation that uses cell values as unsigned.*

2.6. CForth Stacks

CForth is a stack based language. To operate it uses three stacks: The Parameter Stack, the Return Stack and the System Stack. Both the Parameter Stack and the Return Stack have 32bit Cell size elements. Most operations and functions operate on the Parameter Stack data. The stacks operate in Last In First Out (LIFO) fashion. The last element that enters the stack is the first one that gets out.
There are two main operations that can be done on one FIFO stack:

Push: Put one item on top of the stack increasing the stack size in one. A full stack cannot hold more items. If a push operation is attempted on a full stack a stack **overflow** is produced

Pop: Take one item from the top of the stack decreasing the stack size in one. If the stack is empty when a pop operation is attempted a **stack empty** (underflow) error is produced.

2.6.1. Parameter Stack

The parameter stack in CForth is a **circular** stack. In a circular stack that is full, a push operation don't generate a stack overflow error. To make room for the new item last introduced item on the stack is dropped from below.
Having a circular stack as the main stack enables the user to don't have to guarantee that the stack has room for new incoming data as oldest data are silently eliminated to make room for new one.

The **T**op element **O**n the **S**tack is called **TOS**, while the **N**ext **O**ne on the **S**tack is called **NOS**.

Most MFort operations involve the Parameter stack. For instance, the **+** operation takes two elements from the stack (TOS and NOS), adds them and stores the result on the top (TOS).

After each processed line CForth shows the number of elements on the stack and the value of the TOS. For instance:

```
<4> Top: 3
```

Indicates that there are 4 elements on the stack and that the one that occupies the top is 3.

You can see the contents of the parameter stack with the word ***.S***

```
> .S

<4> 1 2 3 4
<4> Top: 4
```

The stack is shown with the first entered element at the left and the last entered element (TOS) on the right. In the example above the stack holds 4 elements 1, 2, 3 and 4 being 4 the TOS.

2.6.2. Return Stack

The return stack in Forth is a non circular stack. That means that it can give both **overflow** and **undeflow** errors.

This stack is used to hold temporal information and some flow control data.
In standard Forth systems this stack holds also the return address from function calls. This is not the case in CForth.

The ***.R*** word exists in CForth but don't show the return stack. In order to show this stack you should use the ***RDUMP*** word:

```
> rdump

R:<2> 10 20
<4> Top: 4
```

In the above example the return stack holds two elements: 10 and 20 being 20 the top one. As always, the last line shows the number of elements and the TOS of the **parameter** stack not the return stack.

The return stack has a stack frame internal register. Each time a word is executed, the current value of the return stack pointer is copied to the stack frame register. Before leaving the word, the stack frame register is copied back into the return stack pointer. The existence of the stack frame register has two important consequences:

1) You are not forced to pop all the elements you push on the return stack when you are defining a new word. The frame register guarantees that anything you push on the return stack is forgotten when you end the word.

2) You cannot use the return stack to pass data from one word to another. Anything you put on the return stack must be used locally. You should also never pop from the return stack anything than you have not pushed locally.

2.6.3. System Stack

CForth features also a third stack called System Stack that is not accessible to the user. This stack holds the return addresses from function calls and some CForth internal operation data. As the user cannot access this stack it is immune to corruption due to bad use.

Referring "The Stack"

As the parameter stack is the stack used for most CForth operations, when we refer to "**the stack**" we will be talking about the return stack. Any talk about the return or system stacks will be properly qualified to avoid confusion.

2.6.4. Stack diagrams

As the stack is the central element of operation in CForth, it is crucial to know how each function affects to the stack. The stack diagram shows how the stack elements change from before to after one function call.
For instance the **+** word operator stack diagram is:

```
( n1 n2 -- n1+n2 )
```

Stack elements are shown from left to right for elements from bottom to top. In the above diagram `n2` is the TOS and `n1` is the NOS before the **+** operation. After the operation, both NOS and TOS are popped and the `n1+n2` value is pushed as the new TOS element.

The stack diagram only shows the elements that affect the operation. The stack in the **+** operation don't need to have only two elements it just have to include at least two elements. Elements below `n1` in the example stack diagram are omited for simplicity.

Most functions don't use the Return stack so it stack diagram is omitted. If one function uses the return stack, its diagram, with a "R" identification, should be shown next to the one on the return stack. For instance, the ***>R*** word takes the TOS of the Parameter Stack and pushes it as TOS on the Return Stack. So its stack diagram is:

```
( n -- ) R( -- n )
```

That is, `n` is popped from Parameter Stack and pushed on return stack.

COMPATIBILITY
ANS compliant systems don't use circular stacks and usually use the return stack to store return addresses from function calls (The operation that the System Stack carry on CForth). Normal Forth systems don't feature a return stack frame register. That means that in Forth compliant systems you should pop anything you push on the return stack.

2.7. Comments

Comments in CForth are, as in all languages, text zones that are not processed and that are only included to make source code more readable.
CForth provides 3 kinds of comments:

2.7.1. Single line comments

A \ Character marks a comment than will end at the end of the current line. As \ identifies a comment you cannot use this character as part of any word name.

Example:

```
> 4 \We put 4 on the stack
<1> Top: 4
```

COMPATIBILITY
Standard Forth single line comments must include a space after the \ identifier. This is not the case in CForth where \ is not a word.

2.7.2. Parenthesis comments

The (word identifies the start of a comment. All following characters will be discarded until a comment close) character is received.
Parenthesis comments can span several lines.
Remember that, as (is a word, it must be followed by a space or tab.

Examples:

```
> 5 ( We put 5 on the stack)
<2> Top: 5

> 6 (This gives an error)
ERROR: Token >>>(This<<< not recognized in interactive mode
<3> Top: 6
```

2.7.3. Printing comments

The ***.(*** word identifies the start of a printing comment.
Printing comments behave as parenthesis comments but their characters are shown on the screen while they are discarded.
This kind of comments are normally included in the source code to give the user some information about the compilation process.

Example:

```
> 7 .( We put 7 on the stack)
We put 7 on the stack<4> Top: 7
```

2.8. Threads

CForth supports multithreading. This capability must be provided, however, by its **port** integration. If CForth is implemented on a system that don't provide threading capabilities thread words won't be recognized.

When multithreading is active, each thread uses a different set of stacks. All three Parameter, Return and System stack are independent from the ones in other threads. The number of threads that CForth can hold is **port** dependent. As the main target of CForth is medium size microcontroller systems that run code mainly on ROM and feature a limited amount of RAM, the number of simultaneous threads that CForth can support is usually low.

We will see the details of the threads operation later in this manual.

NOTE: *Forth systems use threaded coding compilation. In the case of CForth it uses indirect threaded code. That is not related at all to the multithreading concept.*

2.9. Getting help

This manual tries to give a complete tutorial on CForth operation. Sometimes, however, the explained concepts could require a more detailed explanation. As CForth is based on Forth, the reader can use any Forth reference that explains basic Forth concepts better than this manual. Just beware that CForth is not Forth so you should read this manual to learn about the differences.

3. BASIC WORD LIST

In this section we will explain the basic words used to operate with items in the parameter stack, show information on screen and manipulate the stack contents.

3.1. Arithmetic words

Arithmetic operation words perform arithmetic operations that involve the TOS (unary operators) or the TOS and NOS (dual operators). If you have ever used a RPN (Reverse Polish Notation) calculator the concept is the same.

+ `( n1 n2 -- n1+n2 )`
Pops two elements n2, n1 from the stack in this order. Adds them and pushes the result back in the stack.

- `( n1 n2 -- n1-n2 )`
Pops two elements n2, n1 from the stack in this order. Calculates n1-n2 and pushes the result back in the stack.

***** `( n1 n2 -- n1*n2 )`
Pops two elements from the stack. Multiplies them and pushes the result back in the stack.

/ `( n1 n2 -- n1/n2 )`
Pops two elements n2, n1 in this order from the stack. Calculates the integer division n1/n2 discarding the residue and pushes the result back in the stack.

MOD `( n1 n2 -- n3 )`
Pops two elements n2 and n1 in this order from the stack. Calculates n3 as the residue from the integer division n1/n2 and pushes it on the stack.

/MOD `( n1 n2 -- n3 n1/n2 )`
Pops two elements n2, n1 in this order from the stack. Calculates n3 as the residue from the integer division n1/n2 and pushes it on stack. Next calculates the integer division n1/n2 and pushes it on the stack.

ABS `( n -- |n| )`

Calculates the absolute value of the TOS

MAX `( n2 n2 -- max(n1,n2) )`
Pops two numbers of the stack and pushes the greater of them.

MIN `( n2 n2 -- max(n1,n2) )`
Pops two numbers of the stack and pushes the lesser of them.

NEGATE `( n -- (-n) )`
Changes the sign off the TOS

It is very usual to operate the TOS with numbers 1 and 2. CForth has several words to ease these operations. The stack diagram makes unnecessary any further explanations:

1+	(n -- n+1)
1-	(n -- n-1)
2+	(n -- n+2)
2-	(n -- n-2)
2*	(n -- n*2)
2/	(n -- n/2)

You should use the above operations instead of two words ***1*** and **+** for instance. The above operators are coded with only one byte, whereas ***1*** followed by **+** is coded with three bytes and requires more execution time.

Observe that you don't need parenthesis **(** or **)** when operating with numbers in CForth. Using the stack makes them unnecessary. For instance, lets compute the following operation:

$$5 * (4 + 3 * (1 + 2))$$

In CForth you can use the following sequence:

```
> 1 2 + 3 * 4 + 5 *
<1> Top: 65
```

The best way to understand the CForth way to operate is by trying yourself doing some operations.

3.2. Relational operator words

Relational operators convert integer numbers in flag values. A flag value is a value that can only be interpreted as **true** or **false**.
In CForth zero is considered **false** and any other number is considered **true**. Although any not zero value is true, it is convenient, as we will see later, to use -1 as the canonical true value.

FALSE (-- 0)
Pushes a 0 number, all bits at 0, on the stack.

TRUE (-- -1)
Pushes a -1 number, all bits at 1, on the stack.

All relational operators leave as TOS a flag that can only be a 0 (**false**) or -1 (**true**).

The following relational operators pop two items from the stack, compares them, and push the comparison result back in the stack.

< `( n1 n2 -- f )`
Pushes TRUE if n1<n2. Otherwise pushes FALSE.

> `( n1 n2 -- f )`
Pushes TRUE if n1>n2. Otherwise pushes FALSE.

<= `( n1 n2 -- f )`
Pushes TRUE if n1<=n2. Otherwise pushes FALSE.

>= `( n1 n2 -- f )`
Pushes TRUE if n1>=n2. Otherwise pushes FALSE.

= `( n1 n2 -- f )`
Pushes TRUE if n1 is equal to n2. Otherwise pushes FALSE.

<> `( n1 n2 -- f )`
Pushes TRUE if n1 is different of n2. Otherwise pushes FALSE.

As is quite usual to compare numbers with zero there are several words that compare the TOS with zero. The following words pop one item from the stack. Compare it with zero and push the result back in the stack.

0= `( n -- f )`
Pops the TOS. If it is zero pushes TRUE. Otherwise pushes FALSE.

0< `( n -- f )`
Pops the TOS. If it is less than zero pushes TRUE. Otherwise pushes FALSE.

0> `( n -- f )`
Pops the TOS. If it is more than zero pushes TRUE. Otherwise pushes FALSE.

0<>`( n -- f )`
Pops the TOS. If it is not zero pushes TRUE. Otherwise pushes FALSE.

