ciforth manual

A close-to-ISO/common intel/computer intelligence/CH+ forth.

This is a standard-ISO Forth (mostly, see the section portability) for the configuration called lina32:

- version 5.5.0
- 32-bits protected mode
- running under Linux using system calls directly
- contains security words
- the full ISO CORE set is present, possibly after loading
- headers with source fields
- accommodates threads

Albert van der Horst

1 Overview

Forth is an interactive programming system. ciforth is a family of Forth's that can be generated in many different version for many different operation systems. It is sufficiently close to the ISO standard to run most programs intended to be portable. It deviates where less used features where objectionable to implement. See Chapter 4 [Manual], page 7, Section Portability.

This file documents what you as a user needs to know for using this particular version of ciforth called "lina32" once it is installed on your system.

ciforth consists of three files:

- lina32 : the program
- One of `ciforth.ps` `ciforth.pdf` `ciforth.html` : the documentation
- `forth.lab` : source library for auxiliary programs

These files are generated together by a generic system from the file `fig86.gnr` . The documentation applies to the ciforth with which it goes.

If your Forth doesn't fit the description below get a new version. The information below allows an expert to reconstruct how to generate a corresponding version. Not all of it may make sense to you. Tell him whether you want to fit the Forth to the description or vice versa (see Chapter 3 [Rationale & legalese], page 5).

These are the features:

All ciforth's are *case sensitive* . This is version 5.5.0. It is running in protected mode. It is running under Linux Blocks are allocated in files. A number has a precision of 32 bits. It calls linux system from Forth directly. It has compiler security, sacrificing some ISO compatibity. It use `PP` instead of the ISO `>IN` . It contains the full ISO CORE in the kernel, more than is needed to make it self contained. It contains a field in the header to point to source. It is indirect threaded.

If you are new to Forth you may want to read the Gentle Introduction, otherwise you better skip it. The third chapter most users will not be interested in.

2 Gentle introduction

A Forth system is a database of small programs. The database is called the dictionary. The programs are called *word* 's, or definitions. The explanation of words from the dictionary is called a glossary.

First of all, a Forth system is an environment that you enter by running it:
'lina32'
Like in a Disk Operating System a *word* is executed by typing its name, but unlike in a DOS several programs can be specified on the same line, interspersed with numbers. Also names can be anything, as long as they don't contain spaces.

A program may leave one or more results, and the next program can use it. The latest result is used up first, hence the name lifo buffer. (last in, first out).

For example:

```
albert@apple:~/forth/fig86 > lina32

80386 ciforth beta $RCSfile: ci86.gnr,v $ $Revision: 6.122 $

1 2 + 7 *
 OK
.
21  OK
```

1 2 and 7 are numbers and are just remembered as they are typed in. 'OK' and '21 OK' are the answer of the computer. + is a small program with an appropriate name. It adds the two numbers that were entered the latest, in this case 1 and 2. The result 3 remains, but 1 and 2 are consumed. Note that a name can be anything, as long as it doesn't contain spaces. The program * multiplies the 3 and the 7 and the result is 21. The program . prints this results. It could have been put on the same line equally easily.

You will be curious about what are all those commands available. Of course they are documented, but you can find the exact set by typing WORDS . Programs can be added to the database by special programs: the so called *defining word* 's. A defining word generally gets the name of the new word from the input line.

For example: a constant is just a program that leaves always the same value. A constant is created in this way, by the defining word CONSTANT :

```
127 CONSTANT MONKEY   12 .
12 OK
```

You can check that it has been added, by typing WORDS again.

The above must not be read like:
a number, two programs and again a number etc.... ,
but as:
a number, a program and a name that is consumed,
and after that life goes on. The '12 .' we put there for demonstration purposes, to show that CONSTANT reads ahead only one word. On this single line we do two things, defining 'MONKEY' and printing the number 12. We see that CONSTANT like any other program consumes some data, in this case the 127 that serves as an initial value for the constant called 'MONKEY' .

You may get ‘ `constant ? ciforth ERROR # 12 : NOT RECOGNIZED` ’. That is because you didn’t type in the above precisely. lina32 is case sensitive. If you want to change that consult the section "Common problems". (see Chapter 7 [Errors], page 51).

A very important defining word is `:` , with its closure `;` .

```
: TEST 1 2 + 7 * ;     21 .
21 OK
```

In this case not only the name ‘`TEST`’ is consumed, but none of the remaining numbers and programs are executed, up till the semicolon `;` . Instead they form a specification of what ‘`TEST`’ must do. This state, where Forth is building up a definition, is called *compilation mode* . After the semicolon life continues as usual. Note that `;` is a program in itself too. But it doesn’t become part of `TEST` . Instead it is executed immediately. It does little more than turning off compilation mode.

```
TEST TEST + .
42  OK
: TEST+1 TEST 1 + . ;  TEST+1
22 OK
```

We see that ‘`TEST`’ behaves as a shorthand for the line up till the semi colon, and that in its turn it can be used as a building block.

The colon allows the Forth programmer to add new programs easily and test them easily, by typing them at the keyboard. It is considered bad style if a program is longer than a couple of lines. Indeed the inventor of Forth Chuck Moore has written splendid applications with an average program length of about one line. Cathedrals were built by laying stone upon stone, never carved out of one rock.

The implementation of the language Forth you look at is old fashioned, but simple. You as a user have to deal with only three parts/files : this documentation, the executable program, and the library file, a heap of small programs in source form. There may be several documentation files, but they contain the same information in a different format.

There is an ISO standard for Forth and this Forth doesn’t fully comply to it. Still by restricting yourself to the definitions marked as ISO in the glossary, it is quite possible to write an application that will run on any ISO-compliant system.

Because of the way Forth remembers numbers you can always interrupt your work and continue. For example

```
: TEST-AGAIN
1 2 + [ 3 4 * . ]
12 OK
7 * ;
OK
```

What happened here is that some one asked you to calculate “3 times 4” while you were busy with our test example. No sweat! You switch from compilation mode to normal (interpret) mode by `[` , and back by `]` . In the meantime, as long as you don’t leave numbers behind, you can do anything. (This doesn’t apply to adding definitions, as you are in the process of adding one already.)

3 Rationale & legalese

3.1 Legalese

This application currently is copyright by Albert van der Horst. This Forth is called ciforth and is made available by the D.F.W.. This publicationis is available under GPL 2, the GNU public license. The file `COPYING` containing the legal expression of these lines must accompany it.

You can make closed source program with the -c options. Such programs are naturally your own, quite the same as programs build with the gcc compiler. However also if you base an application program on ciforth, you need not make its source available, even if the application contains the Forth interpreter exposed. This would be a derived work and, in a strict interpretation of the GPL, such interpretive systems based on ciforth are always legally obliged to make the source available. Because Forth is "programming by extending the language" , we consider this "normal use in the Forth sense" as expressed by the following statement.

In addition to the GPL Albert van der Horst grants the following rights in writing:

1. The GPL is interpreted in the sense that a system based on ciforth and intended to serve a particular purpose, that purpose not being a "general purpose Forth system", is considered normal use of the compilation system, even if it could accomplish everything ciforth could, under the condition that the ciforth it is based on is available in accordance to the GPL rules, and this is made known to the user of the derived system.
2. Code snippets in the library inasfar not in the public domain are available under the LGPL, so you can freely borrow from it or build on it.

3.2 Rationale

This Forth is meant to be simple. What you find here is a Forth for the Intel 86. You need just the executable to work. You choose the format you prefer for the documentation. They all have the same content. You can use the example file with blocks, you have the assembler source for your Forth, but you can ignore both.

3.3 Source

In practice the GPL means (: this is an explanation and has no legal value!)

They may be further reproduced and distributed subject to the following conditions:

The three files comprising it must be kept together and in particular the reference section with the World Wide Web sites.

The latest version of lina32 is found at
'`http://home.hccnet.nl/a.w.m.van.der.horst/ciforth.html`'.

Via that link you can also download ciforth's for other OS's and the generic system, if you want to make important modifications. Also you can see how you can contact the author. Otherwise in case of questions about this ciforth, contact the person or organisation that generated it for you.

This Forth builds on fig-Forth. It is based on the work of Charlie Krajewski and Thomas Newman, Hayward, Ca. still available via taygeta. The acknowledgments for systems that serves as a base, in particular the original fig-Forth, are found in the generic documentation, including detailed information how these systems can be obtained.

Important:
If you just want to use a Forth, you most certainly do not want the generic system. Great effort is expended in making sure that this manual contains all that you need, and nothing that

might confuse you. The generic system on the contrary contains lots that you don't need, and is confusing as hell.

If you are interested in subjects like history of Forth, the rationale behind the design and such you might want to read the manual for the generic Forth.

3.4 The Generic System this Forth is based on.

The source and executable of this ciforth was generated, out of at least dozens of possibilities, by a generic system. You can configure the operating system, memory sizes, file names and minor issues like security policy. You can select between a 16, 32 and 64 bit word size. You may undertake more fundamental changes by adapting one or more of the macro header files. An important goal was to generate exactly fitting documentation, that contains only relevant information and with some care your configuration will have that too. This generic system can be obtained via '`http://home.hccnet.nl/a.w.m.van.der.horst/ciforth.html`'. '

4 Manual

4.1 Getting started

4.1.1 Hello world!

Type ‘lina32’ to get into your interactive Forth system. You will see a signon message. While sitting in your interactive Forth doing a “hello world” is easy:

```
"Hello world!" TYPE
Hello world! OK
```

Note that the computer ends its output with ‘OK’ to indicate that it has completed the command.
Making it into an interactively usable program is also easy:

```
: HELLO "Hello world!" TYPE CR ;
OK
HELLO
Hello world!
OK
```

This means you type the command ‘HELLO’ while you are in lina32. As soon as you leave lina32, the new command is gone.

If you want to use the program a second time, you can put it in a file hello.frt. It just contains the definition we typed earlier:

```
 : HELLO "Hello world!" TYPE CR ;
```

This file can be INCLUDED in order to add the command ‘HELLO’ to your Forth environment, like so:

```
"hello.frt" INCLUDED
OK
HELLO
Hello world!
OK
```

During development you probably have started with ‘lina32 -e’, so you need just type

```
INCLUDE hello.frt
```

In order to make a stand alone program to say hello you can use that same source file, again hello.frt. Now build the program by
lina32 -c hello.frt

(That is `c` for compile.) The result is a file `hello` . This file can be run from your command interpreter, or shell. It then will execute the last word defined in the source file in this case '`HELLO`' . It is a single file that you can pass to some one else to run on their computer, without the need for them to install Forth. For the compiler to run you must have the library correctly installed.

If that failed, or anything else fails, you will get a message with at least '`ciforth ERROR ###`' and hopefully some more or less helpful text as well. The '`###`' is an error number. See Chapter 7 [Errors], page 51, Section Explanations.

Note for the old hands. Indeed the quoted strings are not ISO. They surely are a Forth-like extension. Read up on denotations, and the definition of `"` .

In lina32 you never have to worry about the life time of those quoted strings, they are allocated in the dictionary and are permanent.

4.1.2 The library.

If you want to run a program written on some other Forth, it may use facilities that are not available in lina32's kernel, but they may be available in the *library* . A library is a store with facilities, available on demand. Forth as such doesn't have a library mechanism, but lina32 does.

lina32 uses the *blocks* as a library by addition of the word `WANTED` and a convention. Starting with '`lina32 -w`' or most any option you have this facility available. If you are already in lina32, you can type '`1 LOAD`'. The extension of '`.lab`' in `forth.lab` means Library Addressable by Block.

Now we will add `DO-DEBUG` using this library mechanism. It is used immediately. It is handy during development, after every line it shows you what numbers Forth remembers for you. Also from now on the header of each block that is `LOAD` -ed is shown.
Type ('`1 LOAD`' may not be necessary):

```
1 LOAD
"DO-DEBUG" WANTED
OK
DO-DEBUG

S[ ] OK 1 2

S[ 1 2 ] OK
```

(You can turn `DO-DEBUG` off with `NO-DEBUG` .)

More convenient than `WANTED` is `WANT` that adds all words that are on the remainder of the line, so without quotes.

If you try to `INCLUDE` a program, you may get errors like '`TUCK? ciforth ERROR # 12 : NOT RECOGNIZED`'. See Chapter 7 [Errors], page 51, Section Explanations. Apparently, lina32 doesn't know about a forth word named `TUCK` , but after '`"TUCK" WANTED`' maybe it does. You may try again.

The library file must be organized for `WANTED` in a particular way to find something. It is in fact familiar. It is divided into blocks of 16 lines. The convention about the way the library file must be organized for `WANTED` to find something is simple. It is divided into blocks of 16 lines. The first line is the header of the block, the so called *indexline*

If the word we are looking for is mentioned in the header, that block is compiled. There may be several blocks that define a particular word. If the first block is terminated early, the word is not yet defined and the next blocks is loaded. E.g. a words like `?32` mark 32-bits code and the screen is terminated if the Forth is 64 bits. This goes on until the word is defined, or the end of the screens is reached. The terminators that cut loading a screen short, are defined by the the `CONFIG` defining word. The last screen is marked by an empty index line. There may be names that represent a whole package. Among those symbolic names are '`-fp- -fpwa- ASSEMBLERi86 ASSEMBLERi86-HIGH -traditional-` '. After `WANT` symbolic names may not be in the dictionary, but that they are not intended to be executed anyway.

There is really nothing much to it. The bottom line is that one library file serves a range of operating systems and cell sizes.

The library file contains examples for you to load using `WANT` . Try

```
WANT SIEVE
LIM # 4 ISN'T UNIQUE
OK
10 SIEVE
KEY FOR NEXT SCREEN
ERATOSTHENES SIEVE -- PRIMES LESS THAN 10 000
0 002 003 ...
(lots of prime numbers.)
```

4.1.3 Development.

If you want to try things out, or write a program – as opposed to just running a ready made program – you best start up lina32 by '`lina32 -e`'. That is *e* for elective. '`lina32 -e`' instructs lina32 to load screen 5 (*e* is the 5-th letter.)

You can configure this screen 5 to suit your particular needs, by just using some programmers editor. We will come back to that later.
You will have available:

1. `WANTED` and `WANT` . '`WANT xxx yyy`' is equivalent to '`"xxx" WANTED "yyy" WANTED`' , but it is more convenient.
2. `DH. H. B. DUMP FARDUMP`

   For showing numbers in hex and parts of memory.
3. `SEE`

   To analyse words, showing the source code of compiled words. (Also known as `CRACK` .)
4. `LOCATE`

   To show the part of the source file where the word is defined, or, if loaded from the library file, the block where it is defined.
5. `OS-IMPORT`

   To be able to type shell-commands from within Forth as if you were in a terminal window.

   . . .

Because this ciforth is "hosted", meaning that it is started from an operating system, you can develop in a convenient way. Start lina32 in a window, and use a separate window to start your editor. Try out things in lina32. If they work, paste the code into your editor. If a word works, but its source has scrolled off the screen, you can recover the source using `SEE` . If you have constructed a part or all of your program, you can save it from your editor to a file. Then by the command '`INCLUDE <file-name>` ' load the program in lina32 and do some further testing.

You are not obliged to work with separate windows. Suppose your favorite editor is called `vi`. After

```
"vi" OS-IMPORT vi
```

you can start editing a file in the same way as from the shell. Of course you now have to switch between editing a file and lina32. But at least you need not set up your Forth again, until your testing causes your Forth to crash.

4.1.4 Finding things out.

If you want to find things out you must start up lina32 again by '`lina32 -e`'. The sequence

```
WANT TUCK
LOCATE TUCK
```

shows you the source for TUCK if it is in the library somewhere.

```
WANT TUCK
SEE TUCK
```

show you the source for TUCK if it is in the library or in the kernel, but without comment or usage information.

4.2 Concepts

A forth user is well aware of how the memory of his computer is organised. He allocates it for certain purposes, and frees it again at will.

The last-in first-out buffer that remembers data for us is called the *data stack* or sometimes *computation stack* . There are other stacks around, but if there is no confusion it is often called just the *stack* . Every stack is in fact a buffer and needs also a *stack pointer* to keep track of how far it has been filled. It is just the address where the last data item has been stored in the buffer.

The *dictionary* is the part of the memory where the *word*'s are (see Section 9.6 [DICTIONARY], page 81). Each word owns a part of the dictionary, starting with its name and ending where the name of the next word starts. This structure is called a *dictionary entry* . Its address is called a *dictionary entry address* or *DEA* . In ciforth's this address is used for external reference in a consistent way. For example it is used as the *execution token* of a word in the ISO sense. In building a word the boundary between the dictionary and the free space shifts up. This process is called *allocating* , and the boundary is marked by a *dictionary pointer* called `DP` . A word can be executed by typing its name. Each word in the dictionary belongs to precisely one *word list* , or as we will say here namespace. Apart from the name, a word contains data and executable code, (interpreted or not) and linking information (see Section 9.4.7 [NAMESPACE], page 76). The order of words in a wordlist is important for looking them up. The most recent words are found first.

The concept word list is part of the ISO standard, but we will use *namespace* . A namespace is much more convenient, being a word list with a name, created by `NAMESPACE` . ISO merely knows *word list identifier* 's, a kind of handle, abbreviated as *WID* . A new word list is created by the use of `NAMESPACE` . When looking up a word, only the wordslists that are in the current *search order* are found. By executing the namespace word the associated word list is pushed to the front of the search order. In fact in ciforth's every DEA can serve as a WID. It defines a wordlist consisting of itself and all earlier words in the same namespace. You can derive the WID from the DEA of a namespace by `>WID` .

A word that is defined using `:` is often called a *colon definition* . Its code is called *high level* code.

A high level word, one defined by `:` , is little more than a sequence of addresses of other words. The *inner interpreter* takes care to execute these words in order. It acts by fetching the address pointed by 'HIP' , storing this value in register 'WOR'. It then jumps to the address pointed to by the address pointed to by 'WOR'. 'WOR' points to the code field of a definition which (at offset forthdef(>CFA)) contains the address of the code which executes for that definition. For speed reasons this offset is choosen to be zero. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of Forth.

If the inner interpreter must execute another high level word, while it is interpreting, it must remember the old value of 'HIP', and this so called *nesting* can go several levels deep. Keeping this on the data stack would interfere with the data the words are expecting, so they are kept on a separate stack, the *return stack* . Apart from 'HIP' and 'WOR' the return and data stack are kept in registers named 'RPO' and 'SPO'. If you're interested in the actual registers, you can inspect the assembler source file that goes with this Forth. The usage of two stacks is another hall mark of Forth.

A word that generates a new entry in the dictionary is called a *defining word* (see Section 9.4 [DEFINING], page 74). The new word is created in the `CURRENT` word list .

Each processor has a natural size for the information. (This is sometimes called a machine word). For a Pentium processor this is 32 or 64 bit, for the older Intel 8086 it is 16 bit. The pendant in Forth is called a *cell* and its size may deviate from the processor you are running on. For this ciforth it is 32, It applies to the data remembered in the data stack, the return addresses on the return stack, memory accesses `@` and `!` , the size of `VARIABLE` ' s and `CONSTANT` ' s. In Forth a cell has no hair. It is interpreted by you as a signed integer, a bit-map, a memory address or an unsigned number. The operator `+` can be used to add numbers, to set a bit in a bitmap or advance a pointer a couple of bytes. In accordance with this there are no errors such as overflow given.

Sometimes we use data of two cells, a *double* . The high-order cell is most accessible on the stack and if stored in memory, it is lowest.

The code for a high level word can be typed in from the terminal, but it can also fed into Forth by redirection from a file, `INCLUDED` from a file or you can *load* it from the file `forth.lab`, because you can load a piece of this library at will once you know the block number. This file is divided into *blocks* of 1 Kbyte. They may contain any data, but a most important application is containing source code. A block contain source code is called a *screen* . It consists of 16 lines of 64 characters. In ciforth the 64-th character is `^J` such that they may be edited in a normal way with some editors. To *load* such a screen has the same effect as typing its content from the terminal. The extension lab stands for *Library Addressable by Block* ,

Traditionally Forthers have things called *number* 's, words that are present in the source be it interpreted or compiled, and are thought of not as being executed but rather being a description of something to be put on the stack directly. In early implementations the word 'NUMBER' was a catch-all for anything not found in the dictionary, and could be adapted to the application. For such an extensible language as Forth, and in particular where strings and floating point numbers play an increasing role, numbers must be generalised to the concept of *denotation* 's. The need for a way to catch those is as present as it was in those early days. Denotations put a constant object on the stack without the need to define it first. Naturally they look, and in fact are, the same in both modes. Here we adopt a practice of selecting a type of the denotations based on the first letters, using `PREFIX` . This is quite practical and familiar. Examples of this are (some from C, some from assemblers, some from this Forth) :

```
10
'a'
^A
0DEAD
$8000403A
0x8000403A
#3487
0177
S" Arpeggio"
"JAMES BROWN IS DEAD"
" JK "
'DROP
' DROP
```

These examples demonstrate that a denotation may contain spaces, and still are easy to scan. And yes, I insist that '`' DROP`' is a denotation. But '`'DROP`' is clearer, because it can only be interpreted as such; it is not a valid word.

Of course a sensible programmer will not define a word that looks like a denotation :

```
: 7 CR "This must be my lucky day" TYPE ; ( DON'T DO THIS)
```

4.3 Portability

If you build your words from the words defined in the ISO standard, and are otherwise careful, your programs may run on other systems that are ISO standard.

There are no gratuitous deviations from the standard, but a few things are not quite conforming.

1. The error system uses `CATCH` and `THROW` in a conforming way. However the codes are not assigned according to the table in the standard. Instead positive numbers are ciforth errors and documented in this manual. ciforth's errors identify a problem more precisely than the standard admits. An error condition that is not detected has no number assigned to it. Negative numbers are identical to the numbers used by the host operating system. No attempt is made to do better than reproduce the messages belonging to the number as given by strerror.
2. As `ABORT" ABORT QUIT` are not implemented using `THROW` it is not possible to catch those words.
3. There is no `REFILL` . This is a matter of philosophy in the background. You may not notice it.
   Consequences are that `BLK` is not inspected for every word interpreted, but that blocks in use are locked. Files are not read line by line, but read in full and evaluated.
4. It uses `PP` instead of the ISO `>IN` . The `>IN` that is available via the library is to be loaded only via `WANT -traditional-` and then work as in expected. In particular `INCLUDED` compiles a file line by line. `PP` could be manipulated to have such effects manipulating `>IN`
5. Counting in do loops do not wrap through the boundary between negative and positive numbers. This is not useful on Forths of 32 bits and higher; for compatibility among ciforths 16 bit ciforths don't wrap either.
6. Namespaces are wordlists with a name. They push the wordlist to the search order, instead of replacing the topmost one, as is done by VOCABULARY (not an ISO-word) that is present in some other Forth's. In this sense `FORTH` and `ASSEMBLER` words are not conforming.

7. This is not strictly non-conforming, but worth mentioning here. In fact lina32 contains only one state-smart word besides `LITERAL` (that word is `."`). All denotations are state-smart only because they use `LITERAL`

   and the result is correct ISO behaviour for numbers. Knowledge of this is used freely in the libraries of ciforth; it is the right of a system developer to do so. The library is not a supposedly ISO-conforming program. It tends to rely on ciforth-specific and lina32-specific – but hopefully documented – behaviour. Understanding it requires some study of non-portable facilities.
8. When a file is `INCLUDED` it is read in as a whole, so there is no need for `REFILL` . After `WANT REFILL` a `REFILL` is loaded that sets the parse pointer to the start of the next line. Moreover `0>IN` will set the parse pointer to the start of the current line. In many cases this will accomplish the effect described by the standard. If this doesn't help, use `-traditional-` . See Chapter 4 [Manual], page 7,, subsection '`REFILL`' .

Here we will explain how you must read the glossary of lina32, in relation to terminology in the ISO standard.

Whenever the glossary specifies under which conditions a word may *crash* , then you will see the euphemism *ambiguous condition* in the ISO standard.

For example:
Using `HOLD` other than between `<#` and `#>` leads to a crash.

Whenever we explicitly mention ciforth in a sentence that appears in a glossary entry, the behaviour may not apply to other ISO standard systems. This is called *ciforth specific behaviour* . If it mentions "this ciforth" or "lina32", you cannot even trust that behaviour to be the same on other ciforth systems. Often this is called an "implementation defined" behaviour in the standard. A typical example is the size of a cell. Indeed we are obliged to specify this behaviour in our glossary, or we don't comply to the standard. The behaviour of the other system may very well be a crash. In that case the standard probably declares this an "ambiguous condition".

For example:
On this ciforth `OUT` is set to zero whenever `CR` is executed.

The bottom line is that you never want to write code where lina32 may crash. And that if you want your code to run on some other system, you do not want to rely on *ciforth specific behaviour* . If you couldn't get around that, you must keep the specific code separate. That part has to be checked carefully against the documentation of any other system, where you want your code to run on.

By using `CELL+` it is easy to keep your code 16/32/64 bit clean. This means that it runs on 16, 32 and 64 bits systems.

4.3.1 REFILL

Some programs rely on line by line loading by `REFILL` . This is substantially different from normal ciforth practice. You want at least `REFILL >IN` , and you probably use `WORD` and `FIND` and the (traditional, but non standard) word of `VOCABULARY` that is different from `NAMESPACE` . This is accomplished by the symbolic target of *-traditional-* .

```
WANT -traditional-
```

All the words that are loaded by this command are in the library and they are marked with '`-traditional-`' in the index line. The word `-traditional-` is not itself defined, expect to get get a warning for that. This is done in order to be able to reload (and forcibly redefine) all words by `WANT -traditional-` .

It is recommended that this is the first facility loaded, that way there is the least chance with interference. The following sequence generates a traditional Forth:

```
lina -a
WANT -traditional- SAVE-SYSTEM
....
"traditionalforth" SAVE-SYSTEM
BYE
```

It is of course possible to enhance this Forth with more words at the place of the dots.

4.3.2 Compatibility with lina32 4.0.x

Since version 5.x changes have been made to increase compatibility with existing practice. By invoking `WANT -legacy-` you load a screen that forces compatibility with 4.x.x versions. You will notice that existing programs either invoke this, or have been reworked to not need legacy items. In either case, those programs have been tested with version 5.x

What the legacy items are can be seen from the screen that has `-legacy-` in its index line. In particular `REQUIRE` `REQUIRED` `PRESENT?` are to be found in those screens. Note that by using legacy items your code may be in conflict with upcoming standards. It is also likely that in those programs traditional words are used that are no longer present in the kernel such as `WORD` `FIND` . These can be loaded by regular `WANT` .

The names `VOCABULARY` and `REQUIRE` are being proposed for standardisation. The ciforth definitions with these names were not compatible with this proposal. So the `REQUIRE` of older versions is now called `WANT` . Likewise `REQUIRED` is renamed to `WANTED` . `VOCABULARY` is renamed to `NAMESPACE` , with the difference that `NAMESPACE` is not immediate. This allows to include the new standardised definitions in a loadable screen.

4.4 Configuring

For configuring your lina32, you may use `"newforth" SAVE-SYSTEM` . This will do most of the time, but then you build in the `SAVE-SYSTEM` command as well. For configuring your lina32, without enlarging the dictionary, you may use the following sequence

```
 S" myforth.lab" BLOCK-FILE $! \ Or any configuration command
1 LOAD
WANT SAVE-SYSTEM
: DOIT
    '_pad 'FORTH FORGET-VOC
    '_pad >NFA @ DP !
    "newforth" SAVE-SYSTEM BYE ;
DOIT
```

Here '`DOIT`' trims the dictionary just before saving your system into a file. `_pad` is the first word of the facilities in screen 1 that was loaded. (This was different in previous version of ciforth.)

`FAR-DP` allows to have a disposable part of the dictionary. If you decide to use this facility for your own purposes, make sure to always `FORGET` the disposed off words. The '`-c`' option uses this to avoid having source files as part of an executable image.

4.5 Saving a new system

We have said it before: "Programming Forth is extending the Forth language.". A facility to save your system after it has been extended is essential. It can be argued that if you don't have that, you ain't have no Forth. It is used for two purposes, that are in fact the same. Make a customised Forth, like **you** want to have it. Make a customised environment, like a customer wants to have it. Such a "customised environment", for example a game, is often called a *turnkey system* in Forth parlance. It hides the normal working of the underlying Forth.

In fact this is what in other languages would be called "just compiling", but compiling in Forth means adding definitions to an interactive Forth. In ciforth "just compiling" is as easy as in any language (see Chapter 4 [Manual], page 7, Hello world!). Of course, whether you have a hosted system like this one or a booted system , it is clear that some system-dependant information goes into accomplishing this.

This has all been sorted out for you. Just use `SAVE-SYSTEM` . This accepts a string, the name you want the program-file to have. Having a program to execute a certain word is even easier,just use the '`-c`' option. See Chapter 4 [Manual], page 7, Section Libraries and options.

In the following it is explained. We use the naming convention of ISO about cells. A cell is the fundamental unit of storage for the Forth engine. Here it is 32 bits (4 bytes).

The change of the boot-up parameters at `+ORIGIN` , in combination with storing an image on disk goes a long way to extending the system. This proceeds as follows:

1. All user variables are saved by copying them from '`U0 @`' to '`0 +ORIGIN`'. The user variable `U0` points to the start of the user area. The length of the area is 0x40 cells. If in doubt check out the variable '`US`' in the assembler code.
2. If all user variables are to be initialised to what they are in this live system skip the next step.
3. Adjust any variables to what you want them to be in the saved system in the `+ORIGIN` area. The initialisation for user variable '`Q`' can be found at '`' Q >DFA @ +ORIGIN`'.
4. Adjust version information (if needed)
5. Copy your lina32 to a new file using `PUT-FILE` . The difficult part is to add to the system specific header information about the new size, which is now from `BM` to `HERE` . The command '`WANT SAVE-SYSTEM`' loads a version that does that correctly for your hosted system.

4.6 Memory organization

A running ciforth has 3 distinct memory areas.

They occur sequentially from low memory to high.

- The dictionary
- Free memory, available for dictionary, from below, and stacks, from above
- The work area for each task with a total size of `TASK-SIZE`

  that must be a power of two. This may be replicated for multi-tasking. It is evenly divided into the data stack, the return stack and user variables, the input buffer for the console, and disk block buffers . The work area is initialised on startup.

  These areas have an alignment of 1/4 of the `TASK-SIZE` . This way the return stack pointer can serve double duty as a user pointer; this is one register less to save for a task switch.

The lowest part of the free memory is used as a scratch area: `PAD` .

The dictionary area is the only part that is initialised, the other parts are just allocated.

The program as residing on disk must contain the first area. In addition it contains a header, to tell the Linux how to transfer the program to memory. Logically the Forth system consists of these 7 parts.

- Boot-up parameters
- Machine code definitions
- Installation dependant code
- High level standard definitions
- High level user definitions
- System tools (optional)
- RAM memory workspace

4.6.1 Boot-up Parameters

The boot-up area contains initial values for the registers needed for the Forth engine, like stack pointers, the pointers to the special memory area's, and the very important dictionary pointer `DP` that determines the boundary between the dictionary and free space.

They are copied to a separate area the *user area* , each time Forth is started. The bootup area itself is not changed, but the variables in the user area are. By having several user area's, and switching between them, ciforth supports multitasking. When you have made extensions to your system, like for instance you have loaded an editor, you can make these permanent by updating the initial values in the boot-area and saving the result to disk as an executable program. The boot-up parameters extend from '`0 +ORIGIN`' and supply an initial value for all of the user area. This is the image for the *user area* .

In ciforth the bootup parameters are more or less the data area belonging to the `+ORIGIN` word. Executing '`0 +ORIGIN`' leaves a pointer in this area.

4.6.2 Installation Dependent Code

`KEY` `EMIT` `KEY?` `TYPE`

`CR` `BLOCK-READ` and `BLOCK-WRITE`

are indeed different for different I/O models. This is of little concern to you as a user, because these are perfectly normal dictionary entries and the different implementations serves to make them behave similarly. There will however be more differences between the different configurations for ciforth for these words than habitually. These definitions are often revectored especially those for output. Output is revectored using `TYPE` . In other Forth's this is mostly done via `KEY` and `CR` separately. Input revectoring cannot be done via `KEY` . Redirection works and is easier most of the time.

4.6.3 Machine Code Definitions

The machine executable code definitions play an important role because they convert your computer into a standard Forth stack computer. It is clear that although you can define words by other words, you will hit a lowest level. The *code word* 's as these lowest level programs are called, execute machine code directly, if you invoke them from the terminal or from some other definition. The other definitions, called *high level* code, ultimately execute a sequence of the machine executable code words. The Forth *inner interpreter* takes care that these code words are executed in turn.

In the assembler source (if you care to look at it) you will see that they are interspersed with the high level Forth definitions. In fact it is quite common to decide to rewrite a code definition in high level Forth, or the other way around. The *Library Addressable by Block* contains an assembler, to add code definitions that will blend in like they were written in the kernel. Such definitions are to be closely matched with your particular ciforth, and you must be aware which registers play which role in ciforth. This is documented in the assembler source of this ciforth that accompanies this distribution. Of course it is also a rich source of examples how to make assembler definitions.

It bears repeating: code words are perfectly normal dictionary entries.

Note: if you want to change this ciforth's assembler source to fit your needs, follow the instructions present in the source, assembling as well as linking instructions.

4.6.4 High-level Standard Definitions

The high level standard definitions add all the colon-definitions, user variables, constants, and variables that must be available in a "Forth stack computer" according to the ISO standard. They comprise the bulk of the system, enabling you to execute and compile from the terminal, execute and *load* code from disk to add definitions etc. Changes here may result in deviations from the standard, so you probably want to leave this area alone. The technique described for the next section, forget and recompile, is not always possible here because of circular references. That is in fact no problem with an assembler listing, but it is if you load Forth code.

Again standard definitions words are perfectly normal dictionary entries.

4.6.5 User definitions

The user definitions contain primarily definitions involving user interaction: compiling aids, finding, forgetting, listing, and number formatting. Some of these are fixed by the ISO standard too. In ciforth most of those facilitities are not available in the kernel, but from the library. This applies even to the ISO standard words from the '`TOOLS`' wordset like `DUMP` (show a memory area as numbers and text) and `.S` (show the data stack). You can `FORGET` part of the high-level and re-compile altered definitions from disc. Mostly this is a mistake, and to make sure you mean it, you must change `FENCE` to defeat a protection mechanism.

A number of entries that could easily be made loadable are integrated in the assembler source of this ciforth version. Instead of forgetting them, you can load your own version on top of the existing system and waste some space.

Again user definitions words are perfectly normal dictionary entries.

4.6.6 System Tools

The boundary between categories are vague. A system tools is contrary to a user tool, a larger set of cooperating words. A text editor and machine code assembler are the first tools normally available. In ciforth those facilities are mostly not available in the kernel, but from the library. For example, an assembler is not part of he kernel as delivered, but it is available after '`WANT ASSEMBLERi86`'. Beware! The assembler can only be loaded on top of a `CASE-SENSITIVE` system. It automatically loads the proper 32-bits version.

You can load a more elaborate assembler. See Chapter 5 [Assembler], page 25, Section Overview. They are among the first candidates to be integrated into your system by `SAVE-SYSTEM` .

An editor is not part of ciforth as delivered. Development in Linux uses the there available editors. Even without tools, code can be tested by piping it into Forth, then commanding Forth to look to the console, as follows :
'`(echo 1 LOAD; cat pascal.frt - )| lina32`'
Primitive and preliminary as this may seem, it has been used for quite substantial developments like the 80386 assembler.

More advanced is using Your Favorite Editor, followed by including files:
‘ `"vim mysrc.frt" SYSTEM` ’
‘ `"mysrc.frt" INCLUDED` ’
See Chapter 4 [Manual], page 7, Section Getting Started.
In an installed system you will put ‘`WANT OS-IMPORT INCLUDE`’ in your electives screen (5), and just type ‘`vim mysrc.frt`’ to edit a file, without leaving lina32 and load it with ‘`INCLUDE mysrc.frt`’

A Pentium 32 and a 8086 Forth assembler are available in `forth.lab`. They are loaded in accordance with the system that is run. The registers used by lina32 are called HIP, SPO, RPO and WOR. The mapping on actual processor registers is documented in the source.

It is essential that you regard lina32 as just a way to get started with Forth. Forth is an extensible language, and you can set it to your hand. But that also means that you must not hesitate to throw away parts of the system you don't like, and rebuilt them even in conflict with standards. Additions and changes must be planned and tested at the usual Forth high level. Some words criticial for speed you can later rewrite as code words. Some words are easier to write in code right away.

Again words belonging to tools are perfectly normal dictionary entries.

4.6.7 RAM Workspace

The RAM workspace contains the compilation space for the dictionary, disc buffers, the computation and return stacks, the user area, and the console input buffer, From the fig-Forth user manual

> For a single user system, at least 2k bytes must be available above the compiled system (the dictionary). A 16k byte total system is most typical.

It is indeed possible to do useful work, like factoring numbers of a few hundred digits, in a workspace of 2k bytes. More typical a workspace is several megabytes to over hundred megabytes.

32 and 64 bits system are set at 64Mbyte but this is arbitrary and could be set much higher or lower without consequences for system load or whatever. Before long we will put the dictionary space on 32-bits Linux to 4G minus something and forget about this issue forever.

The boundary between this area and the previous ones is pretty sharp, it is where `DP` points. The other areas are more of a logical distinction. But even this boundary constantly changes as you add and forget definitions. Multi-tasking requires allocation of extra areas. See Chapter 4 [Manual], page 7, Section Details of memory layout.

4.7 Specific layouts

4.7.1 The layout of a dictionary entry

We will divide the dictionary in entries. A *dictionary entry* is a part of the dictionary that belongs to a specific word. A *dictionary entry address* , abbreviated *DEA* is a pointer to the header of a dictionary entry. In ciforth a header extends from the lowest address of the entry, where the code field is, to the *past header address* , just after the last field address. A *dictionary entry* apart from the header owns a part of the dictionary space that can extend before the header (mostly the name of the entry) and after it (mostly data and code).

A dictionary entry has fields, and the addresses of fields directly offset from the dictionary entry address, are called *field address* . This is a bit strange terminology, but it makes a distinction between those addresses and other addresses. For example, this allows to make the distinction between a *data field address* , that is always present, and a *data field* in the ISO sense that has only a (differing) meaning for `CREATE DOES>` definitions. Typically, a field address contains a pointer. A *data field address* contains a pointer to near the *data field* , whenever the latter exists.

They go from lowest in memory to highest:

1. The code field. This is one cell. A pointer to such a field is called a *code field address* . It contains the address of the code to be executed for this word.
2. The data field, of the DEA, not in the ISO sense. This is one cell. A pointer to such a field is called a *data field address* . It contains a pointer to an area owned by this definition.
3. The flag field. This is one cell. A pointer to such a field is called a *flag field address* . For the meaning of the bits of the flag field sea below.
4. The link field. This is one cell. A pointer to such a field is called a *link field address* . It contains the dictionary entry address of the last word that was defined in the same *word list* before this one.
5. The name field. This is one cell. This contains a pointer to a string. A pointer to such a field is called a *name field address* . The name itself is stored outside of the dictionary header in a regular string, i.e. a one cell count followed by as many characters, then padding for alignment. Unfortunately, *name token* is used in other Forth's to indicate a base to find other fields, what we call a *dictionary entry address*

   This came about because the name is lowest in memory. In this Forth the code field address and the dictionary address happens to be the same. This has a small advantage in *next* , it needs no offset.
6. The source field. This is one cell. This can be used to hold a reference to the source, a block number or a pointer to a string. For kernel words it stays at zero.
7. Past the header . This is actually not a field, but the free roaming dictionary. However, most of the time the part of the dictionary space owned by a dictionary entry starts here. A pointer to such a field is called a *past header address* address . Mostly a *data field address* contains a pointer to just this address.

The entries are not only in alphabetic order, they are in order of essentiality. They are accessed by `>CFA` `>DFA` `>FFA` `>LFA` `>NFA` `>SFA` . `(CREATE)` takes care to generate the dictionary entry data structure; it is called by all defining words.

Note that *data field* has a specific meaning in the ISO standard. It is accessed through `>BODY` from the *execution token* while a data field address is accessed through `>DFA` from the *dictionary entry address* . It is in fact one cell behind where the *data field address* pointer points to. Furthermore only particular words have data fields, those defined by `CREATE` .

The flag bits used in the kernel are:

- The INVISIBLE bit = 1 when *smudge* d; this will prevent a match by `(FIND)` .
- The IMMEDIATE bit = 1 for IMMEDIATE definitions; it is called the *immediate bit* .
- The DUMMY bit = 1 for a dictionary header contained in the data of a namespace; this indicates that it should not be executed.
- The DENOTATION bit = 1 for a prefix word. This means that it is a short word used as a prefix that can parse all *denotation* 's (numbers) that start with that prefix, e.g. 7 or & . Usually it is a one character word, but not necessarily. All built-in prefix words are part of the minimum search order and are one character.

After the last letter of a name follow zero bytes up till the next cell boundary. The *code field* of all *colon definition* 's contains a pointer to the same code, the *inner interpreter* , called '`DOCOL`'. For all words defined via '`CREATE ... DOES>`' the code field contains the same code, '`DODOES`'. On the other hand all *code definitions* (those written in assembler code) have different code fields.

At the *data field address* we find a pointer to an area with a length and content that depends on the type of the word.

- For a code word, it contains the same pointer as in the code field.
- For a word defined by `CONSTANT` , `VARIABLE` , `USER` , or `DATA` it has a width of one cell, and contains data. For `VARIABLE` it is a pointer to a cell, for `DATA` it is a pointer to a memory area of varying length.
- For all *colon definition* 's the data field address contains a pointer to an area of varying length. It contains the compiled high level code, a sequence of *dea* 's.
- For a word defined via '`CREATE ... DOES>`' the first cell of this area contains a pointer to the *high level* code defined by `DOES>` and the remainder is data. A pointer to the data is passed to this `DOES>` code.

The wordset '`DICTIONARY`' contains words for turning a *dictionary entry address* into any of these fields. They customarily start with `>`.

In summary, a dictionary falls apart into

1. Headers, with their fields.
2. Names, pointed to by some *name field address* .
3. Data, pointed to by some *data field address* . This includes high level code, that is merely data fed into the high level interpreter.
4. Code, pointed to by some *code field address* . This is directly executable machine code.

4.7.2 Details of memory layout

The disc buffers are mainly needed for source code that is fetched from disk were it resides in a file.

The disc buffer area is at the upper bound of RAM memory, So it ends at `EM` .

It is comprised of an integral number of buffers, each `B/BUF` bytes plus two cells. `B/BUF` is the number of bytes read from the disc in one go, originally thought of as one sector. In ciforth's `B/BUF` is always the size of one screen according to ISO : 1024 bytes. The constant `_FIRST` has the value of the address of the start of the first buffer. `_LIMIT` has the value of the first address beyond the top buffer. The distance between `_FIRST` and `_LIMIT` is a multiple of `B/BUF` bytes plus two cells.

For this ciforth the number of disk buffers is configured at 16 . The minimum possible is approximately 8 because nesting and locking requires that much blocks available at the same time.

The user area is configured to contain 0x40 cells, `MAX-USER`

contains the size of the area that is in use, in bytes. User variables can be added by the word `USER` , but you have to keep track yourself which offset in the user area can be used. Updating `MAX-USER` is recommended. The user area is just under the disc buffers. So it ends at `_FIRST` .

The console input buffer and the return stack share an area configured at a size of 0x10000 bytes. The lower half is intended for the console input buffer, and the higher part is used for the return stack, growing down from the end. The initial stack pointer is in variable `R0` . The return stack grows downward from the user area toward the terminal buffer.

The computation stack grows downward from the terminal buffer toward the dictionary which grows upward. The initial stack pointer is in variable `S0` .

During a cold start, the user variables are initialised from the bootup parameters to contain the addresses of the above memory assignments.

They can be changed. See Section 9.12.1 [+ORIGIN], page 99, for the bootup area. But take care. You probably need to study the source for how and when they take effect.

In multi-tasking a separate user area is allocated for each task, as well as a separate return stack area and a separate data stack area. A task that asks for input, also needs an extra console input buffer. A task is set up by allocating another area for all four. For task switching, it suffices to switch both stack pointers and the pointer to the user area. In this Forth the user area is moved during startup, such that it has a round address. This makes it possible to derive the user base pointer from the return stack pointer on the fly and handle one less register during task switching.

4.7.3 Terminal I/O and vectoring.

It is useful to be able to change the behaviour of I/O words such that they put their output to a different channel. For instance they must output to the printer instead of to the console. In general this is called *vectoring* . Remember that in normal Forth system, all printing of numbers is to the terminal, not to a file or even a buffer. (On a linux system something like it can be accomplished by the redirection facilities available.)

For this reason character output `CR` , `EMIT` and `TYPE` all go through a common word that can be changed. For lina32 it is `TYPE` . Because this is defined in high level code it can temporarily be replaced by other code. This *revectoring* is possible for all high level words in ciforth, such that we need no special measures to make *vectoring* possible. As an example we replace `TYPE` by `MYTYPE` .
‘`' MYTYPE >DFA @ ' TYPE >DFA !`’
And back to default:
‘`' TYPE >PHA ' TYPE >DFA !`’
Be careful not to define `MYTYPE` in terms of `TYPE` , as a recursive tangle will result. This method works in all versions of ciforth and is called *revectoring* .

A similar technique is not so useful on the input side, because keys entered during `ACCEPT` are subject to correction until <RET> has been pressed. On lina32 `ACCEPT` is left to the operating system, such that inputting to lina32 has the same look and feel as other input. Text can be pasted in with the mouse, etc. Consequently `RUBOUT` is not used. This is a limitation but input direction supplied by the operating system goes a long way to alleviate this.

4.8 Libraries and options

In ciforth there is no notion of object (i.e. compiled) libraries, only of source libraries. A Forth *library* is a block file adorned with one convention. This is that the words defined in a screen are mentioned on the first line of that screen, the *index line* . This is of course quite established as a habit. The word `WANTED` takes a string and loads the first screen where that name occurs in the index line. For convenience also `WANT` is there that looks ahead in the input stream. These words are not in the kernel but are present in screen 1, that corresponds to the ‘`-a`’ option.

Screen 0 and screen 1 to 31 are reserved for options, some of which are available to be filled in by the user.

When a Forth is started up with a first parameter that is a one-letter option, the corresponding screen is to be executed. So ‘`-a`’ or ‘`-A`’ is equivalent to ‘`1 LOAD`’ and ‘`-z`’ or ‘`-Z`’ is equivalent to ‘`26 LOAD`’. In fact all options are mapped onto screen 0..31 by a bitwise and.

4.8.1 Options

ciforth is a primitive system, and can interpret just one option on the command line. Moreover it can interpret options, only if it is properly installed with a connection to a `lab` file. If the first argument is not starting with `-` ciforth returns with error code 3. However the option ‘`-l`’ can bootstrap it into more sophisticated behaviour. The option `-x` triggers the code `^X LOAD` , loading screen 24.

The following options can be passed to lina32 on the command line:

- '-a'

  Make sure WANTED is available. This is a copy of the '-w' command because it is easier to remember '1 LOAD' if the screen must be loaded manually. In addition the signon message is suppressed.

- '-c name'

  Compile the file name to an executable binary. If name ends in '.frt' it is ommitted to arrive at the name of the binary, otherwise the binary is called a.out. Upon invocation of the binary the word defined latest is executed, then Forth goes BYE . name is a regular source file, not a block file. In addition WANT and ARG[] are made available. The source of the source file is temporarily stored in a kind of far PAD . The source is not part of the resulting binary, as would be the case if it is built with TURNKEY . A far PAD can cause problems if during compilation the program allocates a large amount of memory to touch that area; in that case use the -g option that make a larger Forth.

- '-d name ' Include name with [DEFINED] available. This file can be made to load onto other Forths without modifications.

- '-e'

  Load the elective screen, screen 5. This contains *preferences* , the tools you want to have available running an interactive Forth. The default library file contains system wide default preferences. See the '-l' option if the default preferences don't suite you. In an elective screen you just put commands to load or execute at startup of an interactive session, such as ' "fortune -f /usr/lib/forthcookies" SYSTEM ' or ' WANT EDIT '

- '-f forthcode'

  Execute the 'forthcode' in the order present. Beware of the special characters in the shell. Also the shell will collapse multiple spaces into one.

- '-g number name'

  Expand the system by 'number' Megabytes, then save it under the name 'name'. 'number' may be negative, and in that case the system is made smaller.

- '-h'

  Print overview of options.

- '-i binpath libpath [shellpath]'

  Install the forth in 'binpath' and the library in 'libpath' . If the 'shellpath' parameter is specified, it will be installed as the command interpreter used for SYSTEM . All of them must be full path names, not just directories. The ciforth that is running is copied to binpath, and the block file is copied to libpath.

  For system wide installation on a modern large system the following is recommended:

  ```
  su
  ./lina -g 60 lina+
  ./lina+ -i /usr/bin/lina /usr/lib/forth.lab
  chmod 755 /usr/bin/lina
  chmod 644 /usr/lib/forth.lab
  ```

  For a smallish system you may expand by 0 Mbyte '-g 0'. If the system has no swap space, and less than 8 Mbyte of memory, use '-g -3', diminish from 4 to 1 Mbyte. Expanding to the full size of the available RAM does no harm, as Linux overcommits memory.

- '`--help`' '`--version`' '`--`' '`-m`'

  The first option letter is trimmed to 5 bits, and excess characters are ignored. Thus all options that start with `-` are mapped onto *m*. The result is the combination of -h and -v , so both help and version information is printed. This conforms to the FSF-conventions.

- '`-l name [more]`'

  Use a library '`name`'. Restart Forth with as a block file `name` and as options the remainder of the line shifted, such that '`-l name`' disappears and the next option becomes the first. A file specified via '`-l`' is opened for reading and writing. Options ara again handled as described in the beginning of this section. In this way options may be added or reconfigured for personal use.

  Note that the default file is opened for reading only.

- '`-n` '

  This is an option intended for newbies. It loads `AUTOLOAD WANT` and prints the stack after each command. Autoload means that if you type in an unknown Forth word, an attempt is made to load it from the library. You can see the *index lines* of the facilities that are loaded. The definition that you are compiling is interrupted and a jump is made over the space you are using for the new definition. That works for smallish definitions, but not for complete facilities like floating point or an assembler. Thus this is for convenience only, and not absolutely reliable. It may succeed right away, or you have to repeat the command you give. If it fails, you can resort to '`WANT <word>`'. After development you should never rely on `AUTOLOAD` but paste the lines with `WANT` in your source. They are printed for convenience.

- '`-p`'

  Reserved option, not implemented.
  Be pedantic about ISO. Redefine some words to follow the standard as closely as possible.

- '`-q`'

  Be quiet, don't give a startup message.

- '`-s script`'

  Load the file `script` , but ignore its first line. This is intended to be used for Linux scripts, i.e. a piece of code to be interpreted rather than compiled. The first line is probably '`#!lina -s`' or some such as '`#!lina -l /usr/lib/forth/cgi/forth.lab -s`'. In a script `WANT` and `ARG[]` are available and you can use standard in and standard out. This follows the Unix conventions for script files. If you set the execute bit the file file becomes a command and accepts arguments.

- '`-v`'

  Print version and copy right information.

- '`-w`'

  Make sure `WANTED` is available.

- '`-?`'

  Give help, made to act the same as '`-h`'. The trimming makes that this is mapped to screen 31.

The remaining screens are available for options to be added at a later time, or for user defined options in a private library.

4.8.2 Private libraries

Working with source in files is quite comfortable using the default block library, especially if sufficient tools have been added to it. In principle all ISO words should be made available via `WANTED` .

In order to customize the forth library, you have to make a copy of the default `/usr/lib/ciforth/forth.lab` to your home directory, preferably to a lib subdirectory. Then you can start up using a '`-l`' option. You can also use the '`-i`' option to make a customized lina32 in your project directory. See Chapter 4 [Manual], page 7, Subsection Configuring.

Most shells allow you to redefine commands, such as e.g. in bash:
alias lina='lina -l $HOME/lib/forth.lab'

Note that the '`-l`' option hides itself, such that such an alias can be used completely identical to the original with respect to all options, including '`-l`'. Analysing arguments passed to lina32 in your programs can remain the same.

4.8.3 Stand alone programs.

A stand alone program is made relying on the word `SAVE-SYSTEM` . A stand alone program may be an enhanced version of ciforth or a program with a special purpose, what in other languages is normally called a compiled program. In Forth parlance this is called a *turnkey application* , because mostly the program has special requirements from the environment. They are made using the word `TURNKEY` . They take a word, that is to be done, and a string with the file name. Unlike other Forths, the resulting program is a regular executable, that can be run on other computers with the same operating system. Generally an application ignores the absence of a library file. It must make sure to `CATCH` any possible errors and report them in terms of the situation the user is in. Otherwise the user will be confronted with raw Forth error numbers without even the one line description.

Mostly it is much easier to just use the '`-c`' option. For a simple example See Chapter 4 [Manual], page 7, Getting Started Subsection Hello World!

5 Assembler

The `ciasdis` assembler is described in this manual, because the assembler that is in the lab file is compatible. The idea is that you test code with ciasdis relying on comprehensive consistency checking. Then can you use the assembler in the blocks and can the real debugging begin. This chapter is about the assembler itself, the information about what registers are used in ciforth is contained in its assemblers source.

5.1 Introduction

Via '`http://home.hccnet.nl/a.w.m.van.der.horst/forthassembler.html`' you can find a couple of assemblers, to complement the generic ciforth system. The assemblers are not part of the lina32 package, and must be fetched separately.

They are based on the postit/fixup principle, an original and novel design to accommodate reverse engineering. The assembler that is present in the blocks, is code compatible, but is less sophisticated, especially regards error detection. This assembler is automatically loaded in its 16 or a 32 bit form, such that it is appropriate for adding small code definitions to the system at hand. The background information given here applies equally to that assembler.

A useful technique is to develop code using the full assembler. Then with code that at least contains valid instruction enter the debugging phase with the assembler from the library.

The following files comprise the great assembler.
`ass.frt` : the 80-line 8086 assembler (no error detection), a prototype.
`as6809s.frt` : a small 6809 assembler (no error detection).
`asgen.frt` : generic part of postit/fixup assembler
`as80.frt` : 8080 assembler, requires `asgen.frt`
`asi86.frt` : 8086 assembler, requires `asgen.frt`
`asi386.frt` : 80386 assembler, requires `asgen.frt`
`aspentium.frt` : general Pentium non-386 instructions, requires `asgen.frt`
`asalpha.frt` : DEC Alpha assembler, requires `asgen.frt`
`asi6809.frt` : 6809 assembler, requires `asgen.frt`
`ps.frt` : generate opcode sheets
`p0.asi386.ps` : first byte opcode for asi386 assembler
`p0F.asi386.ps` : two byte opcode for same that start with 0F.
`test.mak` : makefile, i.e. with targets for opcode sheets.

The relevant assembler present in

forth.lab is equivalent to asgen.frt plus asi386.frt plus asipentium.frt but without error detection.

The `asi386.frt` (containing the full 80386 instruction set) is in many respects non-compliant to Intel syntax. The instruction mnemonics are redesigned in behalf of reverse engineering. There is a one to one correspondence between mnemonics and machine instructions. In principle this would require a monumental amount of documentation, comparable to parts of Intel's architecture manuals. Not to mention the amount of work to check this. I circumvent this. Opcode sheets for this assembler are generated by tools automatically, and you can ask interactively how a particular instructions can be completed. This is a viable alternative to using manuals, if not more practical. (Of course someone has to write up the descriptions, I am happy Intel has done that.).