NOT `( n -- f )`
Is equivalent to ***0=***. Compares the TOS with zero.

3.3. Bitwise logical operations

The bitwise logical operations operate bit by bit on the 32 bits of the stack elements. As they operate bit by bit they don't consider the 2's complement storage format of Cell data. In other words, bitwise logical operators consider stack numbers as unsigned 32 bit values. Fortunately all positive 2's complement numbers have the same internal representation that the equivalent unsigned numbers so you could only get into problems for very large numbers.

AND `( n1 n2 -- n3 )`
Sets to “1” the n3 bits that are “1” both in n1 and n2.

OR `( n1 n2 -- n3 )`
Sets to “1” the n3 bits that are “1” in n1 or n2.

XOR `( n1 n2 -- n3 )`
Sets to “1” the n3 bits that are “1” in n1 or n2 but not in both.

INVERT `( n1 -- n2 )`
Changes all bits in the TOS from “1” to “0” or from “0” to “1”.

Inverting **false** gives the canonical **true**:

```
> 0 INVERT
<1> Top: -1
```

Beware that in order to use the bitwise operators AND, OR, XOR and INVERT for logical (**true**, **false**) computations you need to use -1 as the **true** representation. One non canonical true value as 7 can be converted to the canonical -1 using the ***0<>*** operator.

LSHIFT `( n1 u -- n2 )`
Pops a non negative value u from the stack. Then pops a n1 number. Finally pushes the n2 number that is obtained shifting all bits in n1 u bit places to the left.
The u LSB bits of n1 will be zero after this operation.

n1 u LSHIFT is equivalent to multiply n1 by 2^u:

```
> 7 2 LSHIFT
<2> Top: 28
```

RSHIFT `( n1 u -- n2 )`
Pops a non negative value u from the stack. Then pops a n1 number. Finally pushes the n2 number that is obtained shifting all bits in n1 u bit places to the right.
The u upper MSB bits of n1 will be zero after this operation.

n1 u RSHIFT is equivalent to divide n1 by 2^u:

```
> 48 3 RSHIFT
<3> Top: 6
```

3.4. Data conversion words

As was explained before, the standard coding of numbers in CForth is 2's complement. So, a byte value, for instance, can hold a number between -128 and 127. Sometimes you want to store a number between 0 and 255 inside a **Char** storage. In this case the conversion words described here come handy.

U8S `( uint8 -- int8 )` CForth

This word converts a 8 bit unsigned number between 0 and 255 in the 2's complement number that uses the same 8 bit coding.

S8U `( int8 -- uint8 )` CForth

This word converts a 8 bit signed number between -128 and 127 in the unsigned 8 bit number that uses the same 8 bit coding.

For instance you can convert 255 to signed and back to unsigned:

```
> 255 U8S
<1> Top: -1

> S8U
<1> Top: 255
```

In general you won't need to use the ***U8S*** word as CForth automatically does this conversion when you try to put a number in **Char** storage. In fact this is the typical *wrap around* on overflow effect. CForth don't make any attempt to prevent this effect so you should be aware that 127+1=-128 when dealing with **Char** storage.

Similar words are defined for 16bit Half Cell integers.

U16S `( uint16 -- int16 )` CForth

This word converts a 16 bit unsigned number between 0 and 65535 in the 2's complement number that uses the same 16 bit coding.

S16U `( int16 -- uint16 )` CForth

This word converts a 16 bit signed number between -32768 and 32767 in the unsigned 16 bit number that uses the same 16 bit coding.

You can try the same 8 bit example for 16 bit numbers:

```
> 65535 U16S
<1> Top: -1

> S16U
<1> Top: 65535
```

Like in 8 bit numbers the ***U16S*** conversion is automatic when storing in **Half Cell** storages. Also a similar wrap effect takes place in 16 bit numbers.

There are no ***U32S*** or ***S32U*** words. As 32 bit is the native **Cell** stack size it is not needed. The previous described words are needed because **Half Cell** and **Char** storages are of different size than stack Cells.

Beware that the wrap effect takes also place in 32 bit Cell numbers:

```
> 1 31 LSHIFT
<1> Top: -2147483648

> 1 -
<2> Top: 2147483647
```

Fortunately you need very big numbers to see this effect.

3.5. Console Display

This section deals with basic showing of data on the console. We will show how to send information to the console so it could be shown.

.   `( n -- )`
Pop TOS and display it on the console followed by a space.

X.   `( n -- )`   CForth
Pop TOS and display it on the console as an hexadecimal number followed by a space.
This considers the stack data as 32 bit unsigned values.

U.   `( n -- )`
Pop TOS and display it on the console as a decimal unsigned number followed by a space.

See how the -1 number is considered as signed using ***.*** and as unsigned using ***X.*** or ***U.***:

```
> -1 .
-1 <Empty Stack>

> -1 X.
ffffffff <Empty Stack>

> -1 U.
4294967295 <Empty Stack>
```

The previous words use a different number of chars depending on the number shown. Sometimes a fixed size fied must be shown.

.R `( n u -- )`
Pops a non negative u number and then a number n. Shows the n number so that the total number of written chars is u adding padding spaces to the left if needed.

X.R `( n u -- )` CForth
Pops a non negative u number and then a number n. Shows the n number in hexadecimal notation so that the total number of written chars is u adding padding spaces to the left if needed.
This word, as the X. word, considers stack Cell data as unsigned 32 bit numbers.

Example:

```
> 94 4 .R
  94<Empty Stack>

> 94 4 X.R
  5E<Empty Stack>
```

The following two words show the contents of the parameter and return stack. They were previously introducing when describing the CForth stacks.

.S
Shows the parameter stack contents in one line.
The number of stack elements is shown at the left side followed with the stack contents from bottom to top.

.RDUMP CForth
Showns the return stack contents in one line.
The number of stack elements is shown at the left side followed with the stack contents from bottom to top.

All the previous words are used to write numbers on the console. In order to write characters you can use one of the following commands:

BS
Writes a backspace on the console.

CR
Writes a line break on the console.

SPACE
Writes a space character on the console.

SPACES `( n -- )`
Writes n character spaces on the console

EMIT `( n -- )`
Writes on the console the character whose ascii code is n.

Example:

```
> 'A' EMIT SPACE 'b' 1+ EMIT CR
A c
<4> Top: 4
```

The line break character sequence is console dependent. The ***SETBREAK*** word configures the character sequence that defines a line break.

SETBREAK `( n -- )` CForth
Changes the line break character sequence.
n can be one of the following values:

0	CR(13)+LF(10)
1	CR(13)
2	LF(10)

CForth uses the console to display several kinds of information data. The amount of data that the console shows is configurable using the VERBOSE word.

VERBOSE `( n -- )` CForth
Sets the verbose level to n. The verbose level is a sum of power of two values that determine which information is shown on the console.

Each one of the values defines one information element:

1	ECHO	Activates echo of each input char
2	ERROR	Activates showing of error messages
4	RESPONSE	Activates showing word responses
8	SHOW TOP	Activates showing the TOS after each line
16	INFO	Activates information about words
32	PROMPT	Activates the prompt at the start of each line
64	DEBUG	Activates debug messages

Verbose level is a per thread information. As all non main threads run in the background their verbose levels default to 0 to prevent their output to be intermixed with the foreground action.

A Verbose level of 127 includes all information items.

In the following example the SHOW TOP element weight is eliminated so the top won't be shown after each line:

```
> 127 8 - VERBOSE
>
```

3.6. ANSI Terminal Words

If you use a very dumb terminal the above words will be the only ones you can use to print information. Chances are, however that you use a more capable terminal.
If you terminal support the ANSI escape sequences you can use the commands described in this s subsection.

AT-XY `( nx ny -- )`
Changes the cursor position to the nx, ny indicated values. The upper left corner is defined as the 0,0 origin.

COLOR `( n -- )` CForth
Changes the color. n can be any of the following colors:

0	Default foreground and background
1	Black foreground
2	Red foreground
3	Green foreground
4	Yellow foreground
5	Blue foreground
6	Magenta foreground
7	Cyan foreground
8	White foreground
11-18	High intensity foreground (aixterm)
21-27	Bold color foreground
100	Default background
101-108	Background colors

Example:

```
> 3 COLOR .( This is a green comment) 0 COLOR CR
```

CSI CForth
Control Sequence Introducer.
Writes the two character sequence ESC+'['

PAGE
Clears the screen and sets the cursor to the upper left corner.

Example:

```
>PAGE 22 COLOR .( Color 22) SPACE 26 COLOR .( Color 26) 0 COLOR CR
```

3.7. Parameter stack operations

CForth uses the Parameter Stack as the main parameter and result storage. As this stack is used a lot, there are several words that ease its manipulation.
For most of these words the stack diagram is all information you need to know how they operate.

CLEAR `( a1 ... au -- )` CForth
Erases the Parameter Stack

DEPTH `( -- u )`
Obtains the current number u of elements on the stack and pushes this number. After the word is executed there will be u+1 elements on the stack.

Example:

```
> CLEAR 10 20 30 DEPTH .S CR
<4> 10 20 30 3
```

DROP `( n -- )`
Removes the TOS of the stack.
If you try to drop from an empty stack you will get a run error:

```
> CLEAR DROP
RUN ERROR: Stack is empty
<Empty Stack>
```

DROPN `( a1..au u -- )` CForth
Removes u+1 elements from the stack. Including the u number itself.

As an example the sequence ***DEPTH DROPN*** is equivalent to ***CLEAR***:

```
> 10 20 30 40 DEPTH DROPN
<Empty Stack>
```

DUP `(n -- n n)`
Duplicates the TOS.

?DUP `( 0 -- 0 | n -- n n )`
If the TOS is not cero duplicates it.
Does nothing if the TOS is zero.

DUPN `( a1..au u -- a1..au a1..au )` CForth
Pops a u number from the stack. Then duplicates the following u numbers.

NIP `( n1 n2 -- n2)`
Removes the NOS from the stack.

OVER `( n1 n2 -- n1 n2 n1 )`
Pushes the NOS over the TOS.

PICK `( au ... a1 a0 u -- au .. a1 a0 au )`
Pops a u number from the stack. Then pushes a copy of the element that is in the u position below the top.

Examples:

```
> 1 2 3 0 PICK .S \0 PICK is equivalent to DUP
<4> 1 2 3 3

> 1 2 3 1 PICK .S \1 PICK is equivalent to OVER
<4> 1 2 3 2
```

ROT `( n1 n2 n3 -- n2 n3 n1 )`
Rotates the first three stack elements.

ROLL `( au .. a1 a0 u -- au-1 .. a1 a0 au )`
Pops a u number from the stack. Then rotates u stack elements.

2 ROLL is equivalent to ROT:

```
> 10 20 30 2 ROLL .S CR
<3> 20 30 10
```

SWAP `( n1 n2 -- n2 n1 )`
Interchanges the TOS and the NOS

SWAPN `( au .. a0 u -- a0 au-1 .. a1 au )` CForth
Pops a u number from the stack. Then swaps the TOS with the element that is at position u below it.

For instance 0 ***SWAPN*** does nothing and 1 ***SWAPN*** is equivalent to ***SWAP***:

```
> 1 2 3 0 SWAPN .S CR
<3> 1 2 3

> 1 SWAPN .S CR
<3> 1 3 2
```

TUCK `( n1 n2 -- n2 n1 n2 )`
Inserts a copy of the TOS below the NOS.