So look at my opcode sheets. If you think an instruction would be what you want, use `SHOW:` to find out how it is to be completed. If you are at all a bit familiar, most of the time you can understand what your options are. If not compare with an Intel opcode sheet, and look up the instruction that sits on the same place. If you don't understand them, you can still experiment in a Forth to find out.

The assembler in the Library Addressable by Blocks (block file) hasn't the advanced features of disassembly, completion and error detection. It is intended for incidental use, to speed up a crucial word. But the code is fully compatible, so you can develop using the full assembler.

5.2 Reliability

I skimped on write up. I didn't skimp on testing. All full assemblers, like `asi386.frt` and `aspentium.frt`, are tested in this way:

1. All instructions are generated. (Because this uses the same mechanism as checking during entry, it is most unlikely that you will get an instruction assembled that is not in this set.)
2. They are assembled.
3. They are disassembled again and compared with the original code, which must be the same.
4. They are disassembled by a different tool (e.g. GNU's objdump), and the output is compared with 3. This has been done manually, just once.

This leaves room for a defect of the following type: A valid instruction is rejected or has been totally overlooked.

But opcode maps reveal their Terra Incognita relentlessly. So I am quite confident to promise a bottle of good Irish whiskey to the first one to come up with a defect in this assembler.

The full set of instructions, with all operand combinations sit in a file for reference. This is all barring the 256-way '`SIB`' construction and prefixes, or combinations thereof. This would explode this approach to beyond the practical. Straightforward generation of all instructions is also not practical for the Alpha with 32K register combinations per instruction. This is solved by defining "interesting" registers that are used as examples and leaving out opcode-operand combinations with uninteresting registers.

5.3 Principle of operation

In making an assembler for the Pentium it turns out that the in-between-step of creation defining words for each type of assembly gets in the way. There are just too many of them.

MASM heavily overloads the instruction, in particular '`MOV`' . Once I used to criticise Intel because they had an unpleasant to use instruction set with '`MOV`' '`MVR`' and '`MVI`' for move instructions. In hindsight I find the use of different opcodes correct. (I mean they are really different instructions, it might have been better if they weren't. But an assembler must live up to the truth.) Where the Intel folks really go overboard is with the disambiguation of essentially ambiguous constructs, by things as '`OFFSET`' '`BYTE POINTER`' '`ASSUME`' . You can no longer find out what the instruction means by itself.
A simple example to illustrate this problem is

```
        INC [BX]
```

Are we to increment the byte or the word at BX? Intel's solution is '`INC BYTE POINTER BX`') The INC instruction in this (the mod/rm) incarnation has a size bit. Here we require that this bit be filled in explicitly, either with ' `X|` ' or ' `B|` '). Failing to do so is a fatal error. This results in the rule: if an instruction doesn't determine the operand size (some do, like `LEA,`), then a size fixup is needed: '' `X|` or '' `B|` .
In this assembler this looks like

```
        INC, B| ZO| [BX]
```

This completely unambiguously determines the actual machine code.

These are the phases in which this assembler handles an instruction:

- POSTIT phase: `MOV,` assembles a two byte instruction with holes.
- FIXUP phase: `X|` or `B|` fits in one of the holes left. Other fixups determine registers and addressing mode.
- COMMA phase: First check whether the fixups have filled up all holes. Then add addresses (or offsets) and/or immediate data, using e.g. `IL,` or `L,`
- Check whether all commaers, requested either by postit's or fixup's are present. This check is actually executed by the next postit prior to assembling, or by `END-CODE` .

Doesn't this system lay a burden on the programmer? Yes. He has to know exactly what he is doing. But assembly programming is dancing on a rope. The Intel syntax tries to hide from you were the rope is. A bad idea. There is no such thing as assembly programming for dummies.

An advantage is that you are more aware of what instructions are there. Because you see the duplicates.

Now if you are serious, you have to study the `asgen.frt` and `as80.frt` sources. You better get your feet wet with `as80.frt` before you attack the Pentium. '`SIB`' is handled as an instruction within an instruction, clever, but hard to understand. It deviates somewhat from the phases explained here.

Another invention in this assembler is the *family of instructions* . Assembler instructions are grouped into families with identical fixups, and an increment for the opcodes. These are defined as a group by a single execution of a defining word. For each group there is one opportunity to get the opcode wrong; formerly that was for each opcode.

5.4 The 8080 assembler

The 8080 assembler doesn't take less place than Cassady's . (In the end the postit-fixup makes the Pentium assembler more compact, but not the 8080.) But... The regularities are much more apparent. It is much more difficult to make a mistake with the code for the '`ADD`' and '`ADI`' instructions. This principle allows to make a disassembler that is independant of the instruction information, one that will work for the 8086. A typical family are the 8 immediate- operand instructions, with an increment of 08.

```
08 C6 8 1FAMILY, ADI ACI SUI SBI ANI XRI ORI CPI
```

The bottom line is : the assembler proper now takes 22 lines of code. Furthermore the "call conditional" and "return conditional" instructions where missing. This became apparent as soon as I printed the opcode sheets. For me this means turning "jump conditional" into a family.

5.5 Opcode sheets

The makefile for the assembler project contain facilities to generate opcode sheets directly from the instruction sets, such as `asi386.ps`.. For the opcode sheets featuring a n-byte prefix you must pass the '`PREFIX`' to make and a '`MASK`' that covers the prefix and the byte opcode, e.g. '`make asi386.ps MASK=FFFF PREFIX=0F`' The opcode sheets `p0.asi386.ps` and `p0F.asi386.ps` are already part of the distribution and can be printed on a PostScript printer or viewed with e.g. '`gv`'.

Compare the opcode sheets with Intel's to get an overview of what I have done to the instruction set. In essence I have re-engineered it to make it reverse assemblable, i.e. from a disassembly you can regenerate the machine code. This is **not** true for Intel's instruction set, e.g. Intel has the same opcode for '`MOV, X| T| AX'| R| BX|` ' and '`MOV, X| F| BX'| R| AX|`'.

To get a reminder of what instructions there are type `SHOW-OPCODES` . If you are a bit familiar with the opcodes you are almost there. For if you want to know what the precise instruction format of e.g. `IMUL|AD,` just type '`SHOW: IMUL|AD,`' You can also type `SHOW-ALL,` but that takes a lot of time and is more intended for test purposes. The most useful of them all is `??`

that for a partially completed instruction shows all possible completions.

5.6 Details about the 80386 instructions

Read the introductory comment of `asgen.frt` for how the assembler keeps track of the state, using the `BI BY BA` tallies.

1. A word ending in `,` is an "opcode" and reserves place in the dictionary. It stand for one assembler instruction. The start of the instruction is kept and there is a bitfield (the tally) for all bits that belong to the instruction, if only mentally. These bits are put as comment in front of the instruction and they are considered filled in. The opcode also determines the instruction length.
2. A fixup mostly ends in `|`. It `OR` s in some bits in an already assembled instruction. Again there is a mask in front of fixups and in using the fixup these bits are considered to be filled in. A fixup cannot touch data before the start of the latest instruction. Some addressing modes fixups do not have `|` in them. This is in order to adhere more closely to conventions regarding those addressing modes. This much can be said. You can be sure that a word containing *[* and/or *]* is a fixup, that it is addressing mode related and that the addressing is indirect.
3. Families can be constructed from instructions or fixups with the same tally bit fields, provided the instructions differ by a fixed increment. The tallies also contain information about data and addresses following. These fields must be the same too.
4. The part before a possible `|` in an instruction – but excluding an optional trailing I – is the opcode. Opcodes define indeed a same action.
5. The part after `|` in an instruction may be considered a built in fixup where irregularity forbids to use a real fixup. A X stands for xell or natural data width. This is 16 bit for a 16 bit assembler and 32 bit for a 32 bit assembler. These can be overruled with `AS:,` applying to `DX|` and `MEM|` and with `OS:,` applying to data required where there is an I suffix. The commaers always reveal their true width. It is either `IW,` or `IL,` .
6. Width fixups determine the data width : `X|`

   (xell or natural data width 16/32) or `B|` (8 bit) unless implied. Offset fixups determine the offset or address width : `XO|`

   (xell or natural data width 16/32) or `BO|` (8 bit) or `ZO|` .
7. Instruction ending in *I* have an immediate data field after all fixups. This can be either `IB, IW, IL, IQ,` (8 16 32 64 bit). If there are width fixups they should correspond with the data.
8. Instructions ending in '`|SG`' builtin fixup (segments) require `SG,` (which is always 16 bits). For Xells in the presence of width overrules, the programmer should carefully insert `W,` or `L,` whatever appropriate.
9. With r/m you can have offsets (for `BO|` and `XO|`) that must be assembled using `B,` or `L,` but mind the previous point.
10. If an instructions with r/m has one register, it is always the target, i.e. it is modified.

11. Instruction with r/m can have a register instead of memory indicated by the normal fixups `AX|` etc.
12. If instructions with r/m have two registers, the second one is indicated by a prime such as `AX'|` . Stated differently, if an instruction can handle two general registers, the one that cannot be replaced by a memory reference gets a prime.
13. If `T|` or `F|` are present they apply to the primed register. `T|` "to" means that the primed register is modified. Absent those the primed register is the one that is modified. e.g. in `LEA,`
14. At the start of an instruction the mask of the previous instruction plus fixup should add up non-overlappingly to a full field. Offsets and immediate data should have been comma-ed in in order as required. This is diagnosed in the great assembler.
15. Instructions ending in ' `:,` ' are prefixes and are considered in their own right. They have no fixups.
16. The Scaled Index Byte is handled internally in the following way: The fixup `SIB|` closes the previous instruction (i.e. fill up its bit field), but possible immediate data and offsets are kept. Then `SIB,` starts a new instruction. The user merely needs to use a fixup with an unbalanced opening square bracket such as `[AX` , that handles this transparently.
17. The `SET,` instruction unfortunately requires a duplicate of the `O|` etc. fixups of the `J,` and `J|X,` instructions, called `O'|` etc. It
18. Similarly, some single byte instructions require `X'|` and `B'|` instead of `X|` and `B|` that are used for the ubiquitous instructions with r/m. (FIXME! This probably is remedied in the first release of ciasdis.)

This is the way the disassembler works.

1. Find the first instruction that agrees with the data at the program counter. Tally the bits. The instructions length follows from the instruction. As does the presence of address offsets and immediate data. In the current implementation the search follows dictionary links. The dictionary must be organized such that the correct instruction is found first. If two instructions agree with the data, in general the one that covers the most bits must be found first.
2. Find the first fixup that agrees with untallied bits. Note that opcode and previous fixups may have set bits in the `BAD` variable. Any fixups that set a bit in `BAD` that would result in a conflict are not considered.
3. If not all bits have been tallied go to 2, searching the dictionary from where we left off
4. Disassemble the address offsets and immediate data, in accordance with the instruction. Length is determined from fixups and prefix bytes. The commaers that were used to assemble the data have an associated execution token to disassemble the data. This is used to advantage to change the representation from program counter relative to absolute, or look up and show the name for a label.

5.7 16, 32 and 64 bits code and segments

The buildin assembler would be cumbersome to use, without the help of `ciasdis`, the great assembler. Not only are the instructions checked as explained before, from version 2.0.0 on, the interplay between segment size as instruction size are checked.

In fixup `X` is used to mean Xell, or the natural word length. This is 16 bits for 16 bits segments, 32 bits for 32 bits segments and 64 bits for 64 bits segments Likewise in PostIt-FixUp `AX` means Intel's `AX` for 16 bits segments, `EAX` for 32 bits segments and `RAX` for 64 bits segments.

The description of 16 or 32 bits in the Intel manuals is messy. These are the rules.

1. In real mode all sizes are 16 bits.

2. In protected mode the size of an address or Xell offset agrees with the size of the code segment.
3. In protected mode the size of an immediate data Xell agrees with the size of the applicable data segment. Mostly this is the data segment, but it may be the stack segment or some extra segment in the presence of segment override prefixes.
4. In all previous cases the code length can be swapped between 16 and 32 bits by a code length override prefix `OS:` , the data length by a data length override prefix `AS:` ,

The 16 bit indexing in a 32 bit assembler have separate fixup's, that all end in a `%`-sign.

In comma-ing, you must always select the proper one, commaers contain either *C*, *W*, *L* or *Q* for 1, 2, 4 or 8 byte widths.

After the directive `BITS-16` code is generated for and checked against 16 bit code and data segments. After the directive `BITS-32` code is generated for and checked against 32 bit code and data segments. After the directive `BITS-64` code is generated for and checked against 64 bit code and data segments.

In a 16 bits segments the following commaers must be used: `W, IW, (RW,)` and `RW,` .

In a 32 bits segments the following commaers must be used: `L, IL, (RL,)` and `RL,` .

In a 64 bits segments the following commaers must be used: `L, IL, (RL,)` and `RL,` and occasionally `IQ,` .

The prefix `OS:` switches the following opcode to use `IL,` instead of `IW,` and vice versa. Similarly the prefix `AS:` switches between `W,` and `L,` , or between `RW,` and `RL,` .

While mixing modes, whenever you get error messages and you are sure you know better than the assembler, put `!TALLY` before the word that gives the error messages. This will *override the error detection* . Proper use of the BITS-xx directives makes this largely unnecessary, but it can be needed if you use e.g. an extra segment `ES|` that is 16 bits in an otherwise 32 bits environment.

In 64 bits mode instructions that contain an immediate address differ from 32 bits mode. Those addresses are specified relative to the program counter, not absolute. Consequently the `MEM|` fixup leads to an error message, and instead `REL|` must be used with either a `RL,` or an `(RL,)` commaer. Absolute 64-bits addresses are nowhere present in the instruction set, as they are not really useful.

The great assembler enforces all these rules.

AMD took advantage of the fact that Intel instruction are available in a short and long form, e.g. `INC|X,` and `INC, X|` . The short form is hijacked, so `DEC|X, AX|` becomes `REX,` . All immediate data and offsets are sign-extended from 32 to 64 bits in 64 bits code, with the rational that full 64 bit is rarely useful. The result is that 32 and 64 code looks the same. In the rare case that a 64 bit value is needed, `MOVI|X,` is hijacked and replaced with `MOVI|Q,` . (Remember `MOVI, X|` is a duplicate.) So only instructions involving ghost registers representing integers and memory storage are different between 32 and 64 bits. That is all, and a 64-bit assembler is practically accommodated in full. Bottom line, the assembler built in into forth.lab is adequate to assemble the floating point wordset.

We need the 64-bit related prefix 0x48 to force the size to 64 bit in all cases where a register is mentioned in the instruction. Floating point instructions don't use regular registers and need not use this prefix unless e.g. `[SP` is used. The three least significant bits in the 0x4# switch the registers (possible in three positions) to the ghost registers. Such prefixes are present in `ciasdis`, but in the lab file only `REX,` is available.

There us more to say about *ghost* registers in using `ciasdis` itself. They appear instead of the regular registers, e.g `AX` is turned into `R8` . We make a distinction between instruction with possibly two register operands, and the others. The first class is called modr/m in Intel and

AMD lit. A two operand instruction always has a primary register that has a prime like `AX'|` and `T| F|` apply to that register. (The other operand may be a register, or indirect such as sib or memory address.) If you learned the distinction and use of primed and unprimed registers it is easy:

1. `'` applies to primed registers, turns `AX'|` to `R8'|`
2. `]` applies to index registers in sib-intructions, turns `AX']` to `R8']`
3. `N` bit applies to all other registers:

The remaining case of use of registers are

1. - unprimed register like `BX|` .
2. - indirect like `[BX]` .
3. - base register in sib like `[AX` .

In summary we get `Q: QN: Q]: QN]: Q': QN': Q']: QN']:` for possible prefixes, that switch at the same time to 64 bits. Similar prefixes are available with E , if you want the 32 bit ghost registers.

Note that an unprimed register cannot be combined with sib (scaled indexing) in any way, which would signify conflict between `AX|` and `[AX` .

Note that most assemblers would conflate `MOV, MOVI, MOVI|X,` etc, instructions and would not allow for such an easy explanation.

5.8 The built in assembler

From within ciforth one can load an assembler from the installed LAB library by the command `WANT ASSEMBLERi86` . Automatically a 32 bit assembler is loaded if the Forth itself is 32/64 bits and a 16 bit assembler for the 16 bit forths. This is a simplified version with no error checking and no provisions for 16/32 bit mixing. (Those are not needed, because you can mix with impunity.) This assembler is now (since 5.0.0) fully compatible with the large file-based one.

Consequently you can take a debugged program and run it through the LAB assembler.

The built in assembler has no error checking.

IMPORTANT NOTE: The 5.169 version and later may contain assembler code in the LAB file that has not yet been converted. This code largely relates to a booting version; It will be updated as soon as I have a booting version in a binary form available.

5.9 A rant about redundancy

You could complain about redundancy in postit-fixup assemblers. But there is an advantage to that, it helps detect invalid combinations of instructions parts. They look bad at first sight. What about
'`MOV, B| T| [BX+SI] R| AX|`'
'`MOV,`' needs two operands but there is no primary operand in sight. `[BX+SI]` would not qualify. and not even `BX|` because the primary operand should be marked with a prime.
'`MOV, X| T| BX| AX|`' looks bad because you know `BX|` and `AX|` work on the same bit fields, so it easy to remember you need the prime. `T|` and `F|` refer to the primary operands, so gone is the endless confusion about what is the destination of the move.
'`MOV, X| T| BX'| R| AL'|` looks bad , because `AL|` could not possibly qualify as an X register.
'`MOV, X| T| BX'| AX|`' looks bad , because soon you will adopt the habit that one of the 8 main register always must be preceeded with `T| F|` or `R|` .
'`MOV, X| T| BX'| R| AX|`' looks right but you still can code '`MOV, AX| BX'| R| T| X|`' if you prefer your fixups in alphabetic order. (A nice rule for those Code Standard Police out there?).

And yes ‘`ES: OS: MOV, X| T| DI’| XO| [BP +8* AX] FFFFF800 L,`’ though being correct, and in a logical order, looks still bad, because it **is** bad in the sense that the Pentium design got overboard in complication. (This example is from the built-in assembler, the one in `asi386.frt` redefines `[BP` c.s. to get rid of the `SIB|,` instruction.)
First remark: lets assume this is 32 bit code,(because otherwise there would not be a `SIB,` sure?) There are 3 sizes involved :

- The size of the data transported this is always the ‘`X`’ as in `X|` . Then the first `X|` changes its meaning to 16 bit, because of the `OS:` prefix.
- The fixups related to address offsets `XO|` and `L,` must agree and are 32 bits because you are in a 32 bits segment and this was not be overridden.
- The offset (in ‘`+AX]`’) is counted in 64 bits. Apparently, the ‘`DI`’ is fetched from two cell records.

And .. by the way the data is placed in the extra segment. Add a bit of awareness of the cost of the instructions in execution time and take care of the difference between the Pentium processors MMX en III and what not and you will see that assembly program is not for the faint of heart. The ‘`ASSUME`’ of the MASM assembler buys you nothing, but inconvenience.

5.10 Reference opcodes, Intel 386

Table one contains all the opcodes used in `asi386.frt` in alphabetic order, with `|` sorted before any letter. The opcodes that lift the assembler to the level of the Pentium is separately in table 3, in order not to make the tables overly long. All opcodes on the first position are the same as Intel opcodes, barring the bar. Note that sometimes parts that are integrated in the opcodes in Intel mnemonics are a separate fixup in the Postit-Fixup assembler. Examples are the condition codes in jumps.

You can use it in two ways.

- You want the opcode for some known Intel opcode.
  Look it up in the first column. One of the opcodes on that line is what you want. To pick the right one, consider the extension that are explained in table 2. Exception: ‘`PUSHI`’ is not on the line with ‘`PUSH`’ . Some times you have to trim built in size designators, e.g. you look up ‘`LODSW`’ but you are stuck at `LODS` , so that’s it. With ‘ `SHOW: LODS,` ’ you can see what the operands look like.
- You want to know what a POSIT/FIXUP code does. Look it up in the table, on the first word on the line you should recognize an Intel opcode. For example you have `CALLFAROI,`

  That is at the line with `CALL,` . So the combination of operands for `CALLFAROI,` are to be found in the description for ‘`CALL`’ in the Intel manuals.

Note. Some things are ugly. `LDS,` should be `L|DS,` . I would replace `MOV|FA,` by `STA,` and `MOV|TA,` by `LDA,` . But that would make the cross referencing more problematic. Note. The meaning of the operands for ‘`JMP`’ and ‘`JMPFAR`’ are totally different. So my suffices are different.

Table 1. Opcode cross reference.

AAA,

AAD,

AAM,

AAS,

ADC, ADCI, ADCI|A, ADCSI,
ADD, ADDI, ADDI|A, ADDSI,
AND, ANDI, ANDI|A, ANDSI,
ARPL,

AS:,

BOUND,

BSF,

BSR,

BT, BTI,

BTC, BTCI,
BTR, BTRI,
BTS, BTSI,
CALL, CALLFAR, CALLFAROI, CALLO,
CBW,

CLC,

CLD,

CLI,

CLTS,

CMC,

CMP, CMPI, CMPI|A,
CMPS, CMPSI,
CPUID,

CS:,

CWD,

DAA,

DAS,

DEC, DEC|X,
DIV|AD,

DS:,

ENTER,

ES:,

FS:,

GS:,

HLT,

IDIV|AD,

IMUL, IMUL|AD, IMULI, IMULSI,
INC, INC|X,
INS,

INT, INT3, INTO,
IN|D, IN|P,
IRET,

J, J|X, (Intel Jcc)
JCXZ,

JMP, JMPFAR, JMPFAROI, JMPO, JMPS,
LAHF,

LAR,

LDS,

LEA,

LEAVE,

LES,

LFS,

LGDT,

LGS,

LIDT,

LLDT,

LMSW,

LOCK,

LODS,

LOOP, LOOPNZ, LOOPZ,
LSL,

LSS,

LTR,

MOV, MOV|CD, MOV|FA, MOV|SG, MOV|TA,
MOVI, MOVI|B, MOVI|X,
MOVS,

MOVSX|B, MOVSX|W,
MOVZX|B, MOVZX|W,
MUL|AD,

NEG,

NOT,

OR, ORI, ORI|A, ORSI,
OS:,

OUTS,

OUT|D, OUT|P,
POP, POP|ALL, POP|DS, POP|ES, POP|FS, POP|GS, POP|SS, POP|X,
POPF,

PUSH, PUSH|ALL, PUSH|CS, PUSH|DS, PUSH|ES, PUSH|FS, PUSH|GS, PUSH|SS,
PUSH|X,
PUSHF,

PUSHI|B, PUSHI|X,
RCL, RCLI,
RCR, RCRI,
REPNZ,

REPZ,

RET+, RET, RETFAR+, RETFAR,
ROL, ROLI,
ROR, RORI,
SAHF,

SAR, SARI,
SBB, SBBI, SBBI|A, SBBSI,
SCAS,

SET, (Intel SETcc)
SGDT,

SHL, SHLI,
SHLD|C, SHLDI,
SHR, SHRI,
SHRD|C, SHRDI,
SIDT,

SLDT,

SMSW,

SS:,

STC,

STD,

STI,

STOS,

STR,

SUB, SUBI, SUBI|A, SUBSI,
TEST, TESTI, TESTI|A,
VERR,

VERW,

WAIT,

XCHG,

XCHG|AX,
XLAT,

XOR, XORI, XORI|A, XORSI,
~SIB,

Table 2 Suffixes, not separated by a |

I : Immediate operand
SI : Sign extended immediate operand
FAR : Far (sometimes combined with OI)
O : Operand
OI : Operand indirect

5.11 Reference opcodes, Pentium only.

Table three contains all the opcodes present in `asipentium.frt` in alphabetic order, with `|` sorted before any letter. All opcodes on the first position are the same as Intel opcodes, barring the bar. Note that again sometimes parts that are integrated in the opcodes in Intel mnemonics are a separate fixup in the Postit-Fixup assembler.

You can use it in the same way as the Intel 386 table. But there are much less instances where the opcodes do not agree exactly with Intels. Memory operands are specified in the same way for floating point instructions. But in those instructions register operands are always floating point registers.

There is at most one register specified in a floating point instruction. For two register operation `ST0` is always implicit. In that case normally it is the first operand as per '`ST0-ST1`'. '`a|`' (abnormal operation) means `ST0`

is the second operand as per '`ST1-ST0`'. Also normally `ST0` gets the result. '`m|`' (modified) means that the explicit register gets modified instead.

And don't forget! '`SHOW: <opcode>`' is your friend.

Table 3. Opcode cross reference. Pentium-only.

BSWAP,

CMPXCHG,
CMPXCHG8B,
F2XM1,

FABS,

FADD,

FADDP,

FBLD,

FBSTP,

FCHS,

FCLEX,

FCOM,

FCOMP,

FCOMPP,

FCOS,

FDECSTP,
FDIV,

FDIVP,

FFREE,

FIADD,

FICOM,

FICOMP,

FIDIV,

FILD, FILD|64,
FIMUL,

FINCSTP,
FINIT,

FIST,

FISTP, *FISTP|64*,
FISUB,

FLD, *FLD|e*,
FLD1,

FLDCW,

FLDENV,

FLDL2E,

FLDL2T,

FLDLG2,

FLDLN2,

FLDPI,

FLDZ,

FMUL,

FMULP,

FNOP,

FPATAN,

FPREM,

FPREM1,

FPTAN,

FRNDINT,
FRSTOR,

FSAVE,

FSCALE,

FSIN,

FSINCOS,
FSQRT,

FST, *FST|u*,
FSTCW,

FSTENV,

FSTP, *FSTP|e*, *FSTP|u*,
FSTSW,

FSTSW|AX,
FSUB,

FSUBP,

FTST,

FUCOM,

FUCOMP,
FUCOMPP,
FXAM,

FXCH,

FXTRACT,
FYL2X,

FYL2XP1,
INVD,

INVLPG,

Illegal-1,

Illegal-2,

RDMSR,

RDTSC,

RSM,

WBINVD,
WRMSR,

XADD,

The fixups for floating point are in lower case to make some distinction with the regular instructions. There is one fixup that conflicts with an uppercase fixup: `n|` . Table 4 Fixups and their meanings, Pentium-only.

ST0| : Register name
ST1|

ST2|

ST3|

ST4|

ST5|

ST6|

ST7|

s| : Single (16 bit)
d| : Double (32 bit)
m| : Explicit register is modified
u| : Explicit is unmodied, result to ST0
n| : ST0 is first operand (normal)
a| : ST0 is second operand (abnormal)
|16 : Int width in memory.
|32 : Int width in memory.

5.12 The dreaded SIB byte

If you ask for the operands of a memory instruction (one of the simple ones is LGDT,) instead of all the *scaled index byte*

(*SIB*) possibilities you see. '`LGDT, BO| ~SIB| 14 SIB,, 18, B,`' This loads the general description table from an address described by a sib-byte of 14 and an offset of 18.

The '~SIB| 14 SIB,,' may be replaced by any sib-specification of the kind '[AX +2* SI]'. You can ask for a reminder of the 256 possibilities by 'SHOW: ~SIB,'

The SIB constituents are not normal fixups. They must always appear between the normal fixups and the commaers, and the first must be the base register, the one with opening bracket, such as [AX .

Error-prone as that may seem, the great assembler only accepts correct instructions. Instructions are verbose, but they are hard to misinterpret.

Table 3 SIB-byte fixups.

[AX : Base register
[CX : Base register
[DX : Base register
[BX : Base register
[SP : Base register
[BP : Base register
[MEM : Base memory
[SI : Base register
[DI : register
+*1* : Scale by 1 byte.*
+*2* : Scale by 2 bytes.*
+*4* : Scale by 4 bytes.*
+*8* : Scale by 8 bytes.*
AX] : Scaled index
CX] : Scaled index
DX] : Scaled index
BX] : Scaled index
0] : No index
BP] : Scaled index
SI] : Scaled index
DI] : Scaled index

For the curious:

Explanation of 'LGDT, BO| ~SIB| 10 SIB,, 14, B,' This way of specifying a sib-byte would be perfectly legal, had I not hidden those words. It shows what is going on: the instruction is completed by ~SIB| telling the assembler that a comma-er SIB,, is required.