3.8. Return Stack operations

The return stack is mainly used in CForth as a temporal storage space. You don't normally need to manage that stack directly so there are only a few words that deal with this stack.

>R `( n -- ) R( -- n )`
Pops one item from the parameter stack and pushes it on the return stack.

R> `( -- n ) R( n -- )`
Pops one item from the return stack and pushes it on the parameter stack.

You can use the above words to change the processing order of the elements of the stack:

```
> CLEAR 1 2 3 . . . CR
3 2 1

> CLEAR 1 2 3 >R >R >R R> . R> . R> . CR
1 2 3
```

R@   `( -- n ) R( n -- n )`
Pushes on the Parameter Stack a copy of the top element of the Return Stack.

RDROP   `R( n -- )`
Drops the top of the Return Stack

RCLEAR   `R( a1 ... au -- )`   CForth
Erase the return stack

There are very few cases where clearing the return stack is a good idea. The return stack holds counting loop parameters and local variables information. You'd better don't use this word if you don't know what you are doing.
You should not pop or drop any return stack element that you have not pushed before as it can give very bad results.

The last 3 words that can be used on the return stack relate to the counting loops that will be explained later in this manual.

I   `( -- n ) R( n -- n )`
Gets index from inner do loop

J   `( -- n3 ) R( n3 n2 n1 -- n3 n2 n1 )`
Gets index from second (enclosing inner) do loop

K   `( -- n5 ) R( n5..n1 -- n5..n1 )`   CForth
Gets index from third (enclosing second) do loop

4. STATIC DATA STORAGE WORDS

This section deals with static data storage.
Static data is stored in CForth in a user region denominated the **User Dictionary**. This region is shared with the code associated to any user defined word. Data storage elements are available to any code developed by the user.
You can use the ***UNUSED*** word to obtain the number of free bytes in the user memory:

```
> UNUSED
<1> Top: 4076
```

In the example we see that we have 4076 bytes of memory free.

4.1. Constants

The CONSTANT word lets you define a constant. That is, a value that cannot change after its definition.

CONSTANT <word> `( n -- )`
Creates a constant storage word.
Pops a number from the stack and a word name from the console and defines a new constant word with the provided name that holds the popped number value.

To use a constant you just need to use its name.

Example:

```
> 2014 CONSTANT THIS_YEAR

> THIS_YEAR . CR
2014
```

Constant words are stored using the minimum storage size needed. So a constant can be stored as a Cell, Half Cell or Char.

This is a good place to introduce several information words:

UWORDS CForth
List the user defined word names from newer to older.

```
> UWORDS
User dictionary:
  THIS_YEAR
```

ULIST CForth
List the user defined word names from newer to older.
Adds also some information about its contents.

```
> ULIST
User dictionary listing:
  THIS_YEAR = 2014 (16bit Constant)
```

4.2. Values

CForth provides two basic kinds of storage elements that can be modified at runtime: Variables and Values. Variables are the classic way to deal with data storage in Forth, but they are somewhat complex to use so we will first explain Values as they are easier on the user.
For first time users it is recommended to use Values as the main storage method and leave Variables to the uses where they are really needed.

To create a new value you just need to use the ***VALUE*** word. Its syntax is exactly the same as the used in constants:

VALUE <word> `( n -- )`
Creates a Cell (32 bits) value storage word.
Pops a number from the stack and a word name from the console and defines a new value word with the provided name that holds the popped number value.

To use a previously defined value you just need to use its word name in the same way that constants:

```
> 2014 VALUE YEAR

> YEAR . CR
2014
```

To this point, the values work exactly the same as constants. The difference is that you can change values using the ***TO*** word.

> ***TO*** <word> `( n -- )`
> Sets to n the value whose name is <word>

For instance you can change the previously defined YEAR value:

```
> 2012 TO YEAR
> YEAR . CR
2012
```

You can also list it in the current user word list:

```
> ulist
User dictionary listing:
  YEAR = 2012 (32bit Value)
  THIS_YEAR = 2014 (16bit Constant)
```

If you want to add a number to a value you can get the value, add the number and store the result:

```
> YEAR 2 + TO YEAR   YEAR . CR
2014
```

Or you can use the ***+TO*** word:

> ***+TO*** <word> (n --) CForth
> Adds n to the value whose name is <word>

For instance:

```
> 2 +TO YEAR   YEAR . CR
2016
```

The values defined with the ***VALUE*** word feature Cell storage, so they can hold a 32 bit number. For 16 bit numbers an equivalent ***HVALUE*** word can be used. Finally ***CVALUE*** provides values with 8 bit storage.

> ***HVALUE*** <word> `( n -- )` CForth
> Creates a Half Cell (16 bits) signed value storage word.
> Pops a number from the stack and a word name from the console and defines a new value word with the provided name that holds the popped number value.

CVALUE <word> (n --)
Creates a character (8 bits) signed value storage word.
Pops a number from the stack and a word name from the console and defines a new value word with the provided name that holds the popped number value.

You can use the same ***TO*** and ***+TO*** words with Half Cell and Char values in the same way that they are used with Cell values. Just beware the wrapping effect that takes place if you try to store a number out of the limits of the storage.

4.3. Variables

Values are easy to use and understand so they are the preferred static storage value in CForth. I don't recommend using variables when values work well. Just leave variables to the special needs where values are not suitable.
The second kind of static storage, **variables**, are more complex to use but are also more powerful and, in fact, they are the basic storage elements in Forth systems.

The use of variables is quite different from the use of values. Comparing with the “C” language, you can think of values as the normal variables in "C" and you can think of variables as pointers in "C".
I was tempted to call them pointers in CForth but that will sure puzzle and infuriate Forth programmers.

In the same way that you don't use pointer for normal storage in "C", you don't use variables for normal storage in CForth. That's at least my recommendation.

You can define a new variable using the ***VARIABLE*** word:

VARIABLE <word>
Creates 32 bit (Cell) variable and sets it to zero.

Observe that you don't provide a number as in constants and values. In standard Forth systems a variable could be undefined at creation time. In CForth, however, variables are always initialized to zero.

If you use the word name of a variable, it pushes on the stack not the contents of the variable but the position of the variable in memory.

Let's create a variable as an example:

```
> VARIABLE MYVAR
> MYVAR
<1> Top: 4297801
```

The 4287801 number is not the contents of the variable but the position of the variable in memory that, in this example, is from a 32 bit windows system.

To obtain the variable contents you need to use the fetch **@** word.

@ `( addr -- n )`	
Recalls a 32 bit signed number n from the memory position addr in the TOS.	

In the previous example remember that we had the MYVAR location in memory in the TOS, so we can use **@** to recall its contents:

```
<1> Top: 4297801
> @
<1> Top: 0
```

! `( n addr -- )`	
Stores the number n on a 32 bit signed variable located in the memory position addr.	

As an example:

```
> 25 MYVAR !   MYVAR @
<2> Top: 25
```

It's not difficult, but it is just more complex than using values.

In the same way that the ***+TO*** word in values, there is ***+!*** word to add a number to a variable.

+! `( n addr -- )`	
Adds the number n to a 32 bit signed variable located in the memory position addr.	

As an example:

```
> 10 MYVAR +!   MYVAR @
<3> Top: 35
```

You can see now the three kinds of storage elements using the ***ULIST*** word:

```
> ULIST
```

```
User dictionary listing:
  MYVAR = 35 (32bit Variable)
  YEAR = 2016 (32bit Value)
  THIS_YEAR = 2014 (16bit Constant)
```

The described variables use 32 bit Cell storage. In order to use 16 or 8 bit storages you need to define the variable with ***HVARIABLE*** or ***CVARIABLE***.

HVARIABLE <word>
Creates 16 bit (Half Cell) variable and sets it to zero.

CVARIABLE <word>
Creates 8 bit (Half Cell) variable and sets it to zero.

The problem with variables is that when they are used, the only information on the stack is their address. Any information about their storage size is not available. That means that the store and fetch words need to be different for Cell, Half Cell and Char variables.

H@ (addr -- n)
Recalls a 16 bit signed number n from the memory position addr in the TOS.

H! (n addr --)
Stores the number n on a 16 bit signed variable located in the memory position addr.

H+! (n addr --)
Adds the number n to a 16 bit signed variable located in the memory position addr.

C@ (addr -- n)
Recalls an 8 bit signed number n from the memory position addr in the TOS.

C! (n addr --)
Stores the number n on a 8 bit signed variable located in the memory position addr.

C+! (n addr --)
Adds the number n to a 8 bit signed variable located in the memory position addr.

This is cumbersome compared to values.

CForth provides two intelligent store and recall words that are able to determine at runtime the correct storage size of variable:

V@	(addr -- n)	CForth
Recalls a number n from the variable position addr in the TOS.		

V!	(n addr --)	CForth
Stores the number n on a variable located in the memory position addr.		

V+!	(n addr --)	CForth
Adds the number n to a variable located in the memory position addr.		

The ***V@***, ***V!*** and ***V+!*** determine the size of the variable storage at run time that means that:

1) There is some impact on the time needed to execute those words compared to the other variable operation words.

2) The "V" variable operations can only be used on variables defined using the VARIABLE, HVARIABLE or CVARIABLE words.

The point 2) is quite a limitation because memory manipulation is the place where the use of variables shine. It is not a surprise as variables are, in fact, pointers. This concept deserve a dedicated sub section in this manual.

4.4. Using variables for memory manipulations

The main use of variables is where you would use pointers in other languages. There are two typical cases: arrays and memory manipulations. Arrays will be explained in the next sub section so we will talk only about memory manipulations here.

In a modern computer system controlled by an operating system that uses virtual memory there it is not easy to directly manage the memory locations. CForth however is targeted to medium size microcontrollers where you can usually access all the physical memory map.

If you know the physical address of one register and you want to modify its value you can do that using pointers.

Let's say that you use CForth inside the SF3 Gizmo that provides a CForth **port** for the STM32F303Vx microcontroller.
You know that the user led 0 is at PE9 (line 9 of the GPIO port E).

Port E start at address 0x48001000
The output data register (ODR) for each port start at an offset 0x14 from the port start.
You can set the bit 9 of the ODR register of GPIO port E using the following code:

```
0x48001000 CONSTANT gpioe   \Start of GPIOE
0x14 CONSTANT odr           \Offset of ODR register
gpioe odr +                 \ODR register of port E
DUP @                       \Get current ODR value
1 9 LSHIFT OR               \OR with bit 9
SWAP !                      \Store new ODR
```

As another example we can use variables to manipulate their contents byte by byte. Remember that we defined MYVAR as a 32 bit variable. We can change only its first byte to 1 (using the C! word) and see the effect on the variable:

```
> 1 MYVAR C! MYVAR @ . CR
1
```

As we can see our machine stores the Lower Significant Byte in the Lower position of the 4 Byte variable. So it is *little endian*. No surprise for an Intel powerered PC.

We can try to change the second byte adding 1 to the MYVAR pointer:

```
> 1 MYVAR 1+ C!    MYVAR @ . CR
257
```

This is the expected result as the second byte has a 256 weight and the first byte was 1.

You can use this method to store as much as 4 indexed Char numbers inside a 32 bit Char variable. The preferred method to do this kind of indexing is, however, the use of arrays that we will see later.

4.5. The Create Word

The ***create*** word is the most powerful word in a Forth system. Forth systems can use the create word to expand the compiler capabilities. In fact most Forth systems are written in Forth thanks to the Forth capability to create new compiler constructs.

This is not the case in CForth. CForth is not written in Forth but in “C”. The CForth create word has only a limited subset of the capabilities that feature on a compliant Forth system. In fact, this is the single most important difference that makes CForth non Forth compliant.

The reason why CForth don't implement all create capabilities is because it don't suit the CForth purpose. CForth don't pretend to be a Forth system. It only pretends to be an easy way to interact with microcontrollers.

In CForth ***CREATE*** can only be used to define static storage elements.

CREATE <word>
Creates a new word with the indicated name and with zero storage space.
Upon use of the new word the address of the first element in the storage area will be pushed on the stack.

The ***CREATE*** word is similar to the ***VARIABLE*** word. Both define a storage space and both push on the stack the start of the storage space when using the associated word.
The difference is that create don't reserve any space. You can think of create as a zero size variable definition.

In order to use a created word for something useful you need to increase its storage space. You can do that with the ***ALLOT*** word.

ALLOT `( u -- )`
Add u bytes of storage to the last defined word.

Using ***ALLOT*** you can create any size storage words. For instance:

```
> CREATE NAME 4 ALLOT
```

Is nearly equivalent to:

```
> VARIABLE NAME
```

Nearly but not the same. Not at least in CForth. The two most important differences are:

1) ***VARIABLES*** are set to zero upon creation while start values of allotted fields are undefined.

2) You cannot use the ***V@ V!*** and ***V+!*** on ***CREATE*** words.

To ease the calculation of storage needs MFort provides several words:

CELLS `( u -- u*cell_size)`
Multiplies the TOS by the cell size that in CForth is 4 bytes.

HCELLS `( u -- u*half_cell_size)` CForth
Multiplies the TOS by the half cell size that in CForth is 2 bytes.

CELL+ `( n -- n+cell_size )`
Adds the cell size (4 in CForth) to the TOS.

HCELL+ `( n -- n+cell_size )` CForth
Adds the half cell size (2 in CForth) to the TOS.

That way, for instance, you can reserve space for 5 32 bit numbers in a word named ***block*** using:

```
> CREATE block 5 CELLS ALLOT
```

To store the number 1000 inside the second element (first has index zero) of the newly created ***block*** storage you can use:

```
> 1000 block CELL+ !
```

You can also chain several ALLOT space reservations.
For instance:

```
> CREATE fields 2 CELLS ALLOT 4 HCELLS ALLOT
```

Assigns storage space into the ***fields*** word for 2 Cells followed by 4 Half Cells. And this is equivalent to:

```
> CREATE fields 2 CELLS 4 HCELLS + ALLOT
```

or

```
> CREATE fields 16 ALLOT
```

If you want to allocate space and set, at the same time, its start value you can use the following words:

, `( n -- )`
Allocates space for one Cell and initializes it to the n value popped from the stack.

H, `( n -- )`	CForth
Allocates space for one Half Cell and initializes it to the n value popped from the stack.	

C, `( n -- )`
Allocates space for one Char and initializes it to the n value popped from the stack.

To allocate in a word named ***fields2*** 3 Cell with start values 100, 101 and 102 you can use:

```
> CREATE fields2 100 , 101 , 102 ,
```

Observe that, as commas are words, you need to leave spaces after and before.

You can use ***ALLOT*** and the ***,*** ***H,*** and ***C,*** words to extend the storage space of the previously defined words. In the case of code programs it is not recommended as the assigned space won't be easy to be used. In the case of variables we can extend their storage space and convert then to something similar to a CREATE field.

For instance:

```
VARIABLE array     \Create a one Cell variable
 4 CELLS ALLOT     \Add space for 4 more Cells
```

4.6. The PAD

CForth includes a PAD zone that the user can use to store temporal data.
The size of this area is **port** dependent. You can see its size calling the ***LIMITS*** word. To obtain the address of this area you can use the PAD word.

PAD `( -- addr )`
Gives the address of the PAD

In the PAD area you can use the variables ***!*** ***H!*** ***C!*** and ***@*** ***H@*** ***C@*** words to store and get data.

In the following example the PAD is used to store and recall two 32bit numbers:

```
2076 PAD !
8901 PAD 1 CELLS + !
PAD @ . CR
PAD 1 CELLS + @ . CR

PAD @ . CR
2076

PAD 1 CELLS + @ . CR
8901
```

COMPATIBILITY
Forth systems also have a PAD area, but in ANS compliant systems the PAD area is used by the Forth engine so you cannot use it for long storage of data. In CForth the PAD is only used by the user or in **port** words. Refer to the **port** documentation to see if there are any words that make use of the PAD.

5. CREATING CODE WORDS

All the above explanations work well when doing interactive calculations on the console. To go beyond that, you need to be able to create new words with some associated code.

5.1. Basic word creation

All the examples shown before used CForth in **Interactive mode**. That is, all words are executed at the time they are introduced. CForth can also work in **Compile mode**. In this mode the introduced words are used to compile a program associated with a new word.
To create a new word that executes a program you can use the **:** and **;** words

: <word>
Enters compile mode.
Starts the definition of a new word <word>. Any word introduced after this one will be part of the code associated to the new word. This word clears the parameter and return stacks.

The compilation process uses the parameter and return stacks for internal use. That's why these stacks get cleared when using the **:** word.

;
Leaves compile mode and returns to interactive mode.
Ends the definition of the word that started with the : word.

The new created word can be uses as any built-in word.

For instance, to create a word that multiplies the TOS by 1000 you can use:

```
> : mult 1000 * ;

>7 mult
<1> Top: 7000
```

Some words can only be used in interactive mode, trying to compile one of these words yield a compile error:

```
> : test basewords ;
ERROR: Token >>>basewords<<< not recognized in compile mode
Compilation aborted at line 1
<Empty Stack>
```

If you use more than one line to compile a new word the prompt changes to `COMP>` for each line that starts in compilation mode.

It is habitual to show in the first line of a new word code a comment with its stack effects. The following example shows a two line word definition that includes a stack diagram comment:

```
> : mult2 ( n -- 100*n )
COMP> 100 * ;
```

5.2. Errors during compilation

If an error happens during the compilation, an abort compilation event takes place and CForth discards the rest of the new word code until a end of compilation ***;*** word is found.
If a line ends before finding the ***;*** word, the prompt changes to `CERR>` to indicate that there was a compilation error. To get out of this state you just need to enter the ***;*** word.

For example:

```
> : err_test
COMP> 1 2 not_exits 3

ERROR: Token >>>not_exits<<< not recognized in compile mode
Compilation aborted at line 2
<Empty Stack>

CERR> ;
<Empty Stack>
>
```

The error message indicates the line of code where the error takes place. This line is counted from the line that contains the ***:*** word.

A word is not registered in the user dictionary until its definition has ended successfully using the ***;*** word. An aborted word compilation leaves no trace in the user dictionary.

For the rest of the example programs in this manual we will omit the prompt to simplify the listings and ease cutting and pasting the code on the console.

5.3. Forgeting words

The new created words, data and code, are added to the user dictionary in a stack fashion. That means that if you want to remove a word from the dictionary you need to remove also any word that you are defined before.

FORGET <word> NC
Removes the word <word> from the dictionary.
Removes also any word that was introduced after this one.

For instance, you can define three values and a code word that show their contents:

```
10 VALUE value1
20 VALUE value2
30 VALUE value3
: show value1 . value2 . value3 . CR ;
show
10 20 30
```

If you forget *value2* you forget also *value3* and *show* that were defined after it.

```
FORGET value2
show
ERROR: Token >>>show<<< not recognized in interactive mode
```

To forget all user defined words you can use the ***FORGETALL*** word.

FORGETALL CForth NC
Forgets all user words clearing all user dictionary data.

5.4. Flow control words

This section deals with flow control words. These words can only be used inside new word definitions (compile mode) and give error if used in interactive mode.

5.4.1. IF ELSE ENDIF

These three words define the basic conditional program execution.

```
IF
  words if true...
ELSE
  words if false...
ENDIF
```

IF `( f -- )` NI
Pops a flag from the stack. If it is true continues the execution after this command. If it is false jumps to the code after ***ELSE*** if it is defined or ***ENDIF*** it no ***ELSE*** is present.

ELSE NI

Defines the start of the code that is executed when the flag evaluated in the associated ***IF*** is false.

ENDIF CForth NI

Defines the end of the IF ELSE ENDIF conditional.

COMPATIBILITY

ANS Forth compliant systems don't use the ENDIF word but a THEN word. In CForth you can use both ENDIF and THEN for the same functionality.

The following word definition sorts the TOS and NOS so that the NOS is the smaller of both:

```
: SORT2 ( n1 n2 -- n1 n2 | n2 n1 )
2 DUPN > IF
SWAP ENDIF ;

 8 6 SORT2 .S CR
 <2> 6 8
```

5.4.2. DO Loops

DO loops are the counted type of loops in CForth. In their basic mode the words DO and LOOP are used.

```
DO
  ...words...
LOOP
```

DO `( limit index -- ) R( -- limit index )` NI

Starts a do loop. Pops two values from the parameter stack and pushes the on the return stack. The TOS is the first value of the loop index and the NOS is the limit that ends the loop operation.

LOOP `( -- ) R(` *too complex to show* `)` NI

End of a DO loop. The index on the top of the return stack is incremented by 1. If the index after the increment is below or at the limit the execution jumps to the start of the loop. Otherwise the limit and the index are eliminated from the return stack and the execution continues after the LOOP word.

You can nest several DO loops one inside another. You can use the ***I*** , ***J*** and ***K*** words to get the index of up to three nested DO loops.

I `( -- n ) R( n -- n )`
Gets index from first do loop

J `( -- n ) R( n a b - n a b )`
Gets index from second do loop

K `( -- n5 ) R( n5..n1 -- n5..n1 )` CForth
Gets index from third do loop

The following example writes the numbers from 0 to 10

```
: loop_test  \Show numbers from 0 to 10
11 0 DO      \Start of loop with limit 11 and index 0
     i .     \Show index
  LOOP ;     \Increase index and repeat loop if less than limit

loop_test
0 1 2 3 4 5 6 7 8 9 10 <Empty Stack>
```

The basic ***DO LOOP*** pair or words always increment the index by one on each loop iteration. In order to use a different increment the ***@LOOP*** can be used.

@LOOP `( n -- ) R(` *too complex to show* `)` CForth NI
End of a DO loop. The index on the top of the return stack is incremented by n. If the index after the increment arrives to the limit or crosses it the execution jumps to the start of the loop. Otherwise the limit and the index are eliminated from the return stack and the execution continues after the @LOOP word.

The following example writes the numbers from 10 to 0 backwards:

```
: loop_test2 -1 10 DO i . -1 @LOOP CR ;
loop_test2
10 9 8 7 6 5 4 3 2 1 0
```

The ***DO...LOOP*** and ***DO...@LOOP*** loops always are executed at least once regardless of the values of the limit and index values.
The ***+DO*** and ***-DO*** words have the same functionality that the ***DO*** word but check the index to the limit before executing the first loop.

+DO `( limit index -- ) R( -- limit index )` NI
Starts a do loop. Skips the loop if index > limit.
This word is designed to be used with positive increment loops.

-DO `( limit index -- ) R( -- limit index )` NI
Starts a do loop. Skips the loop if index < limit.
This word is designed to be used with negative increment loops.

You can leave a DO loop before it ends using the ***LEAVE*** or ***?LEAVE*** words.

LEAVE NI
Leaves the current loop. In the case of DO loops removes the limit and index in the process.

?LEAVE `( f -- )` NI
Leaves the current loop if f is true. In the case of DO loops removes the limit and index in the process.