Instead of the comma-er we use a ~SIB, instruction. This specifies in fact a one byte opcode with three fields examplified by '[AX +2* SI'] (and again you might say '+2* SI] [AX' with the same meaning.) At the same time it is a comma-er in the sense that it reports that the demand for a sib-commaer is fill filled.

Many subtleties are involved to get right the error detection and the disassembly.

5.13 An incomplete and irregular guide to the instruction mnemonics.

The following is an attempted overview of the suffixes and fixup's used. It may be of some help for using the assembler because it gives some idea of some of the names. It doesn't contain all mnemonics, you have to consult an Intel manual anyway, just a few of them that I find hard to remember.

It also doesn't contain all fixup's, only those that are particularly hard or irregular. Neither does it contain fixups that are part of a *SIB* byte (treated elsewhere).

So beware!

Note that some of the instruction are Pentium and as yet not present in the asi386.frt.

Be careful with fixups that end in a % (such as `[BP+IS]%` . They are to be used in incidental 16 bits code, so in 16 bits code segments or for instructions preceeded by an address size overwrite prefix.

The *primed registers* have a prime after the register name such as `AX'|` , compared to `AX|` . Some opcodes allow two operands and then always one of them is a primed register. Whether the primed register is a source or destination is explicitly covered by `T|` and `F|` , **not** by any order in which the operands appear.

The primed conditions such as `Z'|` have a different reason. Those cannot be the same as the unprimed ones, because they occur at a different place in the opcode, though I would prefer them to be.

Some instructions
CPUID: CPU Identification
L : Load Full Pointer
LLDT: Load Local Descriptor Table Register
LGDT: Load General Descriptor Table Register
LIDT: Load Interrupt Descriptor Table Register
LTR: Load Task Register
LMSW: Load Machine Status Word
RDTSC: Read from Time Stamp Counter
RDMSR: Read from Model Specific Register
SHLD: Double Precision Shift Left
SHRD: Double Precision Shift Right
SLDT: Store Local Descriptor Table Register
SMSW: Store Machine Status Word
VERR: Verify a Segment for Reading or Writing
WRMSR: Write to Model Specific Register

Suffixes of the opcode, i.e. part of the opcode word.
|ALL : All
|CD : Control/Debug register
|FS : Replaces FS| in irregular opcodes.
|GS : Replaces GS| in irregular opcodes.
|AD : Implicit A and Double result.
|C : Implicit C (count)

Items in Fixups.
Y| : Yes, Use the condition straight
N| : No, Use the condition inverted
O| : Overflow
C| : Carry
Z| : Zero
CZ| : C || Z (unsigned <=)
S| : Sign (<0)
P| : Parity (even)
L| : S != O (signed <)
LE| : L || Z (signed <=)
T| : To (primed or special register)
F| : From (primed or special register)
V| : Variable number (in shifts)
1| : Just shift by 1.
ZO| : Zero Offset

BO| : Byte Offset
XO| : Xell Offset
Items in Commaers. Note that in commaers, there is never an *X*. You always have to choose between *W* for 16 bits and *L* for 32 bits or *Q* for 64 bits.
OW, Obligatory word
(RL,) Cell relative to IP
(RW,) Cell relative to IP
(RB,) Byte relative to IP
SG, Segment: word
P, Port number : byte
IS, Single obligatory byte
IL, immediate data : cell
IW, immediate data : cell
IB, immediate data : byte
L, address/offset data : cell
W, address/offset data : cell
B, address/offset data : byte
SIB,, Scaled index byte, an instruction with in an instruction
OB, : Obligatory byte
OW, : Obligatory word (=16bits)

There are also `RB,` `RW,` `RL,` based on `(RB,)` `(RW,)` `(RL,)` . They comma in an amount relative to the program counter based on an absolute address, such that you can use labels. These are used preferably, and are made to appear in the disassemblies. Otherwise no labels could appear in disassemblies.

5.14 Assembler Errors

Errors are identified by a number. They are globally unique, so assembler error numbers do not overlap with other ciforth error numbers, or errors returned from operating system calls. Of course the error numbers are given in decimal, always.

The errors whose message starts with '`AS:`' are used by the PostIt FixUp assembler in the file `asgen.frt`. See Chapter 7 [Errors], page 51, for other errors.

- '`ciforth ERROR # 26 : AS: PREVIOUS INSTRUCTION INCOMPLETE`'

  You left holes in the instruction before the current one, i.e. one or more fixups like `X|` are missing. Or you forget to supply data required by the opcode like `OW,` . With `??` you can see what completions of your opcode are possible.
- '`ciforth ERROR # 27 : AS: INSTRUCTION PROHIBITED IRREGULARLY`'

  The instruction you try to assemble would have been legal, if Intel had not made an exception just for this combination. This situation is handled by special code, to issue just this error. (This is rare, most situations are handled by bad bits, resulting in different errors.)
- '`ciforth ERROR # 28 : AS: UNEXPECTED FIXUP/COMMAER`'

  You try to complete an opcode by fixup's (like `X|`) or comma-ers (like `OW,`) in a way that conflicts with what you specified earlier. So the fixup/comma-er word at which this error is detected conflicts with either the opcode, or one of the other fixups/comma-ers. For example specifying both a `SI'|` and a `DI'|` operand for a `LEA,` opcode.
- '`ciforth ERROR # 29 : AS: DUPLICATE FIXUP/UNEXPECTED COMMAER`'

  You try to complete an opcode by fixup's (like `X|`) or comma-ers (like `OW,`) in a way that conflicts with what you specified earlier. So the fixup/comma-er word at which this error is detected conflicts with either the opcode, or one of other fixups/comma-ers. FIXME This explanation is the same as the previous. For example `B|` (byte size) with a `LEA,` opcode .

- '`ciforth ERROR # 30 : AS: COMMAERS IN WRONG ORDER`'

  The opcode requires more than one data item to be comma-ed in, such as immediate data and an address. However you put them in the wrong order. Use `SHOW:` .

- '`ciforth ERROR # 31 : AS: DESIGN ERROR, INCOMPATIBLE MASK`'

  This signals an internal inconsistency in the assembler itself. If you are using an assembler supplied with ciforth, you can report this as a defect ("bug"). The remainder of this explanation is intended for the writers of assemblers. The bits that are filled in by an assembler word are outside of the area were it is supposed to fill bits in. The latter are specified separately by a mask.

- '`ciforth ERROR # 32 : AS: PREVIOUS OPCODE PLUS FIXUPS INCONSISTENT`'

  The total instruction with opcode, fixups and data is "bad". Somewhere there are parts that are conflicting. This may be another one of the irregularities of the Intel instruction set. Or the `BAD` data was preset with bits to indicate that you want to prohibit this instruction on this processor, because it is not implemented. Investigate `BAD` for two consecutive bits that are up, and inspect the meaning of each of the two bits.

6 Optimiser

6.1 Introduction

The optimiser is not yet part of ciforth, Computer Intelligence Forth. Remember originally this Forth is conceived is a basis for a Computer Intelligence that has insight in her own code. As a consequence the Forth is made as transparant and simple as possible, and the theory is that optimisation is made easier the more the language is self referential and simple. This optimizer is thus a touch stone whether these ideas work in practice. This chapter outlines the theory; a practical application is in the works. Different types of optimisations interfere and finding one's way through this certainly requires some heuristics. The bottom line is that an optimiser qualifies as an AI application in the old sense, a result of painstaking, errorprone effort.

6.1.1 Properties

A Forth system is a database of small programs. It is worthwhile to investigate what properties these small programs (words) might have. The flag field of a word allow to add this information to the header. A certain combination of flags allow a particular optimisation.

6.1.2 Definitions

An annihilator is a word that only deletes an item from the stack. Examples are DROP 2DROP NIP RDROP.

A juggler reorders the stack without adding or removing items. Examples are SWAP 2SWAP ROT.

A duplicator copies an item from the stack. Examples are DUP OVER 2DUP.

A sequence of high level code is called stable with respect to branching if there is no branching into or out of the sequence.

A sequence of high level code is called stable with respect to the return stack if it only pops, what it has pushed itself, and the stack is left with the same depth as before.

A sequence is called stable if it is stable with respect to anything that is relevant in the context, mostly with respect to everything.

6.1.3 Notations

In the following we will denote a stack effects as < N – M >. This means that N items are popped and replaced by M new items. So 2DROP has the effect of < 2 – 0 >. Pointy brackets are used to make a distinction with the usual stack effect notation.

6.1.4 Optimisations

Optimisations are manipulations on a program source, intermediate code or machine code to improve the speed of the resulting program. In other respect the result is inferior. Symbolic debugging – one of Forth's strong points – goes through the drain. (The name "optimisation" is a misnomer.)

- Folding.

  Constant folding is a well known technique in optimisation. It means that if an operator works on constants the result may be replaced by a constant that is calculated at compile time. In Forth we generalise this to folding. Folding refers to all words that can be replaced by simpler words in case they receive constant data on the stack.

- Reordering.

  Reordering is not so much an optimisation per se, but it allows other optimisations to kick in. As a rule of thumb constants are moved to the top of the stack, where they fall prey to folding. Reordering might also eliminate a juggler.

- Anihilation.

  Annihilation is the elimination of a whole sequence of operations. In Forth sometimes the result of a calculation is dropped. Depending on the properties of the calculation, the calculation itself can be removed. This type of annihilation is related to an annihilator. On closer analysis it appears that any "no store" sequence with a < N – 0 > stack effect can be replaced by N times DROP.

  Another type is related to conditional branching where the condition is known at compile time. Code known to be skipped is removed.

- Inlining.

  Inlining means replacing a Forth word with its constituents. This technique is very important in Forth, more so than in other languages, due to the small size of Forth words. Inlining is always a winner in speed, and mostly even also a winner with regard to space.

  Even more important is the fact that inlining allows folding to be applied across constituent words. This applies to high level and low level code alike.

  Inlining high level code is trivial. A further inlining stage replaces a high level definition that only calls code words, by a code definition which concatenates the code words.

6.1.5 Data collecting

In order to add introspective information to a Forth, in the first place the machine code words must be analysed, because ultimately everything is defined in terms of code words. For this purpose the code words are disassembled using a disassembler that allows to readily inspect the parts of a disassembled instruction. A Postit-Fixup assembler and disassembler is well suited.

- By inspecting register words in the disassembly, registers usage can be accumulated. This information is then added to the header.
- At the same time information about whether memory or I/O ports are accessed for read or write can be collected. It turns out to be useful make a difference between input and output side effects. Here the words to look out for are the MOVE and IN/OUT instructions, operations that access memory (in fact all operations that not target registers) and special instructions like the string operations on Pentia.
- Finally the stack effect can be deduced from the number of POP's and PUSH'es. And the use of the return stack can be marked, which mostly warrants a special treatment.

After all code words have been analysed, the stack effects and register usage can be concluded for all high level words. The stack effect of a high level words is the concatenation of the stack effect of its constituents. The register usage of a high level word is the logical or of the register usage of the constituents, as are its side effects.

There will no doubt be exceptions. It is wrong to cater for too many exceptional situation in such a heuristic tool. Instead, the exception are filled in by hand before the automated collection is started, it fills in only as yet unknown items. Of course it helps to have a simple and straightforward Forth to begin with.

6.1.6 Purpose

A \ci in general will use optimisation to generate a temporary definition that is much faster, and retain all the valuable partial information about words.

In normal non-AI applications, words are recursively replaced by faster words, and those eventually by code definitions. Meanwhile words that are no longer directly used in the final application are eliminated. For space conservation headers may be removed as well, provided in the application no dictionary lookup is needed.

6.2 Implementation

The following implementation notes apply to a 32 bits Pentium Forth where a full cell (4 bytes, 0..3) is reserved for the flags. They must be considered as an example. The information about a word, optimisation opportunities and stack effect, sits in the flag field. Whenever nothing is filled in in the flag field, it means unknown. This applies equally to the stack effect as to the optimisation flags.

6.2.1 Stack effects

The information about the stack effects sits in the byte 3 of the flag field. The highest nibble of this third byte applies to input. It is the number of stack items popped plus one. The lowest nibble thusly indicates the number of pushed items. 0 means an unknown (not yet analysed) stack effect. 0FH indicates a variable number of items.

The stack effect is added in three steps. For all low level words the stack effect is found by counting pops and pushes. Irregular stack effects are corrected as well as filled in for high level words. All high level stack effects are derived from the stack effect of their constituents.

Code words are analysed by disassembling the code that is pointed to by the code field to the first "next" code encountered. For each instruction the opcode, which is the first part of its disassembly, is looked up in a set of possible pop and push instructions.

Irregularities are just tabulated in the source code of the analyser.

Words are recognized by their code field. High level words are either created by ":" or by a "CREATE .. DOES>" construction . They are recognised by the code field containing DOCOL or DODOES respectively. For both the data field points to a chain of high level calls, i.e. a number of such calls possibly with inlined data and ending in a "(;)", the word compiled by ";". (The result of this "high level next" is to return control is returned to the word that called this one.) For a linear chain of calls the stack effect is calculated as follows:

- Start with a effect of < 0 – 0 >
- For each constituent
  Subtract the pops from the left (output) nibble. If the output nibble is negative, add its (absolute) value to inputs, and make it zero. Add the pushes to the left (output) nibble. (Correction by 11H is not yet done).

  The following exceptions to a linear chain have special treatment:
- LIT BRANCH'es and SKIP are followed by inline data that must be taken care off
- A BRANCH or 0BRANCH forward is always taken, analysing just one path through a definition: the shortest one. A more sophisticated way is to analyse all paths and conclude a variable outcome if it is not consistent or any of the paths contains a variable constituent.
- If the stack effect of a constituent is variable, the result is variable, overruling any other outcome
- If the stack effect of a constituent is unknown, the result is unknown, overruling any other outcome except variable.
- For a CREATE .. DOES> word the linear chain pointed to by the DOES> pointer is analysed. However the stack effect is initialised to < 0 – 1 > to reflect the passing of the data pointer to the DOES> part.
- '<SOME-WORD> EXECUTE has the stack effect of <SOME-WORD> . Other occurrences of EXECUTE lead to a variable stack effect. Lateron we will leave this to the optimiser, but at the stage of analysing the kernel this is useful, especially because all usage of EXECUTE in the kernel is of this type.
- '<SOME-WORD> CATCH has the stack effect of <SOME-WORD> plus an extra output. Other occurrences of CATCH lead to a variable stack effect. So a word is treated as if

exceptions do not occur. This is okay because the stack effect is not relevant in case of exceptions.

A high level word is recognised by its code field address containing DOCOL , i.e. the nesting routine for the interpreter. A CREATE .. DOES> word is detected by is code field address containing DODOES , i.e. the common code that starts up words defined by compiler extension. All other words are treated as code.

The whole of Forth is treated as follows:

- Fill in the exception
- Fill in the code words
- Sweep repeatedly through the dictionary, from early to latest: For each unknown stack effect, try to find it by discriminating between DODOES DOCOL and other words, Stop if no progress is made any more.

Hopefully everything is known now, but maybe we must add to the exceptions. And repeat the above process.

The notion of a simple sequence is one that doesn't reach to the stack outside what is defined within the sequence.

6.2.2 Optimisation classes

As has been pointed out, the optimisation class of a word is indicated by a bit set in the flags field. Each bit set to true opens up a particular opportunity for optimisation. Further a sequence has a certain class if each constituent has that class. For example, if one of the words called does a store, the sequence is found to do a store and the optimisations that would be allowed by "no stores" are blocked. So the optimisation class of a sequence is the logical or of the oc's of the constituents. This can be done efficiently by bit-wise or operations.

6.2.2.1 The no store bit.

The "no store" bit would better be named "no output side effect" bit. It indicates that the outside world doesn't change by executing this word. Again not that the stacks and internal registers are inside. Note that fetching from an input port has an output side effect, (as well as an input side effect.)

The following optimisation are opened up:

- In combination with an annihilator. If the output of a "no store" sequence is annihilated, the whole sequence and the annihilator may be left out. Example: BASE CELL+ XX NIP becomes XX
- In combination with a juggler. If the outputs of "no store" sequence are juggled, the sequences itself may be juggled, eliminating the juggler. Example: XX CELL+ BASE SWAP becomes BASE XX CELL+
- In combination with a duplicator. Again a sequence may be duplicated and the duplicator eliminated. This is not an optimisation, except for the duplication of constants. Going the other direction can be an optimisation. Two identical sequences with no output side effect can be replaced by one and a duplicator. Example: (for a recursive definition with stack effect < 1 – 1 > and no side effects) 12 RECURSE OVER RECURSE becomes 12 RECURSE 12 RECURSE (elimination duplicator)
  12 RECURSE 12 RECURSE becomes 12 RECURSE DUP. (introducing duplicator)

6.2.2.2 The no fetch property.

The "no fetch" bit would better be named "no input side effect" bit. It indicates that the outside world affects the outcome of this word. Input side effects are weaker than output side effects and the knowledge that they are absent allows less optimisation.

6.2.2.3 The no stack effect property.

The "no stack effect fetch" bit refers to absolute accesses of the stacks, i.e. where the data or return stack are not used as stacks. Typical examples are DEPTH and RSP. These words are rare but prevent considerable optimisation.

6.2.2.4 The no side effect property.

The combination of the "no store ", "no fetch " and "no stack effect " properties is quite common. Such a word is said to have the "no side effect" property. The combination allows substantially more optimisation than each alone. We will use the abbreviation NS for this important concept. Examples are CONSTANT's, VARIABLE's, operators like + or NEGATE, and all stack manipulations: jugglers, annihilator, duplicators.

NS-words are amenable to folding:

- If a NS-sequence has only constant inputs, it may be run at compile time. Its inputs and the code sequence may be replaced by the resulting constant outputs. Example: After "12 4 3 SWAP * +" is replaced by 24.
- If a NS-sequence has no inputs, it may be run at compile time and replaced by the resulting constant outputs. The difference with the preceeding example is that the sequence starts with 12 instead of *. Any literals are of course NS.

On closer inspection the second condition is equivalent to the first. It is the more easy one to implement.

6.2.2.5 Associativity.

An operator with two inputs and one output, so called "binary operators" can have, in addition to NS, the property of associativity. This refers to a situation where three operands are involved. Examples are OR and + . However **not F+** . In the following we will denote an associative operator by %. Associativity allows to replace the sequence "x % y %" with "x y % %" where it may be that "x y %" can be folded into a constant. Example: (assuming a 64-bit Forth) "CELL+ CELL+" is first inlined to "8 + 8 +" then associated to "8 8 + +" then folded to "16 +". Note that it is not necessary to look for other patterns, in view of other transformation that are done.

6.2.2.6 Short circuit evaluation.

Another optimisation aplicable to binary NS-operators is short circuit evaluation. This is the situation where the result is known, while only one of the operands is known, such as "FFFF AND" "FFFF OR" "0 +" "0 XOR" and "0 *". Some of these operations can be just dropped, while in other cases the result is known and the other operand (possibly non-constant) can be dropped.

6.2.3 Optimisation by recursive inlining

A word is optimized by first optimizing its constituents, then inlining the constituents and apply any optimisation opportunities like folding that open up.

In more detail we have the following steps:

- Check.

  First of all check whether the item has been optimised already. We do in fact a "depth first" optimisation, so the words lowest in the call hierarchy are optimised first. It is important to only attempt optimisation once. This cuts the recursion short.

- Recurse

  For all constituent words of this definition, do an optimisation, such as defined in these steps.

- Inline.

  Build up an executable sequence in the dictionary. Inline a constituents word, keeping track of all opportunities to optimise.

- Folding Try to build up a sequence of NS-words that starts with constants and where each word following doesn't consume more inputs than are available. Consequently the outputs are available as constants. (In the example program this can be done at the same time as the inlining. Maybe that is unwise.)

- Breakdown.

  When a sequence of NS-words breaks down, we have identified a sequence that can be run at compile time. This sequence is run, and removed from the output sequence. Then the output of the run is compiled, as a sequence of constants.

  A more sophisticated method guarantees that constants move to the top as late as possible, which is favourable for other optimisations. In behalf of this, before compiling the sequence of constant, the code that follows is inspected. If a sequence is found with a < 0 – 0 > effect, that sequence is placed in front of the constants. The sequence need not have any special properties, except for the weak "no stack side effect" property. If a sequence is found with a < N – 0 > effect and N is smaller than the number of constants, a sequence with a < 0 – 0 > is can be constructed by adding N of the constants in front of it. The N constants are added to the output sequence, followed by the "no stack side effect" sequence and the other, very first, constants.

- Special opportunities.

  After inlining the sequence is checked whether it allows special optimisations, by comparing it to a table of patterns. Examples are the associativity optimisation with a "operand % operand %" pattern, and the execute optimisation with a "literal EXECUTE" pattern. In a fashion similar to the inlining a new sequence is built up. If there was any improvement, a new folding step must be attempted.

- Replace.

  After inlining is finished, the sequence is now attached to the word we are optimizing to replace the original sequence. Maybe the original code is kept if no folding took place and/or the sequence is longer that a certain limit.

- Mark properties The current word is marked as optimised. Its stack effect and its optimization classes are derived from its constituents and added to the flags header.

6.2.4 Inlining and control words

In the following with control words we will indicate words like BRANCH DO and EXIT that affect the flow of the program.

With respect to control words the optimiser will have as a goal to ultimately only have a body of code that consists of basic blocks (Dragon book terminology) i.e. straight code ending in a branch or a conditional branch, where branches only end at a the start of a basic block.

Thereafter the code can be replaced by machine code as is found in the constituent words. This code is inspected by a peep hole optimiser, eliminating e.g. a push pop sequences. In exceptional cases Forth calls to high level code, may need to be inserted. This blocks probably most possibilities for further optimisation.

Originally a DO LOOP looks like (DO) OFFSET .... (LOOP) . A LEAVE returns to after (LOOP) by discarding return stack parameters. The OFFSET allows (DO) to find the return address. In order to optimise this DO LOOP was replaced by a (DO) ␣ ... 1 (+LOOP) 0BRANCH OFFSET UNLOOP. Here (+LOOP) leaves a flag. The new (+LOOP) has now normal inlinable word.

Now LEAVE must be replaced by a branch to UNLOOP.

DO OFFSET can be inlined to ␣ 2>R . The dummy return address is no longer used, and hopefully will be removed by further optimisation.

The EXIT must be replaced by a branch to the end of the word. Then this can be unlined as is.

Recursion represents a problem. Surely a recursive word without side effects can be optimised if the input is constant. The word RECURSE can be replaced by inlining the word itself, which may make sense if it triggers folding operations. Tail call replacement is easy enough. Replacing a recursive word by low level code is not straightforward because one of the constituent words, i.e. itself, is not available in machine code form. A recursive word without side effects may be memoized to advantage though.

7 Errors

Errors are uniquely identified by a number. The error code is the same as the `THROW` code. In other words the Forth exception system is used for errors. A ciforth error always displays the text "ciforth ERROR #" plus the error number, immediately and directly. Of course the error numbers are given in decimal, irrespective of `BASE` . This allows you to look up the error in the section "Error explanations". More specific problems are addressed in the section "Common Problems".

7.1 Error philosophy

If you know the error number issued by ciforth, the situation you are in is identified, and you can read an explanation in the next section. Preferably in addition to the number a *mnemonic message* is displayed. It is fetched from the *library file* . But this is not always possible, such is the nature of error situations. A mnemonic message has a size limited to 63 characters and is therefore seldomly a sufficient explanation.

A good error system gives additional specific information about the error. In a plain ciforth this is limited to the input line that generated the error. Via the library file you may install a more sophisticated error reporting, if available.

Within ciforth itself all error situation have their unique identification. You may issue errors yourself at your discretion using `THROW` or, preferably, `?ERROR` and use an error number with an applicable message. However, unless yours is a quick and dirty program, you are encouraged to use some other unique error number, and document it.

7.2 Common problems

7.2.1 Error 11 or 12 caused by lower case.

If you type a standard word like `words` in lower case, it will not be recognised, resulting in error 11. Similarly `' words` results in error 12. This is because the names as defined in the standard are in upper case and lina32 is *case sensitive* , i.e. the difference between lower and upper case is significant and only words that match in this respect too are found in the dictionary.

After '`1 LOAD`' or if started up using '`lina -a`' or '`lina -r`' you have `WANTED` and `WANT` available. You may now issue ' `WANT CASE-INSENSITIVE` ' and switch the system into case-insensitivity and back by issuing the words `CASE-INSENSITIVE` and `CASE-SENSITIVE` .

Case insensitivity applies to the words looked up in the dictionary, as well as digits in numbers, preventing the use of `BASE`

larger than 36.

7.2.2 Error 8 or only error numbers

If you get an error 8 as soon as you try to `LOAD` or `LIST` a screen or use an option, or if errors show up only as numbers without the mnemonic message, this is because you cannot access the library file. It may not be there, or it may not be at the expected place. ciforth contains a string `BLOCK-FILE` , that contains the name of the library file interpreter, with as a default `forth.lab`. If this is not correct you may change it as appropriate by e.g.

'`"/usrlib/ciforth/forth.lab" BLOCK-FILE $!` ' The library is accessible for read and write access and mnemonic message will be fetched from it, after you install it with '`2 BLOCK-INIT 1 WARNING !`'.

7.2.3 Error 8 while editing a screen

If after editing a screen, you get error 8, the screen has not been written to disk, because you have no write access for the library file. You must issue `DEVELOP` which reopens the library file in READ_WRITE mode. Normally this should be part of loading the `EDITOR` . It may be of course that you don't have privilege to write to the file. As non-privileged user you cannot edit the system-wide library file.

You may always edit and use a private copy of the library file. The '`-i`' options installs a copy of ciforth to wherever you want, and you can edit there. Or you can copy the official library file, and edit the copy, then use it by the '`-l`' option. See Chapter 4 [Manual], page 7, for how options work. The '`-l`' option itself works only if at least the official library file has been correctly installed.

7.3 Error explanations

This section shows the explanation of the errors in ascending order. In actual situations sometimes you may not see the part after the semi colon. If in this section an explanation is missing, this means that the error is given for reference only; the error cannot be generated by your lina32, but maybe by other version of ciforth or even a differently configured lina32. For example for a version without security you will never see error 1. If it says "not used", this means it is not used by any ciforth.

The errors whose message starts with '`AS:`' are used by the PostIt FixUp assembler in the file `asgen.frt`,(see Chapter 5 [Assembler], page 25).

Negative error numbers are those reported by Linux. If possible, mnemonic error messages are shown. An explanation of the error is available in the manuals only.

'`ciforth ERROR # -2 : No such file `'

is an example of a Linux message. .

Here are the error explanations.

- '`ciforth ERROR # XXX : ( NO TEXT MESSAGE AVAILABLE FOR THIS ERROR )`'

  This is the only messages that is common to more errors, anything goes at the place of XXX. It means that information about this error is not in the library, but the error number remains to identify the error. The error number is probably used by user programs and hopefully documented there. So you can allocate error numbers not yet in use, and use them to identify your error situations. You can add messages to the library, but errors outside of the range [-256 63] need an edit of the source, or regeneration using adapted values of M4_ERRORMIN M4_ERRORMAX .

- '`ciforth ERROR # 1 : EMPTY STACK`'

  The stack has underflowed. This is detected by `?STACK` at several places, in particular in `INTERPRET` after each word interpreted or compiled. There is ample slack, but malicious intent can crash the system before this is detected.

- '`ciforth ERROR # 2 : DICTIONARY FULL`'

  Not used.

- '`ciforth ERROR # 3 : FIRST ARGUMENT MUST BE OPTION`'

  If you pass arguments to ciforth, your first argument must be an option (such as -a), otherwise it doesn't know what to do with it.

- '`ciforth ERROR # 4 : ISN'T UNIQUE`'

  Not being unique is not so much an error as a warning. The word printed is the latest defined. A word with the same name exists already in the current search order.