COMPATIBILITY
Standard CForth systems don't feature a @LOOP word. They use a +LOOP word instead. This word is not used in CForth because it has a non symmetrical operation with negative increments compared with positive increments.
+DO and -DO are not standard also. ANS Forth systems provide the ?DO word, but this only checks for index equal to limit not for going beyond it.

5.4.3. BEGIN...UNTIL loops

This kind of loop executes always once and, at the end, verifies if it needs to be executed again.

```
BEGIN
   ...words...
UNTIL
```

BEGIN NI
Start of uncounted loop.

UNTIL `( f -- )` NI

Pops a flag f from the stack. If it is false jumps to the start of the loop after the ***BEGIN*** word. If it is true exits the loop executing the code after this word.

For example you can use this kind of loop to show the numbers from 0 to 10

```
: until_test ( -- )      \Show numbers from 0 to 10
0 BEGIN                  \Start of loop with 0 TOS
DUP . 1+                 \Show TOS and add 1
DUP 10 > UNTIL DROP ;    \Repeat until TOS > 10 and drop TOS
<Empty Stack>

until_test
0 1 2 3 4 5 6 7 8 9 10 <Empty Stack>
```

You can use the ***LEAVE*** and ***?LEAVE*** words to leave the loop as in the case of DO loops.

5.4.4. BEGIN...WHILE...REPEAT loops

This kind of loop is the most powerful non counted loop in CForth.

```
BEGIN
  ..test words..
WHILE
  ..execute words..
REPEAT
```

BEGIN NI

Start of uncounted loop.
Words from this point form the test word group that should leave a flag on the stack.

WHILE `( f -- )` NI

This word pops a flag from the stack. If it is true the program continues its execution in the execute words that follow the ***WHILE*** word. If it is false the loop ends and the execution continues after the ***REPEAT*** word.

REPEAT NI

This word marks the limit of the execution words. The execution branches to the code after the ***BEGIN*** word.

You can use ***LEAVE*** and ***?LEAVE*** to exit the loop in the same way that the previously defined loops.

Example that calculates squares below 1000:

```
: squares                \Display squares lower that 1000
0                        \Start with index 0 value
BEGIN
   1+ DUP DUP *           \Add 1 to index and calculate square
   DUP 1000 <             \Compare square with 1000
WHILE
   .                      \Show square
REPEAT
DROP DROP CR ;           \Drop index and last square
```

5.4.5) BEGIN...REPEAT loops

As the ***WHILE*** word in the BEGIN...WHILE...REPEAT is the way out ot the loop, a loop that only uses ***BEGIN*** and ***REPEAT*** is never-ending as it has no way out.
You can always exit the loop with the ***LEAVE*** and ***?LEAVE*** words, however.

Example that calculates powers of 2 below 1000:

```
: 2powers                \Display powers of 2 lower that 1000
1                        \Start with index 1 value
BEGIN
  2*                     \Duplicate
  DUP 1000 >             \Compare with 1000
  IF DROP LEAVE ENDIF    \Leave if greater
  DUP .                  \Show
REPEAT
CR ;                     \New line
```

COMPATIBILITY
Standard forth neverending loops are defined with the ***BEGIN*** and ***AGAIN*** words. In CForth the ***AGAIN*** word is provided for compatibility but its effects are equivalent to the ones of using ***REPEAT***. So for the sake of simplicity, the ***AGAIN*** word is not used in the examples.

5.4.6. CASE...OF...ENDOF...ENDCASE

This flow control construct is the most complex in CForth. Its aim is to substitute big series of nested IF ELSE ENDIF constructs.

```
CASE
n1 OF ..n1 words.. ENDOF
.....
nu OF ..nu words.. ENDOF
.. default words..
ENDCASE
```

CASE `( ncmp -- ncmp )` NI

Starts the construct. You are expected to leave one number ncmp on the stack before executing the CASE word although it is not enforced.

OF `( ncmp n -- ncmp | ncmp n -- )` NI

Before executing this word there should be a number ncmp at the NOS position and a number n at the TOS position.

The ***OF*** word pops the n number and compares it with the ncmp number that is not popped from the stack.

If both numbers are equal ncmp is also dropped and execution continues after this OF word. Otherwise the execution branches after the following ***ENDOF*** word leaving the TOS ncmp value.

ENDOF NI

When this word is executed the execution jumps after the ***ENDCASE*** word.

ENDCASE `( ncmp -- )` NI

This word marks the end of the construct. The TOS ncmp is dropped and execution continues after the word.

You can introduce default words that don't match any case between the final ***ENDOF*** and the ***ENDCASE*** word. Beware however that the TOS is ncmp when entering this region and that the ***ENDCASE*** will always drop the TOS.

You cannot use ***LEAVE*** or ***?LEAVE*** inside ***CASE*** or ***IF*** constructs.

Example that carries one of the four basic operations depending on the TOS value:

```
: OP ( n1 n2 operation -- result )
CASE
  1 OF + ENDOF
  2 OF - ENDOF
  3 OF * ENDOF
  4 OF / ENDOF
  ABORT
ENDCASE ;

2 2 1 op
<1> Top: 4

2 4 3 op
<2> Top: 8
```

5.5 Special flow control words

The above section constructs define structured flow control operations. As we have seen CForth provides the non structured control words ***LEAVE*** and ***?LEAVE*** that enable us to get out of the loops.
CForth provides two additional words that enable us to exit from the current word or from all words currently in execution.

EXIT NI
Leaves the current word.

In Forth you should be very careful when exiting a word from inside a counted loop. As the limit and index for the loop reside on the control stack very bad things can happen if you don't eliminate these elements.

UNLOOP `R( limit index -- )`
Drops the limit and index from the return stack. In normal Forth systems you should call this word for every nested counted loop before calling EXIT.

In CForth you don't need to call ***UNLOOP*** as when leaving a word the return stack pointer is automatically restored to the value that it had upon entering.

ABORT NI

Exits the current word and any other word called by it. The abort process ends when the execution returns to the console prompt in interactive mode.
During the abort process a backtrace with the name of all the words that exit on the way out to the console is shown.

Backtracking example in a three level nested word:

```
: word1 ABORT ;
: word2 word1 ;
: word3 word2 ;

>word3
RUN ERROR: ABORT executed
Backtrace: 8 >> WORD1 <<
Backtrace: 18 >> WORD2 <<
Backtrace: 30 >> WORD3 <<
<Empty Stack>
```

The abort process is automatically called when a run error, like stack underflow, is produced during execution.

CForth ports can provide other ways to start the abort process.

5.6 Strings

CForth is not designed for string manipulations so only very basic string support is provided.

5.6.1 Print strings

Print strings are defined with a **."** point-double quote two char sequence and end with a double quote "

When CForth finds a print string it prints it on the screen.

```
> ."This is a string"
This is a string<Empty Stack>
```

In interactive mode there is little difference between using ***."*** and ***.(***

```
> .( This is a comment)
This is a comment<Empty Stack>
```

The main difference, a part to the fact that ***.(*** is a word and ***."*** is not, is that printing comments cannot be compiled inside a new word definition as they are always executed immediately.

5.6.2. Counted strings

The standard type of strings in CForth are counted strings. A counted string is stored with one byte that indicates the string length followed by one byte for each string character.
The one byte storage limit of counted strings limits its size to 255 characters.

S"*String contents****"*** `( addr u -- )` NI
Codes a counted string inside a word definition.
In the current CForth version it cannot be used in interactive mode.
In CForth the start ***S*** character can be omitted so ***"****String contents****"*** is also a valid string definition.
On execution the string leaves on the NOS the address of the start of the string (following the count byte) and in the TOS the string size.
The address in NOS is a CPU address so it is compatible with ***C@*** and ***C!*** operations.

For example:

```
: stest S"My string" ;
stest .S CR
<2> 4305966 9
```

The TOS 9 value is the length of the string. The NOS holds the address of the first letter 'M' of the string. You can use the ***TYPE*** word to print the string contents on the console.

TYPE `( addr u -- )`
Write on the console u characters starting at the address addr.
Notice that u is not limited to the 255 characters of a counted string.

If we continue from the previous example:

```
TYPE
My string<Empty Stack>
```

If your address points to the count byte you can use the ***COUNT*** word to get the parameters needed by TYPE.

COUNT (addr -- addr+1 u)
Takes the count value of a counted string from the address address addr given. Returns the next address addr+1 and the count value u.

We can use count in the address just one byte before the one returned by the string.

```
4305965 COUNT .S  CR
<2> 4305966 9
```

Observe that in CForth strings can only be included in code words, but their contents can be accessed in interactive mode.
If we have the address of a counted string we can use ***COUNT*** followed by ***TYPE*** or we can use the ***STYPE*** word that has the same effect:

STYPE `( addr -- )`
Prints a counted string on the console given the address of the count byte that precedes the first string character.
Is equivalente to ***COUNT*** followed by ***TYPE***.

COMPATIBILITY
In ANS compliant Forth systems ***."*** and ***S"*** are words, not a string start sequences. That means that in Forth you should leave at least one space between the word and the start of the string like in ***.(***
CForth don't impose such requirement. This difference makes easier to declare strings in CForth that includes spaces at the start.

5.6.3. C like strings

CForth supports null terminated strings like the ones usually found programming in "C" language. There is no specific word to create space for this kind of strings as CREATE is enough.
In the following example we get space for 20 characters:

```
CREATE cstring 20 ALLOT
```

We can put a two char **"CS"** content using this code:

```
cstring
'C' OVER C!       \First char
'S' OVER 1+ C!    \Second char
 0  OVER 2+ C!    \Null terminator
```

To write the contents of the C like string we can use the ***CTYPE*** word:

CTYPE	`( addr -- )`	CForth
Print a zero terminated string given its first character.		

We can use ***CTYPE*** in the previously created string:

```
cstring CTYPE
CS<1> Top: 4305967
```

5.6. Local values

Local values are normally called local variables in Forth jargon. We will call them local values in this document because they operate in all ways like values not like variables.

Local values feature automatic storage on the return stack. All local values have 32 bit **Cell** size. When you exit a word you lose all local values information.

Forth programmers often use the return stack to hold temporal data. Local variables just makes the use of the return stack space easier.

There are three words ***{*** , **--** and ***}*** used to define local variables. The format to define a set of local values is:

{ val1...valu **--** *comment* ***}***

The above definition creates u local values with names val1 to valu. The start value of each local value is popped from the parameter stack in reverse order from valu to val1.
Any text between the **--** word and the ***}*** terminator will be treated as a comment.

The similarity of the local values definition with the stack diagram is intentional. You can use the locals definition at the start of a word to pop its parameters and give a comment about its return values.

The following example shows a word definition that uses locals to calculate a linear equation:

```
: lin { m n x -- m*x+n }
m x * n + ;
```

Using locals is so easy that you can forget that the same operations can be carried out in a more efficient manner using just the stack manipulation words:

```
: lin ( m n x -- m*x+n )
ROT * + ;
```

Observe that in this case the stack diagram is a comment, not a locals definition.

You can use several locals definitions inside a word but they cannot be located inside loops or conditionals.

```
: func { x -- 4x^2+5x+6 }
x DUP * { xx }   \Obtain and store square
xx 4 *    \First term
x 5 * +   \Second term
6 + ;     \Third term
```

It should be evident that the xx local is not really needed but it is OK for an example.

5.7. Structures

CForth don't provide all the powerful capabilities that enable Forth to define new defining words. But you have enough capabilities to create structure defining words.