- '`ciforth ERROR # 5 : EMPTY NAME FOR NEW DEFINITION`'

  An attempt is made to define a new word with an empty string for a name. This is detected by `(CREATE)` . All *defining word* can return this message. It is typically caused by using such a word at the end of a line.
- '`ciforth ERROR # 6 : DISK RANGE ?`'

  Reading to the terminal input buffer failed. The message is probably inappropriate.
- '`ciforth ERROR # 7 : FULL STACK/DICTIONARY FULL `'

  The stack has run into the dictionary. This can be caused by pushing too many items, but usually it must be interpreted as dictionary full. If you have enough room, you have passed a wrong value to `ALLOT` . This is detected at several places, in particular in `INTERPRET` after each word interpreted.
- '`ciforth ERROR # 8 : ERROR ACCESSING BLOCKS FROM MASS STORAGE`'

  An access to the Library Accessible by Block (screen aka block file) has failed. Or if you are an advanced user, and used the block system at your own discretion, it simply means that access to the blocks has failed.

  This is detected by `?DISK-ERROR` called from places where a disk access has occurred. It may be that the library file has not been properly installed. Check the content of `BLOCK-FILE` . You may not have the right to access it. Try to view the file. Normally the library file is opened read-only. If you want to edit it make sure to do `DEVELOP` in order to reopen it in read/write mode. Otherwise you get this message too.
- '`ciforth ERROR # 9 : UNRESOLVED FORWARD REFERENCE`'

  A word can be compiled before it is fully defined, with a standard idiom like `DEFER` or ciforth idiom `:F` . If it is still not fully defined when it is used, this error is issued.
- '`ciforth ERROR # 10 : NOT A WORD, NOR A NUMBER OR OTHER DENOTATION`'

  The string printed was not found in the dictionary as such, but its first part matches a *denotation* . The denotation word however rejected it as not properly formed. An example of this is a number containing some non-digit character, or the character denotation `&` followed by more than one character. It may also be a miss-spelled word that looks like a number, e.g. '`25WAP`' . Be aware that denotations may mask regular words. This will only happen with user-defined denotations. Built-in denotations are in the `ONLY` namespace, that can only be accessed last, because it ends the search order. Note that hex digits must be typed in uppercase, even if `"CASE-SENSITIVE"` is in effect. Error 10 may be caused by using lower case where upper case is expected, such as for ISO standard words. See the section "Common problems" in this chapter if you want to make ciforth case insensitive.
- '`ciforth ERROR # 11 : WORD IS NOT FOUND`'

  The string printed was not found in the dictionary. This error is detected by `'` (tick). This may be caused by using lower case where upper case is required for ISO standard words. See the section "Common problems" in this chapter if you want to make ciforth case insensitive.
- '`ciforth ERROR # 12 : NOT RECOGNIZED`'

  The string printed was not found in the dictionary, nor does it match a number, or some other denotation. This may be caused by using lower case where upper case is required for ISO standard words or for hex digits. See the section "Common problems" in this chapter if you want to make ciforth case insensitive.
- '`ciforth ERROR # 13 : ERROR, NO FURTHER INFORMATION`'

  This error is used temporarily, whenever there is need for an error message but there is not yet one assigned.
- '`ciforth ERROR # 14 : SAVE/RESTORE MUST RUN FROM FLOPPY`'

- ‘ciforth ERROR # 15 : CANNOT FIND WORD TO BE POSTPONED’

  The word following POSTPONE must be postponed, but it can’t be found in the search order.
- ‘ciforth ERROR # 16 : CANNOT FIND WORD TO BE COMPILED’

  The word following [COMPILE] must be postponed, but it can’t be found in the search order.
- ‘ciforth ERROR # 17 : COMPILATION ONLY, USE IN DEFINITION’

  This error is reported by ?COMP . You try to use a word that doesn’t work properly in interpret mode. This mostly refers to control words like IF and DO . If you want control words to work in interpret mode, use WANT -scripting- .
- ‘ciforth ERROR # 18 : EXECUTION ONLY’

  This error is reported by ?EXEC. . You try to use a word that doesn’t work properly in compile mode. You will not see this error, because all words in ciforth do.
- ‘ciforth ERROR # 19 : CONDITIONALS NOT PAIRED’

  This error is reported by ?PAIRS . You try to improperly use control words that pair up (like IF and THEN , or DO and LOOP)

  This detection mechanism makes it impossible to compile some constructions allowed by the ISO standard. You may disable this checking by NO-SECURITY and re-instate it by DO-SECURITY . You can compile even combination of DO and BRANCH controls after WANT -tricky-control-
- ‘ciforth ERROR # 20 : STACK UNBALANCE, STRUCTURE UNFINISHED?’

  This error is reported by ?CSP . It detects stack unbalance between : and ; , or wherever you choose to use the words !CSP and ?CSP . This means there is an error in the compiled code. This message is given also if during compilation you try to use data that is put on the stack before : . Instead of

  ‘<generatedata> : name LITERAL .... ;’

  use

  ‘<generatedata> : name [_ SWAP] LITERAL .... ; DROP’

  to keep the stack at the same depth.
- ‘ciforth ERROR # 21 : IN PROTECTED DICTIONARY’

  The word you are trying to FORGET is below the FENCE , such that forgetting is not allowed.
- ‘ciforth ERROR # 22 : USE ONLY WHEN LOADING’

  This error is reported by ?LOAD . You try to use a word that only works while loading from the BLOCK-FILE , in casu --> .
- ‘ciforth ERROR # 23 : OFF CURRENT EDITING SCREEN’
- ‘ciforth ERROR # 24 : (WARNING) NOT PRESENT, THOUGH WANTED’ This error is reported by WANTED . The word you required, has been looked up in the index lines. It was not found in the index lines, or it was a dummy item, that only marks the screen to be loaded, e.g. ‘-scripting-’. In the latter case it can be safely ignored. This **must** be a warning only, because compilation can still succeed if the word is supplied by other means, in particular conditional compilation.
- ‘ciforth ERROR # 25 : LIST EXPECTS DECIMAL’

  This message is used by a redefined LIST , to prevent getting the wrong screen.
- ‘ciforth ERROR # 33 : INPUT EXHAUSTED’

  A parsing word doesn’t find the input it expects, even after REFILL .

- '`ciforth ERROR # 40 : REGRESSION TEST FAILS, STACK DEPTH ERROR`'

  This message is detected by `REGRESS` . It means that the number of stack items left by the test, doesn't agree with the number of items in the result specification.
- '`ciforth ERROR # 41 : REGRESSION TEST FAILS, RETURN VALUE ERROR`'

  This message is detected by `REGRESS` . It means that the stack items left by the test, don't agree with items in the result specification.
- '`ciforth ERROR # 42 : REGRESSION TEST MALL-FORMED, SECOND PART MISSING`'

  This message is given by `REGRESS` if there is no `S:` part.
- '`ciforth ERROR # 48 : NO BUFFER COULD BE FREED, ALL LOCKED`'

  While a block is in use by `THRU` , it is *locked* , which means that it must stay in memory. In addition blocks can be locked explicitly by `LOCK` . If a free block is needed, and there is no block that can be written back to the mass storage (disk or flash), you get this error.
- '`ciforth ERROR # 49 : EXECUTION OF EXTERNAL PROGRAM FAILED`' The word `SYSTEM` detected an error while trying to execute an external program.
- '`ciforth ERROR # 50 : NOT ENOUGH MEMORY FOR ALLOCATE`' The dynamic memory allocation could not allocate a buffer of the size wanted, because there is not enough consecutive memory available. Fragmentation can cause this to happen while there is more than that size available in total. This is detected by `ALLOCATE` or `RESIZE` .
- '`ciforth ERROR # 51 : UNKNOWN FORMAT IDENTIFIER`' This error is detected by the `FORMAT` wordset. The word following `%` in a format string, is not known. This means that it is not present in the *namespace* `FORMAT-WID` .
- '`ciforth ERROR # 52 : CANNOT HEAPIFY BUFFER`' This error is detected by the `ALLOCATE` wordset. The buffer you want to use as or add to the heap space, must be outside already existing heap space. This error results if you violate this rule. It may also result from corruption of the allocation system, such a writing outside designated space.

See Section 9.29.2 [ASSEMBLER], page 139,, for errors generated by the assembler. In general these have numbers that are higher than the general errors.

8 Documentation summary

The homepage of this Forth is

http://home.hccnet.nl/a.w.m.van.der.horst/lina.html

It is based on a generic system available via

http://home.hccnet.nl/a.w.m.van.der.horst/ciforth.html

All stable versions are copied to

https://github.com/albertvanderhorst/ciforth

The implementation of this Forth is indebted to FIGForth

http://home.hccnet.nl/a.w.m.van.der.horst/fig-Forth.html

A tutorial in English (and Dutch) is to be found at

https://forth.hcc.nl/w/Ciforth/Ciforth?setlang=en

The most important general Forth site is

http://www.forth.org

with links to all Forth chapters and commercial and free Forth implementation.

Like all modern languages the Forth ISO standard is available on the web

http://www.taygeta.com/forth/dpans.htm

The official, printed manual by the Amercian National Standards Institute commands a stiff price.

In print

- Starting forth by Leo Brodie

  A classic still worth reading, despite its age. You must adapt the examples in order to use it with an ISO Forth, A modernized version is available online at http://www.forth.com/starting-forth/
- Going Forth by Leo Brodie
  More timeless, maybe even more important, about the filosofy of Forth.
- The German Fig Chapter has a publication: Vierte Dimension.

For historic interest the following is copied from the FIG documentation 1978.

Caltech FORTH Manual, an advanced manual with internal details of Forth. Has Some implementation peculiarities. The Caltech Book Store, Pasadena, CA.

Kitt Peak Forth Primer, edited by the Forth Interest Group, P. O. Box 1105, San Carlos, CA 94070.

microFORTH Primer, Forth, Inc. 815 Manhattan Ave. Manhattan Beach, CA 90266

Forth Dimensions, newsletter of the Forth Interest Group, $5.00 for 6 issues including membership. F-I-G. P.O. Box 1105, San Carlos, CA. 94070

9 Glossary

Wherever it says single precision number or *cell* 32 bits is meant. Wherever it says *double* or "double precision number" a 64 bits number is meant.

The first line of each entry shows a symbolic description of the action of the proceedure on the parameter stack. The symbols indicate the order in which input parameters have been placed on the stack. The dashes "—" indicate the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right. Any symbol may be followed by a number to indicate different data items passed to or from a Forth word.

The symbols include:

'`addr`' memory address

'`b`' 8 bit byte (the remaining bits are zero)

'`c`' 7 bit ascii character (the remaining bits are zero)

'`d`' 64 bit signed double integer: most significant portion with sign on top of stack

'`dea`' A *dictionary entry address* , the basic address of a Forth word from which all its properties can be found.

'`f`' logical *flag* : zero is interpreted as false, non-zero as true

'`faraddr`' a <selector:address> pair

'`ff`' *Forth flag* , a well-formed logical flag, 0=false, -1=true.

'`false`' a false *Forth flag* : 0

'`n`' 32 bit signed integer number; it is also used for a 32-bit entity where it is irrelevant what number it represents

'`sc`' a *string constant* , i.e. two cells, an address and a length; length characters are present, starting at the address (they must not be changed)

'`true`' a true *Forth flag* : -1.

'`u`' 32-bit unsigned integer, also used whenever a cell is considered as a bitset.

'`ud`' 64-bit unsigned double integer: most significant portion on top of stack

The capital letters on the right show definition characteristics:

'`C`' May only be used within a colon definition.

'`E`' Intended for execution only.

'`FIG`' Belongs to the FIG model

'`I`' Has immediate bit set. Will execute even when compiling.

'`ISO`' Belongs to ISO standard

'`NFIG`' Word belongs to FIG standard, but the implementation is not quite conforming.

'`NISO`' Word belongs to ISO standard, but the implementation is not quite conforming.

'`P`' Word is a prefix, interprets and optionally compiles remainder of word.

'`WANT`' Word is not in the kernel, use the `WANT` to load it from the library. These words are maintained and tested, will only be changed with notice and an upgrade pad will be supplied.

'`U`' A user variable.

Where there is mention of a standard or a model, it means that the word actually complies to the standard or the model, not that some word of that name is present in that standard. Words marked with '`ISO,FIG`' will behave identically over all but the whole spectra of Forth's.

Unless otherwise noted, all references to numbers are for 32-bit signed integers. For 64-bit signed numbers, the most significant part (with the sign) is on top.

All arithmetic is implicitly 32-bit signed integer math, with error and under-flow indication unspecified.

A *nil pointer* is an address containing zero. This indicates an invalid address.

The Forth words are divided into *wordset* s, that contain words that logically belong together. Each wordset has a separate section with a description. The following rules take precedence over any wordset a word may logically belong to.

- A defining word — one that adds to the dictionary — is present in the wordset '`DEFINING`'.
- A denotation word – one that has the prefix bit set – is present in the wordset '`DENOTATIONS`'.
- An environmental query word — one that is understood by `?ENVIRONMENT` — is present in the wordset '`ENVIRONMENTS`'.

9.1 BLOCKS

The block mechanism connects to the Forth system a single background storage divided in numbered *blocks* . The wordset '`BLOCKS`' contains words to input and output to this mass storage. In this ciforth blocks reside in a file, by default named `forth.lab`. . Most blocks are used for the '`SCREEN`' facility, where each block contains source code.

9.1.1 #BUFF

Name: `#BUFF`

Stackeffect: — n

Attributes:

Description: A constant that leaves the number of block buffers. Because a buffer that is being interpreted is locked in memory, this is also a limit to the nesting depth of blocks loading other blocks.

See also: '`BLOCK`' '`THRU`' '`LOAD`' '`LOCK`'

9.1.2 –>

Name: `-->`

No stackeffect

Attributes: I,WANT

Description: Continue interpretation with the next disc screen. If the current input source is not from a block, a crash will ensue. If this new screen is left by throw of an exception, the screen may remain locked until a `QUIT` , or any uncaught exception.

See also: '`LOCK`' '`CATCH`' '`LOAD`'

9.1.3 ?DISK-ERROR

Name: `?DISK-ERROR`

Stackeffect: n—

Attributes:

Description: Interpret '`n`' as the status of a disk i/o call and *signal an error* if it contains an error condition. It is only used to signal errors related to accessing the `BLOCK-FILE` .

See also: '`BLOCK-FILE`' '`BLOCK-HANDLE`' '`BLOCK-INIT`' '`BLOCK-EXIT`'

9.1.4 B/BUF

Name: `B/BUF`

Stackeffect: — n

Attributes:

Description: This constant leaves the number of bytes per disc buffer, the byte count read from disc by `BLOCK` . The ISO standard fixes this to 1024.

See also: ‘`(BUFFER)`’

9.1.5 BLOCK-EXIT

Name: `BLOCK-EXIT`

Stackeffect: —

Attributes:

Description: A block file must have been opened by `BLOCK-INIT` . All blocks are unlocked. Any changed blocks are written back to mass storage. Close the currently open block file `BLOCK-HANDLE` , such that the mass storage words no longer work, and will result in error messages. If error messages were fetched from disk, they no longer are.

See also: ‘`BLOCK`’ ‘`LIST`’ ‘`LOAD`’ ‘`DISK-ERROR`’ ‘`WARNING`’

9.1.6 BLOCK-FILE

Name: `BLOCK-FILE`

Stackeffect: —addr

Attributes:

Description: Leave the address ‘`addr`’ of a counted string, the name of a *library file* in which *blocks* are (to be) allocated. The name may contain a path and be at most 252 characters long. The default name is **forth.lab** . This name is typically changed during installation and is used by the `BLOCK-INIT` command.

See also: ‘`BLOCK-HANDLE`’ ‘`BLOCK-INIT`’ ‘`LIST`’ ‘`LOAD`’ ‘`$@`’

9.1.7 BLOCK-HANDLE

Name: `BLOCK-HANDLE`

Stackeffect: —n

Attributes:

Description: Leave a file handle in ‘`n`’ . If it is negative there is no block file open, otherwise the handle is used by the system to access blocks.

See also: ‘`BLOCK-FILE`’ ‘`BLOCK-INIT`’ ‘`BLOCK-EXIT`’

9.1.8 BLOCK-INIT

Name: `BLOCK-INIT`

Stackeffect: n —

Attributes:

Description: Map the blocks on the block file `BLOCK-FILE` , i.e. the mass storage words refer to the blocks in this file. The handle ‘`BLOCK-HANDLE`’ can be used to access it, with access code ‘`n`’ (2 for read and write). This command signals failure by a negative handle in `BLOCK-HANDLE` . You must activate mnemonic error messages explicitly by setting `WARNING` .

See also: ‘`BLOCK`’ ‘`LIST`’ ‘`LOAD`’ ‘`BLOCK-EXIT`’ ‘`OPEN-FILE`’

9.1.9 BLOCK-READ

Name: `BLOCK-READ`

Stackeffect: addr blk —

Attributes:

Description: The ciforth primitive for reading of blocks. '`addr`' specifies the destination block buffer, '`blk`' is the sequential number of the referenced physical block `BLOCK-READ` determines the location on mass storage, performs the read and throws an exception on errors.

See also: '`BLOCK`' '`DISK-ERROR`' '`BLOCK-WRITE`' '`BLOCK-INIT`'

9.1.10 BLOCK-WRITE

Name: `BLOCK-WRITE`

Stackeffect: addr blk —

Attributes:

Description: The ciforth primitive for writing of blocks. '`addr`' specifies the source or destination block buffer, '`blk`' is the sequential number of the referenced physical block. `BLOCK-WRITE` determines the location on mass storage, performs the write and throws an exception on errors.

See also: '`BLOCK`' '`DISK-ERROR`' '`BLOCK-READ`' '`BLOCK-INIT`'

9.1.11 BLOCK

Name: `BLOCK`

Stackeffect: n — addr

Attributes: ISO,FIG

Description: Leave '`addr`', the disc buffer containing block '`n`', which is the physical disk block '`OFFSET+n`'. The address left is the field within the buffer to be used for data storage. If the block is not already in memory, it is transferred from disc to a new buffer allocated by `(BUFFER)` . Blocks are generally used to contain source code to be interpreted by `LOAD` . They can be equally useful to contain other data, e.g. for implementing a database.

See also: '`(BUFFER)`' '`BLOCK-READ`' '`BLOCK-WRITE`' '`OFFSET`' '`UPDATE`' '`FLUSH`' '`LOAD`'

9.1.12 DISK-ERROR

Name: `DISK-ERROR`

Stackeffect: — addr

Attributes:

Description:

This variable is not used in ci86.lina32.html, errors are thrown.

See also: '`BLOCK`'

9.1.13 EMPTY-BUFFERS

Name: `EMPTY-BUFFERS`

No stackeffect

Attributes: ISO,FIG

Description: Mark all block-buffers as empty. Updated blocks are not written to the disc. This is an initialization proceedure before first use of the disc. The usage as an "undo" is infeasible in ciforth.

See also: '`FLUSH`' '`BLOCK`' '`SCREEN`' '`UPDATE`'

9.1.14 LOCK

Name: `LOCK`

Stackeffect: n —

Attributes: CI

Description: Lock the buffer with identification '`n`', mostly a block number. Multiple locks are possible, and require multiple unlocks. Probably, it is to become the *current input source* . The result is that its buffer will not be reclaimed until an `UNLOCK` occurs.

See also: '`BLOCK`' '`LOAD`' '`(BUFFER)`' '`UNLOCK`' '`#BUFF`'

9.1.15 UNLOCK

Name: `UNLOCK`

Stackeffect: n —

Attributes: CI

Description: Unlock the buffer with identification '`n`', mostly a block number. Probably, because it is no longer the *current input source* . The result is that its buffer can again be reclaimed. Unlocking without a previous lock may lead to a crash.

See also: '`LOCK`' '`(BUFFER)`' '`LOAD`' '`#BUFF`'

9.1.16 UPDATE

Name: `UPDATE`

No stackeffect

Attributes: ISO,FIG

Description: Marks the most recently referenced block (pointed to by `_PREV`) as altered. The block will subsequently be transferred automatically to disc should its buffer be required for storage of a different block. In fact the block is transferred to disk immediately.

See also: '`BLOCK`' '`EMPTY-BUFFERS`'

9.1.17 (BUFFER)

Name: `(BUFFER)`

Stackeffect: n — addr

Attributes:

Description: Return the addres '`addr`' of a buffer assigned to identification '`n`' , probably a block number. Block numbers are positive, so a negative value can be used for a buffer that is used for some other purpose. The buffer layout is as follows: a cell with '`n`', a cell with the status, and the content of length *B/BUF* . The status is negative for locked. The l.s.b. gives zero for free and one for valid data. The block is not read from the disc. The buffer is either one that was already assigned, or else a free buffer. If there is none free, some non-locked buffer is freed. The contents of that buffer is written to the disc, if it was marked as updated. In ciforth this will never happen, because updated blocks are written immediately. In ciforth blocks can be locked, and locked buffers are never freed by `(BUFFER)` . An update flag would somehow be multiplexed with the lock count, but it is not needed in this ciforth. If all buffers were locked, `(BUFFER)` throws exception 48.

See also: '`BLOCK`' '`_PREV`' '`#BUFF`' '`LOCK`' '`UNLOCK`'

9.1.18 +BUF

Name: `+BUF`

Stackeffect: addr1 — addr2 ff

Attributes: FIG

Description: Advance the disc buffer address '`addr1`' to the address of the next buffer '`addr2`' . Boolean '`ff`' is false when '`addr2`' is the buffer presently pointed to by variable `_PREV` .

See also: '`BLOCK`'

9.1.19 BLOCK-SEEK

Name: `BLOCK-SEEK`

Stackeffect: n—

Attributes:

Description: A block file must have been opened by `BLOCK-INIT` . Position the file pointer at block '`n`' in behalf of subsequent reads and writes.

See also: '`BLOCK`' '`LIST`' '`LOAD`' '`BLOCK-READ`' '`BLOCK-WRITE`'

9.1.20 FLUSH

Name: `FLUSH`

No stackeffect

Attributes: ISO,FIG

Description: Make sure that the content of all `UPDATE` d block buffers has been transferred to disk. The buffers are no longer associated with a block and their content is no longer available. In ciforth no transfer takes place, because mass storage is updated automatically in the background.

See also: '`EMPTY-BUFFERS`' '`BLOCK`'

9.1.21 OFFSET

Name: `OFFSET`

Stackeffect: — addr

Attributes: U

Description: A user variable which contains a block offset. The contents of `OFFSET` is added to the argument for `BLOCK` before reading or writing blocks. In this way a part of disc drives can be reserved for boot-code.

As blocks are allocated in a file on the host operating system it is 0.

See also: '`BLOCK`' '`MESSAGE`' '`BLOCK-READ`' '`BLOCK-WRITE`'

9.1.22 _FIRST

Name: `_FIRST`

Stackeffect: — addr1

Attributes:

Description: A constant that leaves the address of the block buffer lowest in memory.

See also: '`BLOCK`' '`_LIMIT`'

9.1.23 _LIMIT

Name: `_LIMIT`

Stackeffect: —- addr1

Attributes:

Description: A constant leaving the address just above the highest memory available for a disc buffer. Actually this is the highest system memory.

See also: '`BLOCK`' '`_FIRST`'

9.1.24 _PREV

Name: `_PREV`

Stackeffect: — addr

Attributes:

Description: A variable containing the address of the disc buffer (not its content field!) most recently referenced. The `UPDATE` command marks this buffer to be written to disc.

See also: '`(BUFFER)`'

9.2 COMPILING

The wordset '`COMPILING`' contains words that compile words and numbers. They need special attention because these words in general execute during compilation See Section 9.6.14 [IMMEDIATE], page 84, . Numbers are compiled *in line* , behind a word that fetches them.

9.2.1 DLITERAL

Name: `DLITERAL`

Stackeffect: d — d (executing) d — (compiling)

Attributes: I

Description: If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack. If executing, the number will just remain on the stack.

See also: '`LITERAL`' '`LIT`' '`STATE`'

9.2.2 LITERAL

Name: `LITERAL`

Stackeffect: n — n (executing) n — (compiling)

Attributes: ISO,I,C

Description: If compiling, then compile the stack value '`n`' as a 32 bit literal. The intended use is: '`: xxx [ calculate ] LITERAL ;`' Compilation is suspended for the compile time calculation of a value. Compilation is resumed and `LITERAL` compiles this value. Later execution of the definition containing the literal will push it to the stack. If executing, the number will just remain on the stack.

See also: '`LIT`' '`SDLITERAL`' '`STATE`'

9.2.3 POSTPONE

Name: `POSTPONE`

No stackeffect

Attributes: ISO,I,C

Description: Used in a colon-definition in the form:

```
: xxx POSTPONE SOME-WORD
```

POSTPONE will postpone the compilation behaviour of 'SOME-WORD' to the definition being compiled. If 'SOME-WORD' is an immediate word this is similar to '[COMPILE] SOME-WORD'.

See also: '[COMPILE]'

9.2.4 [COMPILE]

Name: [COMPILE]

No stackeffect

Attributes: ISO,I,C

Description: Used in a colon-definition in the form:

```
:  xxx ... [COMPILE] IF ...  ;
```

[COMPILE] will force the compilation of an immediate definition, that would otherwise execute during compilation. The above example will perform IF when 'xxx' executes, rather than introducing conditional code in 'xxx' itself.

See also: 'POSTPONE'

9.2.5 LIT

Name: LIT

Stackeffect: — n

Attributes: FIG,C

Description: Within a colon-definition, LIT is compiled followed by a 32 bit literal number given during compilation. Later execution of LIT causes the contents of this next dictionary cell to be pushed to the stack. This word is compiled by LITERAL .

See also: 'LITERAL'

9.2.6 SDLITERAL

Name: SDLITERAL

Stackeffect: d — s/d (executing) / d — (compiling)

Attributes: I

Description: If compiling, compile a stack double number into a literal or double literal, depending on whether DPL contains a *nil pointer* or points into the input. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.

See also: 'LITERAL' 'DLITERAL' 'STATE'

9.3 CONTROL

The wordset 'CONTROL' contains words that influence the control flow of a program, i.e. the sequence in which commands are executed in compiled words. With control words you can have actions performed repeatedly, or depending on conditions.

9.3.1 +LOOP

Name: `+LOOP`

Stackeffect: n1 — (run) / addr n2 — (compile)

Attributes: ISO,I,C

Description: Used in a colon-definition in the form:

```
DO ... n1 +LOOP
```

At run-time, `+LOOP` selectively controls branching back to the corresponding `DO` based on '`n1`' , the loop index and the loop limit. The signed increment '`n1`' is added to the index and the total compared to the limit. The branch back to `DO` occurs until the new index is equal to or greater than the limit ('`n1>0`'), or until the new index is equal to or less than the limit ('`n1<0`'). Negative increments cannot be combined with `?DO` , this deviates from the ISO standard. Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile-time, `LOOP` compiles code to the above effect, existing of `(+LOOP) 0BRANCH UNLOOP`

and uses '`addr`' , left on the stack by `DO` , to calculate the branch offset from `HERE` .

'`n2`' is used for compile time error checking.

See also: '`LOOP`'

9.3.2 ?DO

Name: `?DO`

Stackeffect: n1 n2 — (execute) addr n — (compile)

Attributes: NISO,I,C

Description: Occurs in a colon-definition in form:

```
?DO ... LOOP
```

It behaves like `DO` , with the exception that if '`n1`' is less than '`n2`' the loop body is not executed. This is intended to suppress the unwanted behaviour of looping through the whole number range, ciforth deviates from ISO in that it also suppresses the unwanted behaviour of looping through almost the whole number range for an input of e.g.

```
1 2
```

However, negative increments are made impossible for forthword(?DO) this way.

See also: '`DO`' '`I`' '`LOOP`' '`+LOOP`' '`LEAVE`'

9.3.3 AGAIN

Name: `AGAIN`

Stackeffect: addr n — (compiling)

Attributes: ISO,FIG,I,C

Description: Used in a colon-definition in the form:

```
BEGIN ... AGAIN
```

At run-time, AGAIN forces execution to return to the corresponding BEGIN . There is no effect on the stack. Execution cannot leave this loop except for EXIT . At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. 'n' is used for compile-time error checking.

See also: 'BEGIN'

9.3.4 BEGIN

Name: BEGIN

Stackeffect: — addr n (compiling)

Attributes: ISO,FIG,I

Description: Occurs in a colon-definition in one of the forms:

```
BEGIN ... UNTIL
```

```
BEGIN ... AGAIN
```

```
BEGIN ... WHILE ... REPEAT
```

At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL , AGAIN or REPEAT . When executing UNTIL a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs.

At compile time BEGIN leaves its return address and 'n' for compiler error checking.

See also: '(BACK' 'DO'

9.3.5 CO

Name: CO

No stackeffect

Attributes:

Description: Return to the caller, suspending interpretation of the current definition, such that when the caller exits, this definition is resumed. The return stack must not be engaged, such as between >R and R> , or DO and LOOP .

See also: 'EXIT'

9.3.6 DO

Name: DO

Stackeffect: n1 n2 — (execute) addr n — (compile)

Attributes: ISO,FIG,I,C

Description: Occurs in a colon-definition in form: 'DO ... LOOP' At run time, DO begins a sequence with repetitive execution controlled by a loop limit 'n1' and an index with initial value 'n2' . DO removes these from the stack. Upon reaching LOOP the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after DO ; otherwise the loop parameters are discarded and execution continues ahead. Both 'n1' and 'n2' are determined at run-time and may be the result of other operations. Within a loop I will copy the current value of the index to the stack. With `+LOOP` it can be used with other increments than one.

It deviates from the ISO standard in that that if '`n1`' is equal to '`n2`' the loop is never executed even once.

When compiling within the colon definition, `DO` compiles `(DO)` and leaves the following address '`addr`' and '`n`' for later error checking.

See also: '`I`' '`LOOP`' '`+LOOP`' '`LEAVE`'

9.3.7 ELSE

Name: `ELSE`

Stackeffect: addr1 n1 — addr2 n2 (compiling)

Attributes: ISO,FIG,I,C

Description: Occurs within a colon-definition in the form:

```
IF ... ELSE ... THEN
```

At run-time, `ELSE` executes after the true part following `IF` . `ELSE` forces execution to skip over the following false part and resumes execution after the `THEN` . It has no stack effect.

At compile-time `ELSE` compiles `BRANCH` and reserves a places for a branch offset, leaving its address '`addr2`' and '`n2`' for error testing. `ELSE` also resolves the pending forward branch from `IF` by calculating the offset from '`addr1`' to `HERE` and storing at '`addr1`' .

See also: '`(FORWARD`' '`FORWARD)`' '`BRANCH`'

9.3.8 EXIT

Name: `EXIT`

No stackeffect

Attributes: ISO

Description: Stop interpretation of the current definition. The return stack must not be engaged, such as between `>R` and `R>` , or `DO` and `LOOP` . In ciforth it can also be used to terminate interpretation from a string, block or file, or a line from the current input stream.

See also: '`(;)`'

9.3.9 IF

Name: `IF`

Stackeffect: f — (run-time) / — addr n (compile)

Attributes: ISO,FIG,I,C

Description: Occurs in a colon-definition in form:

```
IF (tp) ...  THEN
```

or

```
IF (tp) ... ELSE (fp) ... THEN
```

At run-time, `IF` selects execution based on a boolean flag. If '`f`' is true (non-zero), execution continues ahead thru the true part. If '`f`' is false (zero), execution skips till just after `ELSE` to execute the false part. After either part, execution resumes after `THEN` . `ELSE` and its false part are optional; if missing, false execution skips to just after `THEN` .

At compile-time `IF` compiles `0BRANCH` and reserves space for an offset at '`addr`' . '`addr`' and '`n`' are used later for resolution of the offset and error testing.

See also: '`(FORWARD`' '`0BRANCH`'

9.3.10 I

Name: `I`

Stackeffect: — n

Attributes: ISO,FIG,C

Description: Used within a do-loop to copy the loop index to the stack.

See also: '`DO`' '`LOOP`' '`+LOOP`'

9.3.11 J

Name: `J`

Stackeffect: — n

Attributes: ISO,FIG,C

Description: Used within a nested do-loop to copy the loop index of the outer do-loop to the stack.

See also: '`DO`' '`LOOP`' '`+LOOP`'

9.3.12 LEAVE

Name: `LEAVE`

No stackeffect

Attributes: ISO

Description: Terminate a do-loop by branching to directly behind the end of a loop started by `DO` or `?DO` , so after the corresponding `LOOP` or `+LOOP` .

See also: '`UNLOOP`'

9.3.13 LOOP

Name: `LOOP`

Stackeffect: — (run) addr n — (compiling)

Attributes: ISO,I,C

Description: Occurs in a colon-definition in form:

```
DO ... LOOP
```

At run-time, `LOOP` selectively controls branching back to the corresponding `DO` based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to `DO` occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead.

At compile-time, `LOOP` compiles code to the above effect, existing of `ONE (+LOOP) 0BRANCH UNLOOP`

and uses '`addr`' to calculate an offset to be used by '`0BRANCH`' . '`n`' is used for compile time error checking.

See also: '`+LOOP`'

9.3.14 RECURSE

Name: `RECURSE`

Stackeffect: (varies)

Attributes: ISO

Description: Do a recursive call of the definition being compiled.

See also: '`:`'

9.3.15 REPEAT

Name: `REPEAT`

Stackeffect: addr1 n1 addr2 n2— (compiling)

Attributes: ISO,FIG,I,C

Description: Used within a colon-definition in the form:

```
BEGIN ... WHILE ... REPEAT
```

At run-time, `REPEAT` forces an unconditional branch back to just after the corresponding `BEGIN` .

At compile-time, `REPEAT` compiles `BRANCH` and the offset from `HERE` to '`addr2`' . Then it fills in another branch offset at '`addr1`' left there by `WHILE` . '`n1 n2`' is used for error testing.

See also: '`WHILE`'

9.3.16 SKIP

Name: `SKIP`

No stackeffect

Attributes: C

Description: Skip over an area in memory, where the length is given in the next cell, then align. This length doesn't include the length cell, so it is compatible with `$@` . Internal, used for nested compilation and compiling strings.

See also: '`BRANCH`'

9.3.17 THEN

Name: `THEN`

Stackeffect: addr n — (compile)

Attributes: ISO,FIG,I,C

Description: Occurs in a colon-definition in form:

```
IF ... THEN
```

```
IF ... ELSE ... THEN
```

At run-time, `THEN` serves only as the destination of a forward branch from `IF` or `ELSE` . It marks the conclusion of the conditional structure. At compile-time, `THEN` computes the forward branch offset from '`addr`' to `HERE` and stores it at '`addr`' . '`n`' is used for error tests.

See also: '`FORWARD)`' '`IF`' '`ELSE`'

9.3.18 UNLOOP

Name: `UNLOOP`

No stackeffect

Attributes: ISO,I,C

Description: Discard the loop parameters. Must be used when the regular end of the loop is by-passed. That means it is not ended via `LOOP` `+LOOP` or `LEAVE` , but my means of `EXIT` or unstructured branching. In this Forth the parameters are the address after the loop af .

See also: '`DO`' '`LEAVE`' '`0BRANCH`' '`(FORWARD`'

9.3.19 UNTIL

Name: `UNTIL`

Stackeffect: f — (run-time) / addr n — (compile)

Attributes: ISO,FIG,I,C

Description: Occurs within a colon-definition in the form:

```
BEGIN ... UNTIL
```

At run-time, `UNTIL` controls the conditional branch back to the corresponding `BEGIN` . If f is false, execution returns to just after `BEGIN` , otherwise execution continues ahead.

At compile-time, `UNTIL` compiles `0BRANCH` and an offset from `HERE` to addr. '`n`' is used for error tests.

See also: '`BEGIN`'

9.3.20 WHILE

Name: `WHILE`

Stackeffect: f — (run-time) / addr1 nl — addr2 n1 addr1 n2 (compile-time)

Attributes: ISO,FIG,I,C

Description: Occurs in a colon-definition in the form: '`BEGIN ... WHILE (tp) ... REPEAT`' At run-time, `WHILE` selects conditional execution based on boolean flag '`f`' . If '`f`' is true (non-zero), `WHILE` continues execution of the true part thru to `REPEAT` , which then branches back to `BEGIN` . If '`f`' is false (zero), execution skips to just after `REPEAT` , exiting the structure.

At compile time, `WHILE` compiles `0BRANCH` and tucks the target address '`addr2`' under the '`addr1`' left there by `BEGIN` . The stack values will be resolved by `REPEAT` . '`n1`' and '`n2`' provide checks for compiler security.

See also: '`(FORWARD`' '`BEGIN`'

9.3.21 (+LOOP)

Name: `(+LOOP)`

Stackeffect: n —

Attributes: C

Description: The run-time proceedure compiled by `+LOOP` , which increments the loop index by n and tests for loop completion.

See also: '`+LOOP`'

9.3.22 (;)

Name: `(;)`

No stackeffect

Attributes:

Description: This is a synonym for `EXIT` . It is the run-time word compiled at the end of a colon-definition which returns execution to the calling proceedure. Stop interpretation of the current definition. The return stack must not be engaged.

See also: '`EXIT`'

9.3.23 (?DO)

Name: `(?DO)`

No stackeffect

Attributes: C

Description: The run-time proceedure compiled by `?DO` which prepares the return stack, where the looping bookkeeping is kept.

See also: '`?DO`'

9.3.24 (BACK

Name: `(BACK`

Stackeffect: — addr

Attributes:

Description: Start a backward branch by leaving the target address `HERE` into '`addr`'. Usage is '`(BACK .. POSTPONE BRANCH BACK)` '

See also: '`BACK)`' '`BEGIN`' '`DO`'

9.3.25 (DO)

Name: `(DO)`

No stackeffect

Attributes: C

Description: The run-time proceedure compiled by `DO` which prepares the return stack, where the looping bookkeeping is kept.

See also: '`DO`' '`UNLOOP`'

9.3.26 (FORWARD

Name: `(FORWARD`

Stackeffect: — addr

Attributes:

Description: Start a forward branch by allocating space for an offset, that must be backpatched into '`addr`'. Usage is '`POSTPONE BRANCH (FORWARD .. FORWARD)` '

See also: '`IF`' '`BRANCH`' '`0BRANCH`'

9.3.27 0BRANCH

Name: `0BRANCH`

Stackeffect: f —

Attributes: FIG,C

Description: The run-time proceedure to conditionally branch. If 'f' is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF , UNTIL , and WHILE .

See also: 'BRANCH' '(FORWARD' '(BACK' 'SKIP '

9.3.28 BACK)

Name: BACK)

Stackeffect: addr —

Attributes:

Description: Complete a backward branch by compiling an offset from HERE to 'addr', left there by (BACK . Usage is '(BACK .. POSTPONE BRANCH BACK) '

See also: 'LOOP' 'UNTIL'

9.3.29 BRANCH

Name: BRANCH

No stackeffect

Attributes: FIG,C

Description: The run-time proceedure to unconditionally branch. An in-line offset is added to the interpretive pointer 'HIP' to branch ahead or back. BRANCH is compiled by ELSE AGAIN REPEAT .

See also: '0BRANCH' '(FORWARD' '(BACK '

9.3.30 FORWARD)

Name: FORWARD)

Stackeffect: addr —

Attributes:

Description: Complete a forward branch by backpatching an offset from HERE into 'addr', left there by (FORWARD . Usage is 'POSTPONE BRANCH (FORWARD .. FORWARD) '

See also: 'LOOP' 'UNTIL' 'REPEAT'

9.4 DEFINING

The wordset 'DEFINING' contains words that add new entries to the dictionary, or are related to those words. A number of such *defining word* 's are predefined, but there is also the possibility to make new defining words, using CREATE and DOES> .

9.4.1 ;

Name: ;

No stackeffect

Attributes: ISO,FIG,I,C

Description: Terminate a colon-definition and stop further compilation. Compiles the run-time (;) .

See also: ':'

9.4.2 CONSTANT

Name: `CONSTANT`

Stackeffect: n —

Attributes: ISO,FIG

Description: A defining word used in the form: '`n`' `CONSTANT` '`cccc`' to create word '`cccc`' , where the content of its *data field address* is '`n`' . When '`cccc`' is later executed, it will push the value of '`n`' to the stack.

See also: '`VARIABLE`' '`>DFA`'

9.4.3 CREATE

Name: `CREATE`

No stackeffect

Attributes: ISO

Description: A defining word used in the form: '`CREATE cccc`' Later execution of '`cccc`' returns its *data field* , i.e. the value of `HERE` immediately after executing `CREATE` .

It can be the base of a new defining word if used in the form:

```
: CREATOR CREATE aaaa DOES> bbbb ;
CREATOR cccc
```

The second line has the effect of creating a word '`cccc`' . Its datastructure is build by the code '`aaaa`' and when executing '`cccc`' , its data field is pushed on the stack, then the code '`bbbb`' is executed.

Space in this *data field* has yet to be allocated. The *DFA* (data field address) points to the the execution action that can be changed by `DOES>` . ciforth is byte aligned, so no extra measures are needed.

See also: '`>BODY`' '`DOES>`' '`;CODE`' '`ALLOT`' '`,`' '`>DFA`'

9.4.4 DATA

Name: `DATA`

No stackeffect

Attributes:

Description: A defining word used in the form: '`DATA cccc`' When `DATA` is executed, it creates the definition '`cccc`' whose *data field address* contains a pointer '`n`' to `HERE` . This code is typically followed by some data allocation word like `ALLOT` or `,` . `DOES>` must not be used with '`cccc`'.

When '`cccc`' is later executed, this pointer '`n`' is left on the stack, so that data can be accessed.

See also: '`VARIABLE`' '`CREATE`' '`>DFA`'

9.4.5 DOES>

Name: `DOES>`

No stackeffect

Attributes: ISO,FIG

Description: A word which is used in combination with `CREATE` to specify the run-time action within a high-level defining word. DOES> modifies the default behaviour of the created word so as to execute the sequence of compiled word addresses following DOES> . When the DOES> part executes it begins with the address of the *data field* of the word on the stack. This allows interpretation using this area or its contents. Typical uses include the Forth assembler, arrays and matrices, and compiler generation.

9.4.6 MAX-USER

Name: MAX-USER

Stackeffect: — addr

Attributes: U

Description: A user variable which contains the offset of next user variable. It is measured in cells from the start of the user area.

See also: 'BLOCK' 'USER' 'MESSAGE' 'U0'

9.4.7 NAMESPACE

Name: NAMESPACE

No stackeffect

Attributes:

Description: A defining word used in the form: NAMESPACE 'cccc' to create a namespace definition 'cccc' . It will create a *word list* in the ISO sense. Subsequent use of 'cccc' will push this word list (the *word list associated with* 'cccc') to the top of the search order in CONTEXT . So it will be searched first by INTERPRET . A word create by NAMESPACE is not immediate. This may differ from VOCABULARY that is presen in many Forth implementations.

A namespace 's data content field contains at first the dovoc pointer (like for any DOES> word) , then follows its body. The body contains the namespace ("vocabulary") link field address (*VLFA*). The VLFA points to the VLFA of the next namespace or a *nil pointer* for the end. Then follows a wordlist *word list identifier* or *WID* . Executing the namespace means pushing its WID on top of the CONTEXT order. In ciforth when there can be at most 16 word list 's in the search order, the oldest one gets lost.

See also: 'ALSO' 'PREVIOUS' 'WORDLIST' 'DEFINITIONS' 'FOR-VOCS' '>WID'

9.4.8 USER

Name: USER

Stackeffect: n —

Attributes: ISO

Description: A defining word used in the form: 'n USER cccc' which creates a user variable 'cccc' . The data field of 'cccc' contains 'n' as a byte offset relative to the user pointer register 'UP' for this user variable, which is 'n CELLS'. When 'cccc' is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable. In this ciforth 'UP' is coupled to the return stack. This means that with a switch of the return stack, 'UP' is switched also automatically. It is best to do 'MAX-USER @ DUP USER cccc 1 MAX-USER ! ' reflecting the current allocation in the user area.

See also: 'VARIABLE' '+ORIGIN' 'CONTEXT' '>DFA'

9.4.9 VARIABLE

Name: VARIABLE

No stackeffect

Attributes: ISO,FIG

Description: A defining word used in the form: '`VARIABLE cccc`' When `VARIABLE` is executed, it creates the definition '`cccc`' whose *data field address* contains a pointer '`n`' to a data location, that can contain one cell. When '`cccc`' is later executed, this pointer '`n`' is left on the stack, so that a fetch or store may access this location.

See also: '`USER`' '`CONSTANT`' '`>DFA`' '`DATA`'

9.4.10 WORDLIST

Name: `WORDLIST`

Stackeffect: — wid

Attributes: ISO

Description: This words creates an empty wordlist which is a single header structure *dea* that serves as a *word list identifier* or *WID* in the sense of the ISO standard. Definitions will be added to it after it has been copied to `CURRENT` . This handle can be placed in the top of the search order `CONTEXT` and the wordlist will be searched. This word underlies `NAMESPACE` , there is no *VLFA* .

A wid's significant fields are the flag field and the link field. The flag field indicates that it is dummy, i.e. not intended to be executed. The *link field address* contains the *dea* of the latest word of the wordlist or a *nil pointer* if empty. The namefield points to an empty string

See also: '`NAMESPACE`' '`ALSO`' '`>FFA`' '`>LFA`' '`FOR-WORDS`'

9.4.11 colon

Name: `colon`

No stackeffect

Attributes: ISO,FIG,E

Description: Used in the form called a colon-definition:

```
: cccc      ...      ;
```

Creates a dictionary entry defining '`cccc`' as equivalent to the following sequence of Forth word definitions '...' until the next `';'` (or possibly `';CODE'`). The word is added as the latest into the `CURRENT` word list. The compiling process is done by the text interpreter as long as `STATE` is non-zero. Words with the *immediate* bit set, attribute '`I`', are executed rather than being compiled.

See also: '`(CREATE)`'

9.4.12 (;CODE)

Name: `(;CODE)`

Stackeffect: addr —

Attributes: WANT,C

Description: The run-time proceedure, compiled by `;CODE` , that fills in the code field of the most recently defined word with '`addr`' .

See also: '`;CODE`'

9.4.13 (CREATE)

Name: `(CREATE)`

Stackeffect: sc —

Attributes:

Description: This is the basis for all defining words, including those which lack a data field in the ISO sense: `:` `USER` `VARIABLE`

`NAMESPACE` `CONSTANT` `DATA` . It creates a header with name '`sc`' in the dictionary and links it into the `CURRENT` word list.

See also: '`HEADER`' '`LINK`' '`CREATE`'

9.4.14 ;CODE

Name: `;CODE`

No stackeffect

Attributes: WANT,ISO,FIG,I,C

Description: Used in the form: '`: cccc CREATE .... ;CODE assembly mnemonics` ' It can only be used if the code and the data are not separated. It is used after `CREATE` instead of `DOES>` if the code following is assembler code instead of high level code. Stop compilation and terminate a new defining word '`cccc`' by compiling `(;CODE)` . Set `ASSEMBLER` to the top of the *search order* order. Start assembling to machine code the following mnemonics.

When '`cccc`' later executes in the form: '`cccc nnnn`' the word '`nnnn`' will be created with its execution proceedure given by the machine code following '`cccc`' . That is, when '`nnnn`' is executed, it does so by jumping to the code after '`nnnn`' . Because of intimate relation to the assembler, it is present in loadable form in the screens file `forth.lab` . Machine code must end in `NEXT` also available with the assembler.

See also: '`(;CODE)`' '`(CREATE)`'

9.4.15 HEADER

Name: `HEADER`

Stackeffect: sc — dea

Attributes:

Description: Create a dictionary entry structure for the word '`sc`' and returns its address into '`dea`'. A pointer to each of its fields is called a "field address" for code data flag line name source extra. In particular that a data field address is not a *data field*

in the ISO sense. `HEADER` initializes the *code field addres* and *data field address* to contain a same pointer to the area owned by this header, i.e. immediately following the completed header as appropriate for a low level (assembler) definition. The flag and link fields are initialised to zero , so not `HIDDEN` . The name '`sc`' is laid down in the dictionary before the header and filled in into the name field. The source field is filled in to best knowledge.

See also: '`(CREATE)`' '`>CFA`' '`>DFA`' '`>FFA`' '`>LFA`' '`>NFA`' '`>SFA`' '`>XFA`' '`LINK`'

9.4.16 LINK

Name: `LINK`

Stackeffect: dea wid —

Attributes:

Description: Links the Forth word represented by '`dea`' into the wordlist represented by '`wid`' as the latest entry. Alternatively, consider '`wid`' as an other '`dea2`'. Link '`dea`' between the '`dea2`' and its successor in the linked list.

See also: '`HEADER`' '`(CREATE)`'

9.5 DENOTATIONS

The wordset 'DENOTATIONS' contains prefixes (mostly one letter words) that introduce a *denotation* , i.e. a generalisation of NUMBER . PREFIX turns the latest definition into a prefix, similar to IMMEDIATE . If a word starting with the prefix is looked up in the dictionary, the prefix is found and executed. Prefix words parse input and leave a constant (number, char or string) on the stack, or compile such constant, depending on STATE . For a kernel system it is guaranteed that they reside in the minimum search order wordlist, associated with the namespace ONLY . To make a distinction with the same words in other wordlists, the names of denotations are prepended with "Prefix_" in the documentation. Actual names consists of the one character following "Prefix_". Apart from Prefix_0 , ONLY contains entries for all hex digits 1...9 and A...F. Like NUMBER always did, all denotations behave identical in interpret and compile mode and they are not supposed to be postponed. The use of prefixes for other purposes than denotations require great care.

9.5.1 Prefix_"

Name: Prefix_"

Stackeffect: — sc

Attributes: I,P

Description: Parse a " delimited string and leave it on the stack as a *stringconstant* , i.e. an address and a length. A " can be embedded in a string by doubling it. The string is permanent and takes dictionary space. The cell below the address contains the length, so 'DROP 1 CELLS -' can be used as a single cell reference. This is a denotation: during compilation this behaviour is compiled.

9.5.2 Prefix_&

Name: Prefix_&

Stackeffect: — c

Attributes: I,P

Description: Leave 'c' the non blank char that follows. Skip another blank character. This is a denotation: during compilation this behaviour is compiled.

See also: '^'

9.5.3 Prefix_+

Name: Prefix_+

Stackeffect: — s/d

Attributes: I,P

Description: A prefix that handles numbers that start with + .

See also: 'NUMBER' '(NUMBER)' '7' 'B'

9.5.4 Prefix_-

Name: Prefix_-

Stackeffect: — s/d

Attributes: I,P

Description: A prefix that handles numbers that start with - .

See also: 'NUMBER' '(NUMBER)' '7' 'B'

9.5.5 Prefix_0

Name: `Prefix_0`

Stackeffect: — s/d

Attributes: I,P

Description: A prefix that handles numbers that start with *0* . Similar words are present for all decimal and hex digits. ISO compatibility ony requires that denotators for decimal digits are present, one can always use a leading zero.

See also: ‘`NUMBER`’ ‘`(NUMBER)`’ ‘`B`’ ‘`7`’

9.5.6 Prefix_7

Name: `Prefix_7`

Stackeffect: — s/d

Attributes: I,P

Description: A prefix that handles numbers that start with *7* . Similar words are present for all decimal and hex digits.

See also: ‘`NUMBER`’ ‘`(NUMBER)`’ ‘`0`’ ‘`B`’

9.5.7 Prefix_B

Name: `Prefix_B`

Stackeffect: — s/d

Attributes: I,P

Description: A prefix that handles numbers that start with *B* . Similar words are present for all decimal and hex digits.

See also: ‘`NUMBER`’ ‘`(NUMBER)`’ ‘`0`’ ‘`7`’

9.5.8 Prefix_TICK

Name: `Prefix_TICK`

Stackeffect: — addr

Attributes: I,P

Description: Used in the form:

```
'nnnn
```

In interpret mode it leaves the *execution token*

(equivalent to the *dea* dictionary entry address) of dictionary word ‘`nnnn`’. If the word is not found after a search of the search order an appropriate error message is given. In compile mode it finds the same address, then compiles it as a literal. It is recommended that one never compiles or postpones it. (Use a combination of `NAME` and `FOUND` or any form of explicit parsing and searching instead.) Furthermore it is recommended that for non-portable code `'` is used in its *denotation* form without the space. Note that if you separate `'` by a space, the ISO-conforming version of `'` is found.

See also: ‘`HEADER`’ ‘`CONTEXT`’ ‘`'`’ ‘`[']`’ ‘`PRESENT`’ ‘`>CFA`’ ‘`>DFA`’ ‘`>FFA`’ ‘`>LFA`’

9.5.9 Prefix_^

Name: `Prefix_^`

Stackeffect: — b

Attributes: I,P

Description: Leave 'b' the control character value of the char that follows i.e. '^A' results in 1 and so on. Skip another blank character. This is a denotation: during compilation this behaviour is compiled.

See also: '&'

9.6 DICTIONARY

The wordset 'DICTIONARY' contains words that at a lower level than the wordset 'DEFINING' concern the memory area that is allocated to the dictionary. They may add data to the dictionary at the expense of the free space, one cell or one byte at a time, or allocate a buffer at once. The dictionary space may also be shrunk, and the words that were there are lost. The *dictionary entry address* or *dea* represents a word. It is the lowest address of a record with fields. Words to access those fields also belong to this wordset.

9.6.1 ' (This addition because texinfo won't accept a single quote)

Name: `'`

Stackeffect: — addr

Attributes: ISO,FIG

Description: Used in the form:

```
' nnnn
```

It leaves the *execution token* (equivalent to the *dea* dictionary entry address) of dictionary word 'nnnn'. If the word is not found after a search of the search order an appropriate error message is given. If compiled the searching is done while the word being compiled is executed. Because this is so confusing, it is recommended that one never compiles or postpones `'` . (Use a combination of `NAME` and `FOUND` or any form of explicit parsing and searching instead.) Furthermore it is recommended that for non-portable code `'` is used in its *denotation* form without the space.

See also: '[']' 'FOUND' '>CFA' '>DFA' '>FFA' '>LFA' '>NFA' '>SFA' '>XFA '

9.6.2 ,

Name: `,`

Stackeffect: n —

Attributes: ISO,FIG

Description: Store 'n' into the next available dictionary memory cell, advancing the *dictionary pointer* .

See also: 'DP' 'C,'

9.6.3 2,

Name: `2,`

Stackeffect: d —

Attributes: ISO,FIG

Description: Store the most significand cell of '`d`' into the next available dictionary cell, advancing the *dictionary pointer* , then the least significand one. .

See also: '`DP`' '`,`'

9.6.4 >BODY

Name: `>BODY`

Stackeffect: dea — addr

Attributes: ISO

Description: Given the *dictionary entry address* '`dea`' of a definition created with a `CREATE` / `DOES>` construct, return its *data* field (in the ISO sense) '`addr`'.

See also: '`'`' '`>CFA`' '`>DFA`' '`>PHA`' '`BODY>`' '`DATA`'

9.6.5 ALLOT

Name: `ALLOT`

Stackeffect: n —

Attributes: ISO,FIG

Description: Add the signed number to the *dictionary pointer* `DP` . May be used to reserve dictionary space or re-origin memory. As the Pentium is a byte-addressable machine '`n`' counts bytes.

See also: '`HERE`' '`,`'

9.6.6 BODY>

Name: `BODY>`

Stackeffect: addr — dea

Attributes:

Description: Convert the *data field* '`addr`' of a definition created with a `CREATE` / `DOES>` construct to its '`dea`'. ('`addr`' is not a header *data field address*) Where `>BODY` keeps working for a copy of the header, `BODY>` does not. There is some logic to this, because the *dea* to which the body belongs is no longer unique.

See also: '`'`' '`>BODY`'

9.6.7 C,

Name: `C,`

Stackeffect: b —

Attributes: ISO,FIG

Description: Store the least significand bits of '`b`' into the next available dictionary byte, advancing the *dictionary pointer* .