Supose that we want to define a datatype that contains day, month and year in this order. We want to allocate a **Char** for day and month and a **Half Word** for year.
You can create a *date1* storage word for all these values for the 10/3/2013 date using:

```
CREATE date1 10 C, 3 C, 2013 H,
```

You can declare words to access the date elements and print its contents.

```
: GET_DAY C@ ;
: SET_DAY C! ;
: GET_MONTH 1+ C@ ;
: SET_MONTH 1+ C! ;
: GET_YEAR 1+ 1+ H@ ;
: SET_YEAR 1+ 1+ H! ;
: .DATE DUP C@ . BS '/' EMIT
        1+ DUP C@ . BS '/' EMIT
        1+ H@ . ;

> date1 .DATE CR
10/3/2013

> 4 date1 SET_MONTH
> date1 .DATE CR
10/4/2013
```

If you want to use several data elements to hold dates, you can repeat the create line for each one of them. This has the drawback that you need to remember the internal structure associated to a date element.

To simplify the creation of new date elements you can create a defining word that creates the date structure programmatically and sets its start value from stack data.
The following example defines a new *CDATE* word that creates a date structure and assigns its values from stack data:

```
\ CDATE creates a date object ( day month year -- )
: CDATE CREATE ROT C, SWAP C, H, ;
```

For non Forth programmers the above code can be puzzling. To understand it you should remember that ***CREATE*** always takes its word name argument from the console at run time, not at compile time.

To create a new date object you just need to use the newly defined word:

```
 11 7 2010 CDATE date2

 > date2 .DATE CR
 11/7/2010
```

When *CDATE* is executed, it calls the ***CREATE*** word. This word takes the *date2* word name from the console to create a new word definition.

COMPATIBILITY
ANS Forth compliant systems not only can implement new data defining words but can also create new code compiling words. That goes beyond the intent and capabilities of CForth.

5.8. Special coding words

You can create most code and data words from the elements describen above. If you are new in Forth you can opt to leave this section alone for a while.
This section describes words that are useful only for some special cases.

5.8.1. Word positions inside the dictionary

' <word> `( -- Uaddr )`
Returns the address of a word from the start of the User Dictionary.
In CForth **'** only works with user defined words no with the base ones.

For instance, you can start CForth and write the following orders:

```
> : test 1 2 3 ;
> ' test
  <1> Top: 7
```

CForth tells you that the first user word you have defined starts at position 7 of the User Dictionary. It is not zero because **'** returns the start of the data region of the word. Before it there is a header that holds:

- Name of the word (4 Bytes for “test" in the example)
- Lenght of the word name (1 Byte)
- Position of next word in dictionary (2 Bytes)

That adds to the 7 number that **'** returns.
If you know where a word starts you can execute from there using the ***EXECUTE*** word.

EXECUTE `( Uaddr -- )`
Executes from the position (relative to the User Diccionary start address) indicated by the TOS.

Remember that we have the 7 returned by **'** in the TOS. We can the call execute and show the stack contents:

```
> EXECUTE .S
<3> 1 2 3
```

So we have indirectly executed the *test* word.
We can use values or variables to hold pointers to User Dictionary words.
The User Dictionary in CForth can't go beyond 64 Kb as it uses 16 bit addresses. That means that you can use **Half Cell** storage to hold address values.
By default **Half Cell** values are signed so you will store negative numbers beyond the 32 KByte limit. ***EXECUTE*** automatically cast the provided value to an unsigned 16 bit number so you don't need to manually convert to unsigned.

In the example below we store one operator in a variable and use it afterwards to carry the operation on the TOS and NOS. We need to encapsulte the operators inside user functions because the current implementation of ' in CForth don't work with non User words.

```
: add + ;
: sub - ;
: mult * ;
: div / ;
' mult HVALUE operator

> 4 6 operator EXECUTE
<1> Top: 24
```

The word ***HERE*** can be used to know the next Byte position of the user dictionary that will be allocated.

HERE `( -- Uaddr )`
Returns the address of the current User Dictionary pointer relative to the start of this dictionary.

Let's use this word:

```
FORGETALL  \Erase User Dictionary
: test ;   \Create a dummy empty word

> ' test
<1> Top: 7
> HERE
<2> Top: 8
```

We see that the code of the *test* word starts at position 7. As it is an empty word it only contains the end of word marker that has one byte size. That's why ***HERE*** shows that the next position to allocate is 8.

5.8.2. Going out of compile mode with [ ]

Sometimes you need to execute a word in interactive mode while you are compiling another. In this cases we use the ***[*** and ***]*** words.

[NI
Enters interactive mode
Can only be called from compile mode

] NC
Enters compile mode
Can only be called from interactive mode entered with a previous [word.

During compilation the stack is used to store the state of the program flow constructs. You should not push nor pop stack elements between program flow constructs like DO, WHILE... You should also pop any data you push on the stack during compilation.

Try the following example:

```
: test2 1 2 [ 3 ] 3 ;
Compilation aborted at line 1
ERROR: Branch inconsistency in word
```

It gives error because it don't expect to find any data on the stack when the word compilation ends at the ***;*** word. Observe that the following case gives the same error:

```
: 10 0 DO i . ;  \We forget the LOOP word
Compilation aborted at line 1
ERROR: Branch inconsistency in word
```

You also cannot define new words in a ***[ ]*** zone as the User Dictionary is being written as you must end the compiling of a word the start coding another.

```
: test [ 9 VALUE val ] 1 ;
Compilation aborted at line 1
ERROR: Already defining a word
```

We have seen several things that cannot be done using ***[ ]***. Let's see one thing that can be done:

```
forgetall
0 VALUE val
: test 1 . 2 . [ HERE TO val ] 3 . 4 . ;

> test
1 2 3 4 <Empty Stack>

> val EXECUTE
3 4 <Empty Stack>
```

We have started the execution from halfway inside the *test* word. This is not something you should normally do.
From here you can develop very obscure constructs. Try to think about what the following code should do and then try it in CForth.

```
0 VALUE val2
: test2 1 . 2 . val2 EXECUTE [ HERE TO val2 ] 3 . 4 . ;
test2
```

5.8.3. The LITERAL word

The ***LITERAL*** word enables you to code a number inside your word using a calculation made in a interactive mode region [ ].

LITERAL `( n -- )` NI
Pops the TOS from the stack and codes its number in the currently edited word. So that it is pushed in the stack at run time.

In the following example ***LITERAL*** is used to statically compile the compilation date.
It uses the date words we defined previously.

```
10 4 2014 CDATE date

: SHOWCDATE [ date GET_DAY ]   LITERAL . BS '/' EMIT
            [ date GET_MONTH ] LITERAL . BS '/' EMIT
            [ date GET_YEAR ]  LITERAL . CR ;

> SHOWCDATE
10/4/2014
```

As the date values are statically coded inside the *SWOWCDATE*, if we change the date infomation the *SHOWCDATE* information will not change.

```
> 2016 date SET_YEAR
> SHOWCDATE
10/4/2014
```

5.8.4. Word recursion

When you are creating a new word, CForth hides its name so that you cannot call the word itself. If you want to call a word from itself you need to use the ***RECURSE*** word.

RECURSE
Calls the currently compiled word.

The following example shows one example for a factorial calculation:

```
\Factorial calculation
: factorial ( n -- n! )
DUP
1 > IF
 DUP 1- RECURSE *
ENDIF
;

> 6 factorial
<1> Top: 720
```

For each recursive function there is usually a non recursive function that uses less system resources. The capability to do deep recursions in CForth depends on its system stack size. As CForth is targeted at medium size microcontrollers that have limited stack space you are advised to limit the use of recursion.

5.8.5. Word location in CPU memory space

If you remember, words created by ***VARIABLE***, ***HVARIABLE***, ***CVARIABLE*** or ***CREATE*** leave on the stack the address of the start of their storage space when they are called.
This address is a 32 bit location inside the memory map as seen by the CPU.

Word position words like ***'*** and ***HERE*** give a 16 bit location relative to the start of the User Dictionary. You can convert between User Dictionary relative addresses to CPU absolute addresses using the following words:

MEM2USER `( addr -- Uaddr )` CForth
Converts the TOS 32 bit CPU absolute address to a 16 bit relative User Ditionary address.

USER2MEM `( Uaddr -- addr )` CForth
Converts the TOS 16 bit User Ditionary address to a 32 bit absolute CPU address.

Every element constructed with ***VARIABLE***, ***HVARIABLE*** and ***CREATE*** is stored in the User Dictionary. You can always convert its CPU address to a User Dictionary relative address that needs only half storage space.

The reason why variables and created elements return absolute CPU addresses is because this way the *@* and ***!*** operators have a consistent behavior when they are used as pointers to variables or are used as pointers to known memory locations outside the User Dictionary.

Let's try this example:

```
FORGETALL
VARIABLE var
var
<1> Top: 4305963
MEM2USER
<1> Top: 7
```

We see that the 4 bytes of storage of the *var* variable start at position 7 from the User Dictionary start position.
Let's see what the ' word gives about the variable "var":

```
>' var
<2> Top: 6
```

The data start position of variable *var* is not 7 but 6. The byte located at the position 6 is a tag that instructs the execution unit in CForth that this location holds a 32 bit variable and that it should give the CPU address of the next position.

This example is more interesting:

```
: four 4 . ;
four
4 <Empty Stack>

' four USER2MEM 1+ 10 SWAP C!
four
10 <Empty Stack>
```

What happened?
The start location of the *four* word holds the number 4. It is coded with two bytes: a tag byte that indicates that an 8 bit signed number follows and the 8 bit signed number itself. We have obtained the CPU address of the byte that coded the 4 number and we have changed it to 10.
Observe that the SWAP is needed because ***C!*** expects the address to be at the TOS and the number to store to be at the NOS.

6. THREADS

CForth supports multithreaded execution.
The thread capabilities are not provided by CForth itself. The must be implemented as part of the PORT were CForth is integrated.

6.1. Thread basics

Each thread in CForth runs in an independent execution context. Each execution context includes:

- A parameter stack
- A return stack
- A system stack
- A program counter
- A verbose level variable

When CForth starts the only thread available is the main thread that runs in the main context. The forth interactive mode runs in this thread. This is the only thread that can get data from the console so we call it also the foreground thread. Any other thread will be a background thread.

All background processes start isolated from the console. Not only they don't receive data from the console, they also cannot write anything on the console because they start with a verbose variable set at zero. In order to write on the console a thread must change is verbose level using the ***VERBOSE*** word.

CForth uses static storage for the thread data. The maximum number of background threads is defined when CForth is compiled and cannot be changed afterwards. Use the ***LIMITS*** word to know which is the thread number limit.

6.2. Creating threads

When we start a new thread the new context return stack is cleared and the parameter stack is cloned from the one in the parent context. That way you can send information to the new thread.

THREAD <word> `( -- nth )` CForth
Executes the indicated word in a new thread.
Returns the newly created thread number nth or zero if the thread could not be created.

The ***THREAD*** word creates a thread with the same priority than the main thread. To create a thread with a different priority you can use the ***THPRIO*** word.

THPRIO <word> `( np -- nth )` CForth
Executes the indicated word in a new thread.
Returns the newly created thread number nth or zero if the thread could not be created.
The new tread gets a relative priority obtained from the element at the TOS:

0 : Same priority as the main thread
Negative: Less priority than the main thread
Positive: More priority than the main thread

The created thread runs until it returns from the starting word or aborts calling ***ABORT***, due to a run error or by external command.