See also: '`DP`' '`,`'

9.6.8 DP

Name: `DP`

Stackeffect: —- addr

Attributes: FIG,U,L

Description: A user variable, the *dictionary pointer* , which contains the address of the next free memory above the dictionary. The value may be read by `HERE` and altered by `ALLOT` .

9.6.9 FIND

Name: `FIND`

Stackeffect: addr —xt 1/xt -1/addr 0

Attributes: ISO,WANT

Description: For the *old fashioned string* (stored with a preceding character count) at '`addr`' find a Forth word in the current search order. Return its execution token '`xt`'. If the word is immediate, also return 1, otherwise also return -1. If it is not found, leave the original '`addr`' and a zero. In ciforth the alternative `FOUND` is used, that uses a regular *stringconstant* , hence this is a loadable extension.

See also: '`FOUND`' '`CONTEXT`' '`PRESENT`' '`(FIND)`'

9.6.10 FORGET

Name: `FORGET`

No stackeffect

Attributes: ISO,FIG,E

Description: Executed in the form: `FORGET` '`cccc`' Deletes definition named '`cccc`' from the dictionary with all entries physically following it. Recover the space that was in use.

See also: '`FENCE`' '`FORGET-VOC`'

9.6.11 FOUND

Name: `FOUND`

Stackeffect: sc — dea

Attributes:

Description: Look up the string '`sc`' in the dictionary observing the current search order. If found, leave the dictionary entry address '`dea`' of the first entry found, else leave a *nil pointer*. If the first part of the string matches a *denotation* word, that word is found, whether the denotation is correct or not. The `dea` allows to retrieve all properties of the word such as whether it is immediate.

See also: '`PP`' '`FIND`' '`PRESENT`' '`CONTEXT`' '`(FIND)`' '`PREFIX` '

9.6.12 HERE

Name: `HERE`

Stackeffect: — addr

Attributes: ISO,FIG

Description: Leave the address '`addr`' of the next available dictionary location.

See also: '`DP`'

9.6.13 ID.

Name: `ID.`

Stackeffect: dea —

Attributes:

Description: Print a definition's name from its dictionary entry address. For dummy entries print nothing.

See also: '`'`' '`>FFA`' '`>NFA`'

9.6.14 IMMEDIATE

Name: `IMMEDIATE`

No stackeffect

Attributes:

Description: Mark the most recently made definition so that when encountered at compile time, it will be executed rather than being compiled, i.e. the immediate bit in its header is set. This method allows definitions to handle special compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceding it with `POSTPONE` .

9.6.15 PAD

Name: `PAD`

Stackeffect: — addr

Attributes: ISO,FIG

Description: Leave the address of the text output buffer, which is a fixed offset above `HERE` . The area growing downward from `PAD` is used for numeric conversion. The use of `PAD` is reserved for applications.

9.6.16 PREFIX

Name: `PREFIX`

No stackeffect

Attributes:

Description: Mark the most recently made definition a *prefix* . If searching the wordlists for a name that starts with the prefix, the prefix is a match for that name. This method allows to define numbers, and other *denotation* 's such as strings, in a modular and extensible fashion. A prefix word finds the interpreter pointer pointing to the remainder of the name (or number) sought for, and must compile that remainder. Some prefixes, like `Prefix_7` , decrement the parse pointer by one before starting to parse. Prefix words are mostly both immediate and *smart* , i.e. they behave differently when compiled, than interpreted. The result is that the compiled code looks the same and behaves the same than the interpreted code. Postponing prefix words voids your warranty. It is recommended that the only smart words present are prefix words.

See also: '`PP@@`' '`IMMEDIATE`' '`Prefix_0`' '`ONLY`'

9.6.17 PRESENT

Name: `PRESENT`

Stackeffect: sc — dea

Attributes:

Description: If the string '`sc`' is present as a word name in the current search order, return its '`dea`', else leave a *nil pointer* . For a *denotation* word, the name must match '`sc`' exactly.

See also: '`FOUND`' '`CONTEXT`' '`(FIND)`' '`NAMESPACE`' '`FIND `'

9.6.18 WORDS

Name: `WORDS`

No stackeffect

Attributes: ISO

Description: List the names of the definitions in the topmost word list of the search order.

See also: '`CONTEXT`'

9.6.19 [']

Name: `[']`

Stackeffect: — addr

Attributes: ISO,I

Description: Used in the form:

```
['] nnnn
```

In compilation mode it leaves the *execution token* (equivalent to the dictionary entry address) of dictionary word '`nnnn`'. So as a compiler directive it compiles the address as a literal. If the word is not found after a search of the search order an appropriate error message is given. It is recommended that where you can't use a *denotation* , or don't want to, you use a combination of `NAME` and `FOUND` (or `WORD` and `FIND`) instead.

See also: '`FOUND`' '`'`' '`EXECUTE` '

9.6.20 (FIND)

Name: `(FIND)`

Stackeffect: sc wid — sc dea

Attributes:

Description: Search down from the *WID* '`wid`' for a word with name '`sc`'. A *WID* is mostly a dummy *dea* found in the data field of a namespace, fetched from `CURRENT` or an other wid in the *search order* . Leave the dictionary entry address '`dea`' of the first entry found, else leave a zero. Do not consume the string '`sc`', as this is a repetitive action.

See also: '`~MATCH`' '`FOUND`' '`PRESENT`' '`>WID`'

9.6.21 >CFA

Name: `>CFA`

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres '`dea`' return its *code field address* '`addr`'. By jumping indirectly via this address the definition '`dea`' is executed. In ciforth it has offset 0, so it is actually the same as the *dea* .

See also: '`Prefix_'`' '`HEADER`'

9.6.22 >DFA

Name: `>DFA`

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres return its *data field address* '`addr`' . This contains a pointer to the code for a code word, to the data for a word defined by `VARIABLE` or `DATA` , to the high level code for a colon-definition, and to the `DOES>` pointer for a word build using `CREATE` . Normally this is the area behind the header, found via `>PHA` .

See also: '`Prefix_'`' '`>BODY`' '`HEADER`'

9.6.23 >FFA

Name: `>FFA`

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres return its *flag field address* '`addr`' .

See also: '`Prefix_'`' '`HEADER`' '`IMMEDIATE`' '`PREFIX`'

9.6.24 >LFA

Name: `>LFA`

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres return its *link field address* '`addr`'. It contains the *dea* of the previous word.

See also: '`Prefix_'`' '`HEADER`'

9.6.25 >NFA

Name: `>NFA`

Stackeffect: dea — nfa

Attributes:

Description: Given a dictionary entry addres return the *name field address* .

See also: '`Prefix_'`' '`HEADER`' '`ID.`'

9.6.26 >PHA

Name: `>PHA`

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres return the *past header address* . Here starts the area that no longer belongs to the header of a dictionary entry, but most often it is owned by it.

See also: '`Prefix_'`' '`HEADER`' '`>BODY`'

9.6.27 >SFA

Name: `>SFA`

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres return the *source field address* '`addr`'. If its content is higher than 1000, that is a pointer to after the name of the definition in the source code. This assumes that the source is still present in memory, which is true during development. If the content is zero, it is a kernel word and the source is not available interactively. Otherwise it is the block number from which the definition was compiled. No attempt is made to erase this information when it becomes incorrect, for e.g. turnkey programs.

See also: '`Prefix_'`' '`HEADER`'

9.6.28 >VFA

Name: `>VFA`

Stackeffect: dea — cfa

Attributes:

Description: Given the dictionary entry addres of a namespace return the address of the link to the next namespace. Traditionally this was called vocabulary, hence the *V*.

See also: '`NAMESPACE`' '`>CFA`' '`>WID`'

9.6.29 >WID

Name: `>WID`

Stackeffect: dea — wid

Attributes:

Description: Given the dictionary entry addres '`dea`' of a namespace return its *WID* '`wid`', a dummy *dea* that serves as the start of a dictionary search.

See also: '`NAMESPACE`' '`>VFA`' '`(FIND)`'

9.6.30 >XFA

Name: `>XFA`

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres return the *extra field address* '`addr`' . This is a field in the header that is free for an application to use for whatever purpose.

See also: '`Prefix_'`' '`OPT`' '`HEADER`'

9.6.31 FENCE

Name: `FENCE`

Stackeffect: — addr

Attributes: FIG,U

Description: A user variable containing an address below which `FORGET` ting is trapped. To forget below this point the user must alter the contents of `FENCE` .

9.6.32 FOR-VOCS

Name: `FOR-VOCS`

Stackeffect: x1..xn xt — x1...xn

Attributes:

Description: For all vocabularies execute '`xt`' with as data the *dea* of those words. '`xt`' must have the stack diagram '`x1..xn dea --- x1..xn`'

See also: '`FOR-WORDS`' '`EXECUTE`'

9.6.33 FOR-WORDS

Name: `FOR-WORDS`

Stackeffect: x1...xn xt dea — x1'...xm'

Attributes:

Description: For all words starting with and including '`dea`' execute '`xt`' with as data '`x1..xn`' plus the *dea* of those words by following the link fields. Mostly the dea will identify a WID. '`xt`' must have the stack diagram '`x1..xn dea' --- x1'..xm'`'. In that case all words of a wordlist are handled. If you don't want to include the WID itself, you can ignore it based on the dummy flag in its flag field. Note that you can use the *dea* of any word as a WID and the remainder of the word list will be searched. This word is similar to *TRAVERSE-WORDLIST* (ISO 2012), but the '`xt`' is not burdened by returning a goon-flag.

See also: '`FOR-VOCS`' '`EXECUTE`'

9.6.34 FORGET-VOC

Name: `FORGET-VOC`

Stackeffect: addr wid — addr

Attributes:

Description: Remove all words whose *dea* is greater (which mostly means later defined) than '`addr`' from a wordlist given by '`wid`' . This works too if links have been redirected, such that some earlier words point back to later defined words. Leave '`addr`' (as `FORGET-VOC` is intended to be used with `FOR-VOCS`) . If any whole namespace is removed, the search order is reset to '`ONLY FORTH`'. The space freed is not recovered.

See also: '`FORGET`'

9.6.35 HIDDEN

Name: `HIDDEN`

Stackeffect: dea —

Attributes:

Description: Make the word with dictionary entry address '`dea`' unfindable, by toggling the "smudge bit" in a definitions' flag field. If however it was the '`dea`' of an unfindable word, it is made findable again. Used during the definition of a colon word to prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error. It also prevents that a word can be used recursively.

See also: '`IMMEDIATE`' '`RECURSE`'

9.6.36 OPT

Name: `OPT`

Stackeffect: sc dea — sc dea1

Attributes:

Description:

This is a vector in behalf of optimisation, that starts as a noop, and can be filled in if the extra field is used for optimisation. It must adhere to the following specification: Search down from the *dea* '`dea`' for a word with name '`sc`'. Replace '`dea`' with '`dea1`' if the dictionary part between both could not contain the name searched for. '`dea1`' being a null pointer means that the dictionary doesn't contain the name.

See also: '`~MATCH`' '`FIND`' '`>XFA`'

9.6.37 ~MATCH

Name: `~MATCH`

Stackeffect: sc dea — sc dea n

Attributes:

Description: Intended to cooperate with `(FIND)` . Compares the *string constant* '`sc`' with the dea '`dea`''s name, Returns into '`n`' the difference between the first characters that compare unequal, or zero if the strings are the same up to the smallest length. It is required that the dea contains a pointer to a string variable, which may contain an empty string.

See also: '`FOUND`' '`CORA`'

9.7 DOUBLE

The wordset 'DOUBLE' contains words that manipulate *double* 's. In this 32 Forth you would hardly need doubles if it weren't for the NUMBER formatting wordset that uses them exclusively.

9.7.1 D+

Name: D+

Stackeffect: d1 d2 — dsum

Attributes: ISO,FIG

Description: Leave the double number 'dsum': the sum of two double numbers 'd1' and 'd2' .

See also: 'DNEGATE' '+'

9.7.2 DABS

Name: DABS

Stackeffect: d — ud

Attributes: ISO,FIG

Description: Leave the absolute value 'ud' of a double number 'd' .

See also: 'DNEGATE' 'ABS'

9.7.3 DNEGATE

Name: DNEGATE

Stackeffect: d1 — d2

Attributes: ISO

Description: 'd2' is the negation of 'd1'.

See also: 'D+'

9.7.4 S>D

Name: S>D

Stackeffect: n — d

Attributes: ISO

Description: Sign extend a single number to form a double number.

9.8 ENVIRONMENTS

The wordset 'ENVIRONMENTS' contains all words of the ENVIRONMENT namespace and those words needed to recognize them as Forth environment queries. Note that these are not environment variables in the sense that they are passed from an operating system to a program.

9.8.1 CORE

Name: CORE

Stackeffect: — ff

Attributes: ISO

Description: An environment query whether the CORE wordset is present.

See also: 'ENVIRONMENT?'

9.8.2 CPU

Name: `CPU`

Stackeffect: — d

Attributes: CI

Description: An environment query returning the cpu-type to be printed as a base-36 number.

See also: '`ENVIRONMENT?`'

9.8.3 ENVIRONMENT?

Name: `ENVIRONMENT?`

Stackeffect: sc — i*x true/false

Attributes: ISO

Description: If the string '`sc`' is a known environment attribute, leave into '`i*x`' the information about that attribute and a true flag, else leave a false flag. In fact the flag indicates whether the words is present in the `ENVIRONMENT` namespace and '`i*x`' is what is left by the word if executed.

See also: '`NAMESPACE`'

9.8.4 NAME

Name: `NAME`

Stackeffect: — sc

Attributes: CI

Description: An environment query giving the name of this Forth as a *string constant*.

See also: '`ENVIRONMENT?`'

9.8.5 SUPPLIER

Name: `SUPPLIER`

Stackeffect: — sc

Attributes: CI

Description: An environment query giving the SUPPLIER of this Forth as a *string constant*.

See also: '`ENVIRONMENT?`'

9.8.6 VERSION

Name: `VERSION`

Stackeffect: — sc

Attributes: CI

Description: An environment query giving the version `"5.5.0"` of this Forth as a *string constant*.

See also: '`ENVIRONMENT?`'

9.9 ERRORS

The wordset '`ERRORS`' contains words to handle errors and exceptions.

9.9.1 ?ERROR

Name: `?ERROR`

Stackeffect: f n —

Attributes:

Description:

If the boolean flag is true, *signal an error* with number 'n'. This means that an exception is thrown, and it is remembered that this is the original place where the exception originated. If the exception is never caught, an error message is displayed using ERROR . All errors signaled by the kernel go through this word.

See also: 'ERROR' '?ERRUR' 'THROW'

9.9.2 ?ERRUR

Name: ?ERRUR

Stackeffect: n —

Attributes:

Description: Handle the possible error 'n' in Unix fashion. If it is zero or positive, this means okay. If it is negative, its value identifies an error condition. This error is handled in the same way as by ?ERROR .

See also: 'ERROR' '?ERROR'

9.9.3 ABORT"

Name: ABORT"

Stackeffect: f —

Attributes: WANT,ISO,I,C

Description: Usage is ': <SOME> ... ABORT" <message>" ... ;'. If ABORT" finds a non-zero 'f' on the stack, the '<message>' is displayed and an ABORT is executed. Otherwise proceed with the words after '<message>'. This word can only be used in compile mode.

See also: '?ERROR'

9.9.4 CATCH

Name: CATCH

Stackeffect: ... xt — ... tc

Attributes: ISO

Description: Execute 'xt'. If it executes successfully, i.e. no THROW is executed by 'xt', leave a zero into 'tc' in addition to any stack effect 'xt' itself might have. Otherwise in 'tc' the non-zero throw code is left, and the stack depth is restored. The values of the parameters for 'xt' could have been modified. In general, there is nothing useful that can be done with those stack items. Since the stack depth is known, the application may DROP those items.

See also: 'THROW' 'QUIT' 'HANDLER'

9.9.5 ERROR

Name: ERROR

Stackeffect: n —

Attributes:

Description: Notify the user that an uncaught exception or error with number 'n' has occurred. The word that caused it is found using WHERE and displayed . Also 'n' is passed to MESSAGE in order to give a description of the error, clamped to the range [-256, 63]. This word is executed by THROW before restarting the interpreter and can be revectored to give more elaborate diagnostics, or even do a last minute recovery.

See also: 'MESSAGE' '?ERROR' 'WARNING'

9.9.6 ERRSCR

Name: `ERRSCR`

Stackeffect: — addr

Attributes:

Description: A variable containing the address of the number of the screen from which messages are offset. Messages correspond with lines and the offset may be positive or negative. '`0 MESSAGE`' prints the first line of this screen. Traditionally this was screen 4, but the negative Unix error numbers makes this infeasible.

See also: '`C/L`' '`MESSAGE`'

9.9.7 MESSAGE

Name: `MESSAGE`

Stackeffect: n —

Attributes:

Description: `MESSAGE` is generally used to print error and warning messages. If `WARNING` is zero (disc unavailable), this is a noop. Print on the standard error channel the text of line '`n`' relative to screen `ERRSCR` . '`n`' may be positive or negative and beyond just screen `ERRSCR` . A certain range around 0 is reserved. The messages with small positive offset contain ciforth error messages. The messages with small negative offset contain the strings belonging to the return values for Linux system calls . '`0 MESSAGE`' will print version information about the library file where the messages sit in.

See also: '`ERROR`' '`BLOCK-FILE`'

9.9.8 THROW

Name: `THROW`

Stackeffect: ... tc — ... / ... tc

Attributes: ISO

Description: If '`tc`' is zero, it is merely discarded. If we are executing under control of a `CATCH` , see `CATCH` for the effect of a non-zero '`tc`'. If we are executing not under control of a `CATCH` , a non-zero '`tc`' gives a message to the effect that this exception has occurred and starts Forth anew.

See also: '`CATCH`' '`QUIT`' '`HANDLER`' '`?ERROR`' '`ERROR`'

9.9.9 WARNING

Name: `WARNING`

Stackeffect: — addr

Attributes: FIG,U

Description: A user variable containing a value controlling messages. If it is 1, a library file is assumed to be open, and messages are fetched from it. If it is 0, error messages will be presented by number only.

See also: '`MESSAGE`' '`ERROR`' '`ERRSCR`'

9.9.10 WHERE

Name: `WHERE`

Stackeffect: — addr

Attributes: U

Description: A user variable pair which contains the start of the source and the position after the last character parsed when an error thrown by `?ERROR` , so not of exceptions thrown. The contents of `WHERE` is interpreted by `ERROR` if the corresponding exception was never caught.

See also: '`THROW`' '`CATCH` '

9.9.11 (ABORT")

Name: `(ABORT")`

Stackeffect: f —

Attributes: WANT

Description: The run time action of `ABORT"` .

9.9.12 HANDLER

Name: `HANDLER`

Stackeffect: — addr

Attributes:

Description: A user variable address containing a pointer to the last exception intercepting frame activated by `CATCH` . It points into the return stack. If there is a `THROW` , the return stack is restored from `HANDLER` effecting a multiple level return. It is called a frame because more things are restored, such as the position of the data stack top, and the previous value of `HANDLER` .

See also: '`CATCH`' '`THROW`'

9.10 FILES

The wordset '`FILES`' contains words to input and output to files, or load words from files. They also are underlying the '`BLOCKS`' facilities.

9.10.1 CLOSE-FILE

Name: `CLOSE-FILE`

Stackeffect: fileid — ior

Attributes: ISO

Description: Close the file with file handle in '`fileid`'. Return a result code into '`ior`'. The latter is the Linux error code negated, usable as a throw code.

See also: '`OPEN-FILE`' '`READ-FILE`' '`WRITE-FILE`' '`CREATE-FILE`' '`DELETE-FILE`'

9.10.2 CREATE-FILE

Name: `CREATE-FILE`

Stackeffect: sc fam — fileid ior

Attributes: ISO

Description: Create a file with name '`sc`' and file access method '`fam`' . If the file already exists, it is truncated to zero length. Return a file handle into '`fileid`' and a result code into '`ior`'. The latter is the '`errno`' negated, to be inspected using `MESSAGE` . The handle is open for READ_WRITE, so '`fam`' is in fact ignored. The access privileges default to octal 755. This can be changed by applying `mask 8 LSHIFT OR` where `mask` contains the privilege bits one wants to change.

See also: '`OPEN-FILE`' '`READ-FILE`' '`WRITE-FILE`' '`DELETE-FILE`'

9.10.3 DELETE-FILE

Name: `DELETE-FILE`

Stackeffect: sc — ior

Attributes: ISO

Description: Delete the file with name '`sc`'. Return a result code into '`ior`'. The latter is the '`errno`' negated, to be inspected using `MESSAGE` .

See also: '`OPEN-FILE`' '`READ-FILE`' '`WRITE-FILE`' '`CREATE-FILE`'

9.10.4 FILE-POSITION

Name: `FILE-POSITION`

Stackeffect: fd — ud ior

Attributes: ISO

Description:

'`ud`' is the file current position for the file that is open at '`fd`'. '`ior`' is 0 for success, or otherwise '`errno`' negated . Information about error codes can be found by `MESSAGE` .

See also: '`OPEN-FILE`' '`REPOSITION-FILE`'

9.10.5 GET-FILE

Name: `GET-FILE`

Stackeffect: sc1 — sc2

Attributes:

Description: Get the content of the file with name '`sc1`'; leave it as a string '`sc2`'. Any errors are thrown. The file is permanently stored at `HERE` .

See also: '`PUT-FILE`' '`OPEN-FILE`' '`THROW`'

9.10.6 INCLUDED

Name: `INCLUDED`

Stackeffect: sc1 — i*x

Attributes: ISO

Description: Interpret the content of the file with name '`sc1`' as if it was typed from the console, leaving result '`i*x`'. The file is permanently stored in the dictionary, but trimmed in compiled programs.

See also: '`LOAD`' '`INCLUDE`'

9.10.7 INCLUDE

Name: `INCLUDE`

Stackeffect: `"name"` — i*x

Attributes: ISO

Description: Interpret the content of the file with `"name"` as if it was typed from the console, leaving result '`i*x`'. The file is permanently stored in the dictionary, but trimmed in compiled programs.

See also: '`LOAD`'

9.10.8 OPEN-FILE

Name: `OPEN-FILE`

Stackeffect: sc fam — fileid ior

Attributes: ISO

Description: Open the file with name '`sc`' and file access method '`fam`'. Return a file handle into '`fileid`' and a result code into '`ior`'. The latter is the '`errno`' error code negated, usable as a throw code. '`fam`' is one of 0=READ_ONLY, 1=WRITE_ONLY, 2=READ_WRITE. all files are opened in binary mode.

See also: '`READ-FILE`' '`WRITE-FILE`' '`CREATE-FILE`' '`DELETE-FILE`' '`CLOSE-FILE`'

9.10.9 PUT-FILE

Name: `PUT-FILE`

Stackeffect: sc1 sc2 —

Attributes:

Description: Save the *string constant* '`sc1`' to a file with the name '`sc2`'. Any errors are thrown.

See also: '`GET-FILE`' '`OPEN-FILE`' '`THROW`'

9.10.10 READ-FILE

Name: `READ-FILE`

Stackeffect: addr n1 fd — n2 ior

Attributes: ISO

Description: Read '`n`' characters to '`addr`' from current position of the file that is open at '`fd`' . '`n2`' is the number of characters successfully read, this may be zero. '`ior`' is 0 for success, or otherwise '`errno`' error code negated, usable as a throw code.

See also: '`OPEN-FILE`' '`WRITE-FILE`' '`REPOSITION-FILE`' '`BLOCK-READ`'

9.10.11 REPOSITION-FILE

Name: `REPOSITION-FILE`

Stackeffect: ud fd — ior

Attributes: ISO

Description: Position the file that is open at '`fd`' at position '`ud`'. '`ior`' is 0 for success, or otherwise '`errno`' negated . Information about error codes can be found by `MESSAGE` .

See also: '`OPEN-FILE`' '`READ-FILE`' '`WRITE-FILE`'

9.10.12 WRITE-FILE

Name: `WRITE-FILE`

Stackeffect: addr n fd — u1

Attributes: ISO

Description: Write '`n`' characters from '`addr`' to the file that is open at '`fd`' , starting at its current position. '`u1`' is 0 for success, or otherwise '`errno`' error code negated, usable as a throw code.

See also: '`OPEN-FILE`' '`READ-FILE`' '`REPOSITION-FILE`' '`BLOCK-WRITE`'

9.10.13 RW-BUFFER

Name: `RW-BUFFER`

Stackeffect: — addr

Attributes:

Description: A constant that leaves the address of a disk buffer used by file i/o words.

See also: '`READ-FILE`' '`OPEN-FILE`' '`ZEN`'

9.11 FORMATTING

The wordset '`FORMATTING`' generates formatted output for numbers, i.e. printing the digits in a field with a certain width, possibly with sign etc. This is possible in any *number base* . (Normally base 10 is used, which means that digits are found as a remainder by dividing by 10). Formatting in Forth is always based on *double* numbers. Single numbers are handled by converting them to *double* first. This requires some double precision operators to be present in the Forth core. See Section 9.7 [DOUBLE], page 89, wordset. See Section 9.19 [MULTIPLYING], page 115, wordset.

9.11.1 #>

Name: `#>`

Stackeffect: d — sc

Attributes: ISO,FIG

Description: Terminates numeric output conversion by dropping '`d`', leaving the formatted string '`sc`' .

See also: '`<#`'

9.11.2 #S

Name: `#S`

Stackeffect: d1 — d2

Attributes: ISO,FIG

Description: Generates ASCII text in the text output buffer, by the use of `#` , until a zero double number '`d2`' results. Used between `<#` and `#>` .

9.11.3

Name: `#`

Stackeffect: d1 — d2

Attributes: ISO,FIG

Description: Generate from a double number '`d1`', the next ASCII character which is placed in an output string. Result '`d2`' is the quotient after division by `BASE` , and is maintained for further processing. Used between `<#` and `#>` .

See also: '`#S`' '`HLD`'

9.11.4 <#

Name: `<#`

No stackeffect

Attributes: ISO,FIG

Description: Setup for pictured numeric output formatting using the words: `<# # #S SIGN #>` The conversion is done on a double number producing text growing down from `PAD`

.

See also: 'DPL' 'HLD' 'HOLD' 'FLD'

9.11.5 >NUMBER

Name: >NUMBER

Stackeffect: ud1 addr1 u1 — ud2 addr2 u2

Attributes: ISO

Description:

'ud2' is the result of converting the characters within the character string specified by 'addr1 u1' into digits, using the number in BASE , and adding each into ud1 after multiplying 'ud1' by the number in BASE . Conversion continues until a character that is not convertible is encountered or the string is entirely converted. 'addr2' is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. 'u2' is the number of unconverted characters in the string. If 'ud2' overflows, in ciforth 'ud2' will be incorrect, but no crash will result. Both - and + are considered unconvertible character's by '>NUMBER' .

See also: 'NUMBER' 'DIGIT' 'DPL'

9.11.6 BASE

Name: BASE

Stackeffect: — addr

Attributes: ISO,FIG,U

Description: A *user variable* containing the current number base used for input and output conversion.

See also: 'DECIMAL' 'HEX' '<#'

9.11.7 DECIMAL

Name: DECIMAL

No stackeffect

Attributes: ISO,FIG

Description: Set the numeric conversion BASE for decimal input-output.

See also: 'HEX'

9.11.8 HEX

Name: HEX

No stackeffect

Attributes: ISO,FIG

Description: Set the numeric conversion BASE for hexadecimal (base 16) input-output.

See also: 'DECIMAL'

9.11.9 HOLD

Name: HOLD

Stackeffect: c —

Attributes: ISO,FIG

Description: Add the character 'c' to the beginning of the output string. It must be executed for numeric formatting inside a <# and #> construct .

See also: '#' 'DIGIT'

9.11.10 SIGN

Name: `SIGN`

Stackeffect: n —

Attributes: ISO,FIG

Description: Add an ASCII minus-sign `-` inside a `<#` and `#>`

construct to the beginning of a converted numeric output string in the text output buffer when '`n`' is negative. Must be used between `<#` and `#>` .