You can see the list of running threads using the ***TLIST*** word.

TLIST CForth
Lists the currently active threads with their thread number and priority.
It also shows its status that can be ***running*** or ***aborting***.

6.3. Terminating threads

A thread terminates normally when its associated word ends.
You can also force the abortion of a thread using the ***TKILL*** word.

TKILL `( nth -- )` CForth
Forces the abortion of the thread numbrer nth.

When you abort a thread its termination is not instantaneous. An ***abort*** flag is set for the thread context and all nested words return one after another until the main word that started the thread returns. Currently executed built-in words cannot abort and must terminate normally. That means that the aborting process cannot start until the currently executed built-in word terminates.

To force the abortion of all running threads you can use the ***TKILLALL*** word.

TKILLALL CForth
Forces the abortion of all running threads

7 DEBUGING

CForth provides several words to check the status of the system and viewing the engine internals. It also provides some words to assert the correct operation of the developed code.

7.1. Showing system status

The following words show information about the CForth system in general.

SHOWFLAGS
Show information about several CForth general flags that can be active (TRUE) or inactive (FALSE). Shows also some CForth capabilities determined at compilation time.

LIMITS
Show different numeric limits. They are version and **port** dependent.

7.2. Viewing the User Dictionary

This section describes the words we can use to obtain information about the user dictionary contents.

UNUSED `( -- n )`
Show the free memory, in bytes, on the User Dictionary.

UDATA
Shows information about the current status of the User Dictionary.

UWORDS
Shows all user defined words in several columns in order from newer to older.

For instance:

```
UWORDS
User dictionary:
  YEAR        HVAR        DUMMY         WORD1
  FIEL        VAL1        VAR1
```

ULIST
Shows a list of all user defied word with one word in each line. Gives some information about each word. It also detects, for variables and values, if its storage space has be extended using user memory allocation words like ***ALLOT* , *H,*** or ***C,*** .

For instance:

```
ULIST
User dictionary listing:
  YEAR = 2014 (16bit Constant)
  HVAR = 0 (16bit Variable) Allocates 6 bytes!!
  DUMMY : Null Program
  WORD1 : Program word of 5 bytes
  FIEL : Create word of 16 bytes
  VAL1 = 10 (32bit Value)
  VAR1 = 0 (32bit Variable)
```

Observe that ***ULIST*** indicates that the variable *HVAR* that should only allocate 2 bytes really allocates 6 bytes. That means that its storage space has been extended.

You can see the contents of a code word using the ***SEE*** word.

SEE <word>
Shows the code associated to a user word.
If it is a variable, value or create storage it only explains the kind of word you are referencing.

Let's see how see shows the internals of a simple word:

```
: test1 5 * 4 + ; ( n -- 5*n+4 )
SEE test1
Decoding of word test1
      8 : 5
     10 : *
     11 : 4
     13 : +
     14 : ENDWORD
```

We see that the word code starts at position 8 and ends at position 14. The code is quite similar to the source. The comments, as expected, are not included in the compiled word. Observe that the numbers are coded with 2 bytes and the operators with one.

Another example:

```
2014 CONSTANT year
SEE year
Decoding of word year
     22 : 2014
     25 : ENDWORD
```

MFoth codes the constant values as programs that only include a number.

When we have flow control words things start to get complicated. The next example shows the factorial example as seen by the ***SEE*** command:

```
\Factorial calculation
: factorial ( n -- n! )
DUP
1 > IF
 DUP 1- RECURSE *
ENDIF
;
SEE factorial
Decoding of word factorial
     38 : DUP
     39 : 1
```

```
          41 : >
          42 : JZ 51
          45 : DUP
          46 : 1-
          47 : FACTORIAL
          50 : *
          51 : ENDWORD
```

Most of the code is the same that in the source. The ***RECURSE*** word has been substituted by a call of the word itself. The most important difference is the IF that has been substituted by a JZ (Jump if zero) word.
The JZ word is a hidden word of the base dictionary. You cannot compile this word directly. It can only be included in the program by the compiler during the parsing of the control flow structures.

As a final example we will see the case example we have seen before:

```
: OP ( n1 n2 operation -- result )
CASE
 1 OF + ENDOF
 2 OF - ENDOF
 3 OF * ENDOF
 4 OF / ENDOF
 ABORT
ENDCASE ;
SEE OP
Decoding of word OP
     89 : 1
     91 : OF 98
     94 : +
     95 : JMP 127
     98 : 2
    100 : OF 107
    103 : -
    104 : JMP 127
    107 : 3
    109 : OF 116
    112 : *
    113 : JMP 127
    116 : 4
    118 : OF 125
    121 : /
    122 : JMP 127
    125 : ABORT
    126 : DROP
    127 : ENDWORD
```

See how the case construct has been converted in unconditional jumps (JMP) and OF words. Those OF words are not the same as the ones in the source code as they include absolute branch addresses.
Observe that a ***DROP*** has been added at the end of the program. That is the compilation code that leaves ***ENDCASE*** to drop the number that is compared against all cases.

You can use the ***SEE*** word to understand the word operation, but you cannot recover the source code from the compiled one. In fact, the code generated by ***SEE*** cannot be recompiled again.
We will explain later in this manual the ***DECOMPILE*** word that can generate a decompiled source from the code of a word.

The ***SEE*** word enables us to see the code associated to a word. To obtain more low level information about a word we can use the ***UWDUMP*** word.

UWDUMP <word> CForth
User Word memory Dump.
Shows in the console the memory contents associated to a user word.

For instance, remember the year constant:

```
2014 CONSTANT year
UWDUMP year
Word position : 15
Word size : 11
       15 :  89  69  65  82   4   8   0   8 | YEAR....
       23 : 222   7   0                     | ...
```

We see that the word *year* starts at position 15 and ends at position 25 totaling 11 bytes. The first 4 bytes are the word name. The following byte is the number of word name letters (4). Following is the two byte address of the next older word in the dictionary (8 0). As the example system is low endian that means that this position is 8. Then comes another 8 number that is a tag indicating that a 2 byte number follows (222 7) this number is 256*7+222=2014. The final 0 marks the end of the word.

Of course you don't normally need to know all of this data. I know because I have coded CForth ;-)

The above code use decimal numbers. You can select between decimal and hexadecimal numbers using the ***DB_DEC*** and ***DB_HEX*** words.

DB_DEC CForth
Request that debug dumps are written in decimal

DB_HEX CForth
Request that debug dumps are written in hexadecimal

```
DB_HEX
UWDUMP year
Word position : 15
Word size : 11
       fh : 59 45 41 52  4  8  0  8 | YEAR....
       17h : de  7  0               | ...
```

UWDUMP shows one word contents. If you want to see a range of the user dictionary that is not exactly one word you can use the **DUMP** word.

DUMP `( Ustart size -- )` CForth
Dumps the user dictionary from the Ustart position during size bytes.

An example for the 50 first bytes of user memory:

```
0 50 DUMP
        0h : 54 45 53 54 31  5 ff ff | TEST1...
        8h :  7  5 3a  7  4 38  0 59 | ..:..8.Y
       10h : 45 41 52  4  8  0  8 de | EAR.....
       18h :  7  0 46 41 43 54 4f 52 | ..FACTOR
       20h : 49 41 4c  9 16  0 2b  7 | IAL...+.
       28h :  1 44 1a 33  0 2b 53  c | .D.3.+S.
       30h : 26  0                   | &.
```

7.3. Viewing the CPU memory

The previous dump words show the contents of the user dictionary. To see the contents of one arbitrary memory range as seen from the CPU you can use the **MEMDUMP** word.

MEMDUMP `( addr size -- )`
Dumps the memory as seen from the CPU from addr position during size bytes.

As an example the following instruction dumps the 50 bytes of memory just before the User Dictionary.

```
USER2MEM 50 - 50 MEMDUMP
  41b3f2h :  0  0  0  0  0  0  0  0 | ........
  41b3fah :  0  0  0  0  0  0  0  0 | ........
  41b402h :  0  0  0  0  0  0  0  0 | ........
  41b40ah :  0  0  9  0  0  0  0  0 | ........
  41b412h :  0  0 98 32 f0  0 87  0 | ...2....
  41b41ah : 8c  0 ff ff  0  0  0  0 | ........
  41b422h :  0  0                   | ..
```

7.4. Debug zones and Assertions

Debug zones are code regions that can be conditionally compiled. They are coded between the markers ***DEBUG(*** and ***)***

DEBUG (word1...wordn) CForth

Codes a sequence of words word1...wordn if debug is enabled with DEBUG-ON
If debug is not enabled the words are not coded.

DEBUG-ON

Activates debug and assertions. Debug zones and assertions will be coded in any new compiled word.

DEBUG-OFF

Deactivates debug and assertions. Debug zones and assertions won't be coded in any new compiled word.

Example of using debug with debug activated. The debug code is compiled:

```
\Debug Example
DEBUG-ON
: dbtest
1 2
DEBUG( ."DEBUG: Top of stack: " DUP . CR )
3 4
;
dbtest
DEBUG: Top of stack: 2
<4> Top: 4
```

Example when debug is not active. The debug code is not compiled in the word:

```
DEBUG-OFF
: dbtest
1 2
DEBUG( ."DEBUG: Top of stack: " DUP . CR )
3 4
;
dbtest
<4> Top: 4
```

You can use ***SEE*** to verify that the code is not present.
Using the word ***SHOWFLAGS*** you can check if debug/assert is active.

Debug zones is adequate if you want to interpret yourself the messages generated. A more efficient way to detect coding errors is using assertions.

Assertions are code verifications that you can insert in your code. Using an assertion you can verify a necessary condition for the correct operation of the program. That way the code verification is automatic.

ASSERT(*word1...wordn* ***)*** `( -- flag ncode )` CForth

The ASSERT(and) words define an assert region that can contain any number of words. At the assertion end when the ***)*** word is processed two numbers are popped from the stack. A flag from the NOS and a ncode from the TOS.
If the flag is true the code continues after the assertion region. If the flag is false the program aborts with an assertion fail message that includes the ncode number.

Assertions are conditional compilation elements. By default the assertions are treated like comments and are not included in the compiled code.

If you activate the assertions with the ***ASSERT-ON*** word, any following word that is compiled will include in the compiled code any assertion it finds in the source.

As an example we will add an assertion in the factorial function that checks that the argument is greater than zero.

```
ASSERT-ON
\Factorial calculation
: factorial ( n -- n! )
ASSERT( DUP 0> 1 )
DUP
1 > IF
 DUP 1- RECURSE *
ENDIF
```

```
;
-1 factorial
RUN ERROR: Assert failed with code 1
Backtrace: 12 >> FACTORIAL <<
<3> Top: -1
```

As the assertion is included in the compiled factorial word it will be checked in any use o the factorial word regardless of the assert activation:

```
ASSERT-OFF
0 factorial
RUN ERROR: Assert failed with code 1
Backtrace: 12 >> FACTORIAL <<
```

If you want to eliminate the assertion you need to deactivate assertions and recompile the word.

```
FORGET factorial
ASSERT-OFF
\Factorial calculation
: factorial ( n -- n! )
ASSERT( DUP 0> 1 )
DUP
1 > IF
 DUP 1- RECURSE *
ENDIF
;
-1 factorial
<1> Top: -1
```

You can use ***SEE*** with the factorial example to view how the assertion is inserted in the code.

The normal use of assertions and debug zones is to include them in the source code and activate them during the debugging of the application. After the application is debugged assertions and debug zones are no longer needed and we can recompile all the code deactivating the assertions.

Debug zones and assertions cannot be nested. The following example tries to nest two debug zones when debug is not active:

```
: nested
DEBUG( DEBUG( 1 2 ) ) ;

Compilation aborted at line 2
ERROR: Out of context ")"
```

The first ***debug(*** word ignores all words until the first ***)*** . The second ***)*** generates an error.

If you do the same with debug activated a different error takes place:

```
: nested
DEBUG( DEBUG( 1 2 ) ) ;

Compilation aborted at line 2
ERROR: Nested debug zone
```

8. SYSTEM MANAGEMENT

This section describes several words for system management in CForth.

8.1. Saving and loading the User Dictionary

The user dictionary resides in RAM memory. That means that you need to save it before you disconnect the system if you want to conserve its contents.
CForth is designed to be able to work without filesystem support. That means that the user dictionary save location is **port** dependent. In medium size microcontrollers it can, for instance, be saved in flash memory.

SAVE	CForth
Saves the user dictionary.	
The save location is **port** dependent.	

If there is save data available it will be automatically loaded at startup. A start message will show that the data was loaded.
You can reload the saved data using the ***LOAD*** word to eliminate any change made since the last load.

LOAD	CForth
Loads the user dictionary if saved information is available.	

8.2. Setting a start word

One of the target applications of CForth is the development of embedded applications for microcontrollers. Setting a start word enables the system to execute a selected word upon startup so that you don't need to manually use the console to start the application operation.

@START <word>
Set a word to execute at start-up
Use the word NOWORD to indicate that no start word should be used.
You need to save the User Dictionary using ***SAVE*** for the changes to take effect on the next startup of the system.

The start word is executed in the main thread. That means that while it is running you don't get access to the interactive console. If you want to have access to the console you can set the start word to launch another word in a new thread before returning to give control to the console.

8.3. Introducing source code

CForth is designed for interactive work. You develop the application word by word testing all elements as they are introduced.

Normally you should conserve a copy of the source code you introduce in the system. That way you will be able in the future to introduce a large set of tested words.

You can cut and paste from a text file to the console to introduce a set of words. It is OK if all goes well. If there is any error inside the introduced code it will be difficult to relate to the original text file. Moreover, the error can lead to a cascade of errors that will clutter the console output.
The two words ***FSTART*** and ***FEND*** eases dumping source files on the console.

FSTART
Marks the start of a file stream entering in **FILE** mode.
Line numbers will be counted from 1 in the line that contains this word.
In **FILE** mode the prompt and the top of the stack are not shown after each processed line.
On error it enters in FILE ERROR condition. In this condition the following tokens will be ignored and the verbose level will be set to null until the word ***FEND***.

FEND
Marks the end of a file stream going out of FILE mode.
Clears the FILE ERROR condition.

As an example of the use of the file mode words we can cut and paste on the console the following code. This code include line numbers as reference.

```
\Start of file ------------
FSTART           \01
10 constant c1   \02

: w1 1 2 ;       \04

: w2 err 4 ;     \06
;                \07

\Comment         \09
: w3 1 2 ;       \10
FEND             \11
\End of file --------------
```

CForth should process all lines trough line 6 and show this error:

```
ERROR: Token >>>err<<< not recognized in compile mode
Compilation aborted at line 6
```

After the error no message should be shown until the FEND word is read.
Depending on the console/terminal implementation used to communicate with MFoth it is possible that automatic echo is enabled. In this case the rest of the file will also be shown on the console.

Observe in the example that the misplaced ***;*** in line 7 is not detected as in file mode only the first error is shown.

8.4. Decompiling the code

We have seen the use of the SEE word to view the code associated to a word. This code, however, cannot be compiled again.
The decompiling allows you to obtain source code that is equivalent of the compiled code of a word.

You should always have a copy of all code you introduce in CForth as there is no better code that the source to modify an application. Decompilation is only a fast way to get the code of the compiled words so they can be introduced again.

Due to how the ***FORGET*** word operates you cannot eliminate a word without eliminating any word you have entered after it. When you need to eliminate a word that is not the last one without losing the words you entered after it you can decompile the words you want to conserve and introduce them after the ***FORGET*** command is used.

DECOMPILE <word> CForth
Decompiles a word.
Generates a sequence of text lines that can be used to recompile the word in the future.

As an example we will decompile the until_test we used before:

```
: until_test ( -- )        \Show numbers from 0 to 10
0 BEGIN                    \Start of loop with 0 TOS
DUP . 1+                   \Show TOS and add 1
DUP 10 > UNTIL DROP ;      \Repeat until TOS > 10 and drop TOS
```

We can see the code generated inside the memory using the SEE word:

```
SEE until_test
Decoding of word until_test
     44 : 0
     46 : DUP
     47 : .
     48 : 1+
     49 : DUP
     50 : 10
     52 : >
     53 : JZ 46
     56 : DROP
     57 : ENDWORD
```

As we know, the compilation of the source code converts the flow constructs in jump instructions. Observe that the JZ (Jump if zero) word uses the absolute address 46 that points to the start of the loop.

Let's see what we get using DECOMPILE:

```
DECOMPILE until_test

\UNTIL_TEST Decompilation
: UNTIL_TEST
0
DUP
.
1+
DUP
10
>
[ -7 ] JZ
DROP
;
```

In general we get the same we have seen using the ***SEE*** word. The address of the JZ word has been converted to relative to make the code independent of its coding position. The -1 is between ***[*** and ***]*** because that makes -7 go to the TOS during compilation. The JZ word, at compile time, gets the relative jump address from the stack to create the absolute JZ word.

As CForth understands the JZ word, you could use it inside your own programs. It is not recommended however because CForth source files have no elements to work easily with numeric addresses.

Decompile not only works with code words, it also works with values, variables and create fields. In the following example we create a variable, with two added fields. Then we store a number in the variable and decompile it.

```
VARIABLE var 6000 , 78 C,
6700 var !
DECOMPILE var

\VAR Decompilation
VARIABLE VAR 6700 VAR !
112 C, 23 C, 0 C, 0 C, 78 C,
```

As the variable is not zeroed, the decompiled code includes the necessary code to store the value it held when decompiled. The decompiler cannot know that the 6000 32 bit value was introduced with **,** so the added data to the variable is decompiled as a series of chars. The same applies with create fields:

```
CREATE crf 6 C, 200 H, 2049 , 2 ALLOT
DECOMPILE crf

\CRF Decompilation
CREATE CRF
6 C, -56 C, 0 C, 1 C, 8 C, 0 C, 0 C, -1 C, -1 C,
```

All stored data is decompiled as char values independent of how it was coded. The memory contents are however the same.

If you want to get a decompiled dump of all the contents of the memory you can use the ***DECOMPILEALL*** word.

DECOMPILEALL
Decompiles the User Dictionary from the first to the last coded word.
Compiling the obtained decompiled text should give same exact memory contents on the User Dictionary.

The decompiled source text includes a FSTART and a FEND word to ease the reintroduction of the code.
One use of decompilation and recompilation is removing the unnecessary or obsolete words from the decompiled source before reintroducing it in the system.

6. DICTIONARIES

CForth uses four dictionaries for its words. Three dictionaries give support to buil-in words and the last is the User Dictionary that stores user defined words.

6.1. Base dictionary

The base dictionary is the dictionary that holds the words that can be coded inside a new word. Most CForth built-in words are in this dictionary. This dictionary also contains some hidden words that CForth uses to code numbers, variables, etc...

6.2. Interactive dictionary

The interactive dictionary holds the words that can be used in interactive mode and cannot be coded inside a program.
Immediate words that cannot be coded inside a new word but can also be used in compile mode have entries in both the interactive and generator mode.

6.3. Generator dictionary

The generator dictionary holds the words that can be used in compile mode and are not directly coded. That includes program flow words like branches and loops that are coded as jumps and other hidden words.

6.4. User dictionary

The user dictionary holds all user defined words. This dictionary holds alls user programs. This dictionary can be saved using the ***SAVE*** or ***DECOMPILEALL*** words.

7. CFORTH PORTS

CForth can be compiled in any 32 bit system. To work in a system it needs at least some **port** code. The **port** code provides at least the access to the console.
Currently there area two CForth ports.

7.1. MinGW Port

The MinGW port is a CForth port for 32 bit Windows under MinGW.
It provides most MFort capabilities but it doesn't provide threads.
This port was created to easily program the CForth core. Cross compiling for a microcontroller requires more time and gives wear on the microcontroller flash memory. Programming in MinGW gives a more easy way to program and test CForth.

7.2. F3 Gizmo

This is were all started. This port is the reason why CForth exist. CForth was developed to create an interactive test and develop system for the STM32F3 Discovery Board.

This port currently provides access in this board to 8 LEDs, 10 digital I/O, 3 PWM channels, 6 analog inputs, one DAC output a SPI bus, an I2C bus, 2 timers and the three board peripheral devices: an accelerometer, a magnetometer and a gyroscope.
More devices will be supported in future versions.

8. CFORTH IS NOT FORTH

CForth is not ANS Forth compliant and don't pretend to be. In fact it is not even Forth compliant. Though this manual several Forth compatibility aspects have been explained. This section explains in a single place the most important departures of CForth from the ANS Forth standard.

8.1. Console centric

CForth is designed to operate using a console as the only user interaction device.
CForth don't use files in most ports
Input anb output data goes through the console.

8.2. Circular parameter stack

The CForth parameter stack is circular. That means that it cannot overflow. It can, however, silently lose elements from the bottom when it is full.

8.3. Numeric constants

Number base is implicit in the numbers. There is no BASE variable in CForth. An hexadecimal constant number must be preceded by 0x.
To write hexadecimal numbers special words like X. or X.R must be used.
Chars can be introduced as numeric constants associated to their ASCII code using single quotes like in 'A'.

8.4. Half Cell storage

The normal storage options in Forth are Cell and char. CForth supports also 16 bit Half Cell storage options.
The storage size is encoded in the variables so in CForth it is possible to have commands ***V!*** ***V+*** and ***V@*** that store and fetch using sizes tailored to the storage.

8.5. String delimiters

CForth don't use words to mark the start of strings but delimiters. That means that you don't need to leave a space between ." or S" and the start of one string.
You don't need to leave a space between the comment \ marker and the start of the comment.
That also means that you cannot create words whose names start with ***." S" "*** or \

8.6. No compiler expansion

Most Forth compilers are mainly written in Forth. They make use of the Forth capabilities to create new defining words.
This is not the case of CForth that is written in C.
In CForth you don't have the words DOES> INMEDIATE or POSPONE you would need to create rich defining words.
It is not that it wasn't possible. It just don't suit the CForth objectives.

9. FORTH REFERENCES

CForth is not Forth but has a lot of elements in common. This section provides some very good Forth references.

Forth.org is the site to obtain information about Forth.
The following link contains several Forth tutorials:
http://www.forth.org/tutorials.html

The stating forth by Leo Brodie is a very good introduction to the Forth language
http://www.forth.com/starting-forth/

You can find the standard ANS forth word list here:
http://lars.nocrew.org/dpans/dpans6.htm

GForth is the Forth implementation of the GNU project. I you want to program in Forth, GNU is the way to go. The GNU forth extends ANS Forth including some new words. You can find the GNU Forth details in the reference below:
http://bernd-paysan.de/gforth.html
Its manual is at the following url:
http://www.complang.tuwien.ac.at/forth/gforth/Docs-html/index.html