See also: '`HOLD`'

9.11.11 (NUMBER)

Name: `(NUMBER)`

Stackeffect: — d1

Attributes:

Description: Convert the ASCII text at the *current input source* with regard to `BASE` . The new value is accumulated into double number '`d1`' , being left. A decimal point, anywhere, signifies that the input is to be considered as a double. ISO requires it to be at the end of the number. ciforth allows any number of decimal points with the same meaning. ciforth also allows any number of comma's that are just ignored, to improve readability. If the first unconvertible digit is not a blank, this is an error. This word is used by all number prefixes. ciforth guarantees that PP points directly after a number parsed, such that for double numbers '`PP DPL  - 1-`' gives the number of decimals.

See also: '`NUMBER`' '`?BLANK`' '`Prefix_7`'

9.11.12 DIGIT

Name: `DIGIT`

Stackeffect: c n1 — n2 true (ok) c n1 — x false (bad)

Attributes:

Description: Converts the ASCII character '`c`' (using base '`n1`') to its binary equivalent '`n2`' , accompanied by a true flag. If the conversion is invalid, leaves a don't care value and a false flag.

9.11.13 DPL

Name: `DPL`

Stackeffect: —- addr

Attributes: FIG,U

Description: A user variable containing the address of the decimal point on double integer or floating point input, or a *nil pointer*.

See also: '`<#`' '`FLD`' '`HLD`'

9.11.14 ECL

Name: `ECL`

Stackeffect: —- addr

Attributes:

Description: A user variable containing the address of the exponential sign in a floating point input, or a *nil pointer*.

See also: '`DPL`'

9.11.15 FLD

Name: `FLD`

Stackeffect: — addr

Attributes: FIG,U

Description: A user variable for control of number output field width. Unused in the kernel of ciforth.

9.11.16 HLD

Name: `HLD`

Stackeffect: — addr

Attributes: FIG

Description: A user variable that holds the address of the latest character of text during numeric output conversion.

See also: '`<#`' '`DPL`' '`FLD`' '`HOLD`'

9.11.17 NUMBER

Name: `NUMBER`

Stackeffect: — s/d

Attributes:

Description:

This word is intended to be called from (aliased as) single character denotation words, marked as `PREFIX` `IMMEDIATE` . Therefore it decrements the parse pointer to include this first character. Starting with this character, it converts characters from the current input source into a number, and compiles or executes this number, depending on `STATE` . If the string contains a decimal point it is a double else a single integer number. If numeric conversion is not possible, an error message will be given. Any comma's in the number are ignored, to help in structuring large numbers. Traditionally comma's also signified double precision numbers. Revectoring `NUMBER` affects all applicable number prefixes. In this way floating point numbers can be accomodated.

See also: '`BASE`' '`(NUMBER)`'

9.12 INITIALISATIONS

The wordset '`INITIALISATIONS`' contains words to initialise, reinitialise or configure Forth.

9.12.1 +ORIGIN

Name: `+ORIGIN`

Stackeffect: n — addr

Attributes:

Description: Leave the memory address relative by '`n`' bytes to the area from which the user variables are initialised during startup. Given a *user variable* the phrase '`>DFA +ORIGIN`' finds the address where its initial value is stored. One can access or modify those boot-up parameters, prior to saving a customised image in behalf of turnkey programs. This will affect the initialisation by `COLD` .

In this forth switching of the user area in behalf of multi-tasking is accomplished automatically when the return stack is switched.

See also: '`USER`'

9.12.2 ABORT

Name: `ABORT`

No stackeffect

Attributes: ISO,FIG

Description: Restart the interpreter. In addition and before the actions of `QUIT` , clear also the data stack and reset the exception mechanism. This word is silent. This may be confusing at times, because you can't tell the difference between a word that is still busy or that has aborted.

See also: '`WARM`' '`_INIT`'

9.12.3 COLD

Name: `COLD`

No stackeffect

Attributes: FIG

Description: When called for the first time it initialises all user variables to their boot up values values, i.a. the stacks and the *dictionary pointer* . Initialise the system as per `_INIT` . Handle command line options, if any, as per `OPTIONS` . Show signon message and restart via `ABORT` . May be called from the terminal to restart, while keeping all definitions. But it is better to say `BYE` to Forth and start again.

See also: '`WARM`' '`_INIT`' '`OPTIONS`'

9.12.4 OK

Name: `OK`

No stackeffect

Attributes: ISO,FIG

Description: Takes care of printing the okay-message, after interpreting a line. Default it prints "OK" only for an interactive session in interpret `STATE` . It can be revectored to show debugging info or give a prompt.

See also: '`QUIT`' '`COLD`'

9.12.5 OPTIONS

Name: `OPTIONS`

Stackeffect: ft — f2

Attributes: CI

Description: Handle command line option. If an option is given at startup, execute the screen corresponding to the option letter and return a false flag into '`f2`', otherwise leave a true flag. This is used to suppress the signon message if there is an option. An option screen can always decide to execute `.SIGNON` . By redefining `OPTIONS` as '`DROP 0`', the handling of options during cold boot is suppressed, useful for turnkey applications.

See also: '`COLD`'

9.12.6 QUIT

Name: `QUIT`

No stackeffect

Attributes: ISO,FIG

Description: Restart the interpreter. Clear the return stack, stop compilation, and return control to the operators terminal, or to the redirected input stream. This means `(ACCEPT)` user input to somewhere in the terminal input buffer, and then `INTERPRET` with that as a `SOURCE` . No message is given.

See also: '`TIB`' '`ABORT`'

9.12.7 WARM

Name: `WARM`

No stackeffect

Attributes: FIG

Description: Perform a so called "warm" start. Reinitialise the system as per `INIT` . Show the signon message and restart via `ABORT` .

See also: '`ABORT`'

9.12.8 _INIT

Name: `_INIT`

No stackeffect

Attributes: FIG

Description: Initialise or reinitialise the system. Reset the data stack, the search order, the number base and the exception mechanism. Initialise the block mechanism. Any blocks that have not yet been written back to mass storage are discarded. Now open the file that contains the blocks, in read-only mode. After `_INIT` we have the following situation. The search order contains the `FORTH` words, plus `ONLY` with i.a. number handling. Definitions are added to the `FORTH` *namespace* . The number base is decimal.

See also: '`WARM`' '`COLD`' '`FORTH`' '`BLOCK-INIT`' '`BASE`' '`S0`'

9.13 INPUT

The wordset '`INPUT`' contains words to get input from the terminal and such. See Section 9.10 [FILES], page 93, for disk I/O. See Section 9.1 [BLOCKS], page 60, for access of blocks.

9.13.1 (ACCEPT)

Name: `(ACCEPT)`

Stackeffect: — sc

Attributes:

Description: Accept characters from the terminal, until a *RET* is received and return the result as a constant string '`sc`'. It doesn't contain any line ending, but the buffer still does and after `1+` the string ends in a *LF*. The editing functions are the same as with `ACCEPT` . This is lighter on the system and sometimes easier to use than `ACCEPT`

This input remains valid until the next time that the console buffer is refilled. With TIBSIZE characters the terminal input buffer overflows, with unpredictable results.

See also: '`KEY`' '`KEY?`' '`ACCEPT`' '`PP@@`' '`REFILL-TIB`'

9.13.2 ACCEPT

Name: `ACCEPT`

Stackeffect: addr count — n

Attributes: ISO

Description: Transfer at most '`count`' characters from the terminal to '`addr`' . The transfer takes place when a *RET* is received, and excess characters after '`count`' are ignored. The simple tty editing functions of Linux are observed, i.e. the "erase" (delete a character) and "kill" (delete a line) characters. Typically these are the backspace key and ^U. The number of characters not including the *RET* is returned into 'n'.

See also: '(ACCEPT)' 'KEY' 'KEY?'

9.13.3 KEY?

Name: KEY?

Stackeffect: — ff

Attributes: ISO

Description: Return into 'ff' whether a character is available. The next execution of KEY will return the character immediately.

See also: 'KEY' 'ACCEPT' 'KEY?'

9.13.4 KEY

Name: KEY

Stackeffect: — c

Attributes: ISO,FIG

Description: Leave the ASCII value of the next terminal key struck.

See also: 'ACCEPT' 'KEY?'

9.13.5 PP

Name: PP

Stackeffect: — addr

Attributes:

Description: A user variable containing a pointer within the current input text buffer (terminal or disc) from which the next text will be accepted. All parsing words use and move the value of PP . This is different from how ISO >IN works, in the sense that zeroing it doesn't reset the parse pointer to the start of the current line.

See also: '(>IN)' 'WORD' 'NAME' 'NUMBER' 'PARSE' 'PP@@'

9.13.6 RUBOUT

Name: RUBOUT

Stackeffect: — c

Attributes:

Description: A user variable, leaving the key code that must delete the last character from the input buffer. In this ciforth it is not used, as the terminal input editing is left to the host operating system.

See also: 'USER'

9.13.7 TIB

Name: TIB

Stackeffect: — addr

Attributes: ISO,FIG,U

Description: A user variable containing the address of the terminal input buffer. Traditionally, this was used for file i/o too, but not so in ciforth.

See also: 'QUIT'

9.13.8 (>IN)

Name: `(>IN)`

Stackeffect: — addr

Attributes:

Description: This is a user variable that is foreseen for code that fakes the behaviour of `>IN` in ciforth. This could be part of the code loaded by '`"-legacy-" WANTED` '. (The word `>IN` is intended to be used in the context that the source is interpreted line by line.) In ciforth the parse pointer `PP` is used by the system instead of `>IN` .

See also: '`>IN`' '`PP`'

9.13.9 REFILL-TIB

Name: `REFILL-TIB`

Stackeffect: —

Attributes:

Description: Accept characters from the terminal input stream such as to fill up `TIB` . Normally this means until a ***RET***. It is now consumable by `ACCEPT` or after `SET-SRC` by Forth parsing words like `NAME` . The editing functions are those described by `ACCEPT` . Immediately, after `REFILL-TIB` '`REMAINDER 2@`' defines the characters ready in the input buffer. All characters are retained including the ***RET***.

If the input is redirected (such that after a ***RET*** more characters are available) '`REMAINDER 2@`' contains the part of `TIB` that is not yet consumed by `(ACCEPT)` , and outside the reach of `SRC` .

All errors from Linux result in an exception. The end of normal or redirected I/O results in an an end-of-pipe exception in this word, which terminates ci86.lina32.html and is not considered an error condition.

See also: '`ACCEPT`' '`(ACCEPT)`'

9.13.10 REMAINDER

Name: `REMAINDER`

Stackeffect: — addr

Attributes:

Description: A pointer to a constant string that contains the balance of characters fetched from the the input stream, but not yet part of the input buffer. Unless there is redirection from a file, this contains an empty string. Used as in '`REMAINDER 2@`' .

See also: '`REFILL-TIB`'

9.13.11 SET-TERM

Name: `SET-TERM`

Stackeffect: len b —

Attributes:

Description: Set the terminal length to '`len`' and toggle the c_lflag field with '`b`' in the termios structure `TERMIO` . In particular toggling with 0x0A makes that the terminal doesn't wait for a ***RET***. This word is invalid for pipes, and is replaced by a `2DROP` in that case.

See also: '`TERMIO`' '`KEY`'

9.13.12 TERMIO

Name: `TERMIO`

Stackeffect: — addr

Attributes:

Description: Leave the address of the terminal description, this has the layout of a c-structure '`termios`' .

See also: '`SET-TERM`'

9.14 JUGGLING

The wordset '`JUGGLING`' contains words that change order of data on the *data stack* . The necessity for this arise, because the data you want to feed to a Forth word is not directly accessible, i.e. on top of the stack. It is also possible that you need the same data twice, because you have to feed it to two different words. Design your word such that you need them as little as possible, because they are confusing.

9.14.1 2DROP

Name: `2DROP`

Stackeffect: n1 n2 —

Attributes: ISO

Description: Drop the topmost two numbers (or one double number) from the stack.

See also: '`DROP`' '`2DUP`'

9.14.2 2DUP

Name: `2DUP`

Stackeffect: d — d d

Attributes: ISO

Description: Duplicate the double number on the stack.

See also: '`OVER`'

9.14.3 2OVER

Name: `2OVER`

Stackeffect: d1 d2 — d1 d2 d1

Attributes: ISO

Description: Copy the second stack double, placing it as the new top.

See also: '`OVER`'

9.14.4 2SWAP

Name: `2SWAP`

Stackeffect: d1 d2 — d2 d1

Attributes: ISO

Description: Exchange the top doubles on the stack.

See also: '`SWAP`'

9.14.5 ?DUP

Name: `?DUP`

Stackeffect: n1 — n1 (if zero) / n1 — n1 n1 (non-zero)

Attributes: ISO,FIG

Description: Reproduce '`n1`' only if it is non-zero. This is usually used to copy a value just before `IF` , to eliminate the need for an `ELSE` part to drop it.

See also: '`DUP`' '`_`'

9.14.6 DROP

Name: `DROP`

Stackeffect: n —

Attributes: ISO,FIG

Description: Drop the number from the stack.

See also: '`DUP`' '`SWAP`' '`OVER`'

9.14.7 DUP

Name: `DUP`

Stackeffect: n — n n

Attributes: ISO,FIG

Description: Duplicate the value on the stack.

See also: '`OVER`' '`SWAP`' '`DROP`'

9.14.8 NIP

Name: `NIP`

Stackeffect: n1 n2—n2

Attributes: ISO

Description: Drop the second number from the stack.

See also: '`DUP`' '`DROP`' '`SWAP`' '`OVER`'

9.14.9 OVER

Name: `OVER`

Stackeffect: nl n2 — nl n2 n1

Attributes: ISO,FIG

Description: Copy the second stack value, placing it as the new top.

See also: '`DUP`'

9.14.10 ROT

Name: `ROT`

Stackeffect: nl n2 n3 — n2 n3 nl

Attributes: ISO,FIG

Description: Rotate the top three values on the stack, bringing the third to the top.

See also: '`SWAP`'

9.14.11 SWAP

Name: `SWAP`

Stackeffect: n1 n2 — n2 n1

Attributes: ISO,FIG

Description: Exchange the top two values on the stack.

See also: '`ROT`'

9.15 LIBRARY

The words of the library have nothing particular in common, except that they are available in the Library Accessable by Block *LAB* and not the ciforth kernel.

All of them can be loaded from the library by '`WANT`' '`SOME_WORD`'. Most common ISO words are available. The assembler is loaded by '`WANT ASSEMBLERi86`' . Floating point is loaded by '`WANT -fp-`' . Interactive controle structures are loaded by '`WANT -scripting-`' . Multi-tasking is loaded by '`WANT THREAD-PET`' . The memory wordset is loaded by '`WANT ALLOCATE`' .

Useful application words in the *LAB* are i.a. : `ALIAS QSORT MERGE-SORT` . Useful debugging words in the *LAB* are i.a. : `DO-DEBUG LOCATE SEE REGRESS` . Non-iso words are supposedly usable from the description that goes with the code.

9.16 LOGIC

The wordset '`LOGIC`' contains logic operators and comparison operators. A comparison operator (such as `=`) delivers a *Forth flag* , -1 for true, 0 for false, representing a condition (such as equality of two numbers). The number -1 has all bits set to one. The logical operators (`AND` etc.) work on all 32 bits, one by one. In this way they are useful for mask operations, as well as for combining conditions represented as flag's. But beware that `IF` only cares whether the top of the stack is non-zero, such that `-` can mean non-equal to `IF` . Such conditions (often named just *flag* 's) cannot be directly combined using logical operators, but '`0= 0=`' can help.

9.16.1 0<

Name: `0<`

Stackeffect: n — ff

Attributes: ISO,FIG

Description: Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

See also: '`<`' '`0=`'

9.16.2 0=

Name: `0=`

Stackeffect: n — ff

Attributes: ISO,FIG

Description: Leave a true flag '`ff`' is the number '`n`' is equal to zero, otherwise leave a false flag. It may be aliased to `NOT` , which inverts a flag.

See also: '`=`' '`0<`'

9.16.3 <>

Name: `<>`

Stackeffect: n1 n2 — ff

Attributes: ISO

Description: Leave a true flag if '`n1`' is not equal than '`n2`' ; otherwise leave a false flag.

See also: '`>`' '`=`' '`0<`'

9.16.4 <

Name: `<`

Stackeffect: n1 n2 — ff

Attributes: ISO

Description: Leave a true flag if '`n1`' is less than '`n2`' ; otherwise leave a false flag.

See also: '`=`' '`>`' '`0<`'

9.16.5 =

Name: `=`

Stackeffect: n1 n2 — ff

Attributes: ISO,FIG

Description: Leave a true flag if '`n1=n2`' ; otherwise leave a false flag.

See also: '`<`' '`>`' '`0=`' '`-`'

9.16.6 >

Name: `>`

Stackeffect: n1 n2 — ff

Attributes: ISO

Description: Leave a true flag if '`n1`' is greater than '`n2`' ; otherwise leave a false flag.

See also: '`<`' '`=`' '`0<`'

9.16.7 AND

Name: `AND`

Stackeffect: n1 n2 — n3

Attributes: ISO,FIG

Description: Leave the bitwise logical and of '`n1`' and '`n2`' as '`n3`' . For Forth flags (0 or -1) this is the *logical and* operator.

See also: '`XOR`' '`OR`'

9.16.8 INVERT

Name: `INVERT`

Stackeffect: n1 — n2

Attributes: ISO

Description: Invert all bits of '`n1`' leaving '`n2`' . For pure flags (0 or -1) this is the *logical not* operator.

See also: '`AND`' '`OR`'

9.16.9 OR

Name: `OR`

Stackeffect: n1 n2 — n3

Attributes: ISO,FIG

Description: Leave the bit-wise logical or of two 32-bit values. For Forth flags (0 or -1) this is the *logical or* operator.

See also: '`AND`' '`XOR`'

9.16.10 U<

Name: `U<`

Stackeffect: u1 u2 — ff

Attributes: ISO

Description: Leave a true flag if '`u1`' is less than '`u2`' ; otherwise leave a false flag.(Interpreted as unsigned numbers).

See also: '`<`'

9.16.11 XOR

Name: `XOR`

Stackeffect: nl n2 — n3

Attributes: ISO,FIG

Description: Leave the bitwise logical exclusive or of two 32-bit values. For Forth flags (0 or -1) this is the *logical xor* operator.

See also: '`AND`' '`OR`'

9.17 MEMORY

The wordset '`MEMORY`' contains words to fetch and store numbers from *double* s, *cell* s or bytes in memory. There are also words to copy blocks of memory or fill them, and words that fetch a *cell* , operate on it and store it back.

9.17.1 !

Name: `!`

Stackeffect: n addr —

Attributes: ISO,FIG

Description: Store all 32 bits of n at '`addr`' .

See also: '`@`' '`C!`' '`2!`' '`FAR!`' '`PW!`' '`PC!`'

9.17.2 +!

Name: `+!`

Stackeffect: n addr —

Attributes: ISO,FIG

Description: Add '`n`' to the value at '`addr`'.

See also: '`TOGGLE`' '`!`'

9.17.3 2!

Name: `2!`

Stackeffect: x1 x2 addr —

Attributes: ISO

Description: Store a pair of 32 bits values '`x1`' '`x2`' to consecutive cells at '`addr`' . '`x2`' is stored at the lowest address.

See also: '`2@`' '`!`' '`C!`'

9.17.4 2@

Name: `2@`

Stackeffect: addr— x1 x2

Attributes: ISO

Description: Leave a pair of 32 bits values '`x1`' '`x2`' from consecutive cells at '`addr`' . '`x2`' is fetched from the lowest address.

See also: '`2!`' '`@`' '`C@`'

9.17.5 @

Name: `@`

Stackeffect: addr — n

Attributes: ISO,FIG

Description: Leave the 32 bit contents '`n`' of '`addr`' .

See also: '`!`' '`C@`' '`2@`' '`PW@`' '`PC@`' '`FAR@`'

9.17.6 ALIGNED

Name: `ALIGNED`

Stackeffect: addr1 — addr2

Attributes: ISO

Description: Make sure that '`addr1`' is *aligned* by advancing it if necessary to '`addr2`'.

See also: '`ALIGN`'

9.17.7 ALIGN

Name: `ALIGN`

Stackeffect: —

Attributes: ISO

Description: Make sure that `HERE` is *aligned* by advancing it if necessary. This means that data of any size can be fetched from that address efficiently.

See also: '`ALIGNED`'

9.17.8 BLANK

Name: `BLANK`

Stackeffect: addr count —

Attributes: ISO

Description: This is shorthand for "`BL FILL` ".

9.17.9 BM

Name: `BM`

Stackeffect: —- addr

Attributes:

Description: A constant leaving the address of the lowest memory in use by Forth.

See also: '`DP`' '`EM`'

9.17.10 C!

Name: `C!`

Stackeffect: b addr —

Attributes: ISO

Description: Store 8 bits of '`b`' at '`addr`' . In ciforth , running on the Intel architectures there are no restrictions regarding byte addressing.

See also: '`C@`' '`!`'

9.17.11 C@

Name: `C@`

Stackeffect: addr — b

Attributes: ISO

Description: Leave the 8 bit contents of memory address. In ciforth , running on the Intel architectures there are no restrictions regarding byte addressing.

See also: '`C!`' '`@`' '`2@`'

9.17.12 CELL+

Name: `CELL+`

Stackeffect: n1 — n2

Attributes: ISO

Description: Advance the memory pointer '`n1`' by one (in this case 32 bits) cell to '`n2`'. This is invaluable for writing portable code. Much of the library code of ciforth runs on 64, 32 and 16 bits systems, thanks to this.

9.17.13 CELLS

Name: `CELLS`

Stackeffect: n1 — n2

Attributes: ISO

Description: Return the equivalent of '`n1`' cells in bytes: '`n2`'. This is invaluable for writing portable code. Much of the library code of ciforth runs on 64, 32 and 16 bits systems, thanks to this.

See also: '`CELL+`'

9.17.14 CHAR+

Name: `CHAR+`

Stackeffect: n1 — n2

Attributes: ISO

Description: Advance the memory pointer '`n1`' by one character to '`n2`'. In ciforth this means one byte. Bytes are the address units ISO is talking about. Unfortunately the ISO standard has no way to address bytes.

See also: '`CELL+`'

9.17.15 CHARS

Name: `CHARS`

Stackeffect: n1 — n2

Attributes: ISO

Description: Return the equivalent of '`n1`' chars in bytes: '`n2`'. In ciforth this is a NOOP. Unfortunately the ISO standard has no way to address bytes.

See also: '`CELLS`'

9.17.16 CMOVE

Name: `CMOVE`

Stackeffect: from to count —

Attributes:

Description: Move the '`count`' quantity of characters beginning at address '`from`' to address '`to`' . The contents of address from is moved first proceeding toward high memory, such that memory propagation occurs. As the Intel 86-family is byte-addressing there are no restrictions in ciforth.

See also: '`MOVE`' '`CHARS`'

9.17.17 CORA

Name: `CORA`

Stackeffect: addr1 addr2 len — n

Attributes: CIF

Description: Compare the memory areas at '`addr1`' and '`addr2`' over a length '`len`' . For the first bytes that differ, return -1 if the byte from '`addr1`' is less (unsigned) than the one from '`addr2`', and 1 if it is greater. If all '`len`' bytes are equal, return zero. This is an abbreviation of `COMPARE-AREA` .

9.17.18 EM

Name: `EM`

Stackeffect: —- addr

Attributes:

Description: A constant leaving the address just above the highest memory in use by Forth.

See also: '`DP`' '`BM`'

9.17.19 ERASE

Name: `ERASE`

Stackeffect: addr n —

Attributes: ISO

Description: This is shorthand for '`0 FILL`'.

See also: '`BLANK`' '`FILL`'

9.17.20 FAR!

Name: `FAR!`

Stackeffect: u faraddr —

Attributes:

Description: Store 32 bits of '`u`' at segment address pair '`faraddr`'.

See also: '`FAR@`' '`!`' '`C!`' '`2!`'

9.17.21 FAR@

Name: `FAR@`

Stackeffect: faraddr — u

Attributes:

Description: Leave the 32 bit contents '`u`' of segment address pair '`faraddr`'.

See also: '`FAR!`' '`@`' '`C@`' '`2@`'

9.17.22 FARMOVE

Name: `FARMOVE`

Stackeffect: faraddr1 faraddr2 count —

Attributes:

Description: Move '`count`' bytes beginning at '`faraddr1`' to '`faraddr2`'. Segments that are zero are replaced by defaults: data segment _SEGMENT_ for source, extra segment _SEGMENT_ for destination. Copying is done from low to high offset, so there may be memory propagating. This is not easily seen from the _SEGMENT_'s, particularly not so in protected mode. As the Intel 86-family is byte-addressing there are no restrictions.

See also: '`MOVE`' '`CMOVE`'

9.17.23 FILL

Name: `FILL`

Stackeffect: addr u b —

Attributes: ISO,FIG

Description: If '`u`' is not zero, store '`b`' in each of '`u`' consecutive bytes of memory beginning at '`addr`' .

See also: '`BLANK`' '`ERASE`'

9.17.24 MOVE

Name: `MOVE`

Stackeffect: from to count —

Attributes: ISO

Description: Move '`count`' bytes beginning at address '`from`' to address '`to`', such that the destination area contains what the source area contained, regardless of overlaps. As the Intel 86-family is byte-addressing there are no restrictions. Because in ciforth bytes (address units) and characters are the same the difference with `CMOVE` amounts to `MOVE`

being an intelligent move.

9.17.25 PC!

Name: `PC!`

Stackeffect: b port —

Attributes:

Description: Store a byte '`b`' to the port address '`port`'. You can use ports only after asking permission (using e.g. '`ioperm`') and being super user.

See also: '`PC@`' '`PW!`' '`P@`' '`!`'

9.17.26 PC@

Name: PC@

Stackeffect: port — b

Attributes:

Description: Fetch a byte 'b' from the port address You can use ports only after asking permission (using e.g. 'ioperm') and being super user.

See also: 'PC!' 'PW@' 'PW!' '@'

9.17.27 PW!

Name: PW!

Stackeffect: n port —

Attributes:

Description: Store the 16-bit data 'n' to the 16-bits port address 'port'. You can use ports only after asking permission (using e.g. 'ioperm') and being super user.

See also: 'PW@' 'PC!' 'PC@' '!'

9.17.28 PW@

Name: PW@

Stackeffect: port — n

Attributes:

Description: Fetch the 16-bit contents 'n' from the port address 'port'. You can use ports only after asking permission (using e.g. 'ioperm') and being super user.

See also: 'PW!' 'PC@' 'PC!' '@'

9.17.29 TASK-SIZE

Name: TASK-SIZE

Stackeffect: —- n

Attributes:

Description: A constant leaving the total size of the per task areas: data stack, return stack, terminal input buffer plus user area. The data stack of the main task runs down to HERE .

See also: 'DSP@' 'RSP@' 'TIB'

9.17.30 TOGGLE

Name: TOGGLE

Stackeffect: addr b —

Attributes: NFIG

Description: Complement the contents of 'addr' by the bit pattern 'b' .

See also: 'XOR' '+! '

9.17.31 WITHIN

Name: WITHIN

Stackeffect: n1 n2 n3 — ff

Attributes: ISO

Description: Return a flag indicating that 'n1' is in the range 'n2' (inclusive) to 'n3' (non-inclusive). This works for signed as well as unsigned numbers. This is shorthand for: 'OVER - >R - R U<'

See also: '<' 'U<'

9.18 MISC

The wordset ‘`MISC`’ contains words that defy categorisation.

9.18.1 .SIGNON

Name: `.SIGNON`

Stackeffect: —

Attributes:

Description: Print a message identifying the version of this Forth. The name of the processor known from the environment query `CPU` is printed using the bizarre convention of a base-36 number. This is a tribute to those FIG-pioneers.

See also: ‘`ABORT`’ ‘`COLD`’

9.18.2 EXECUTE

Name: `EXECUTE`

Stackeffect: xt —

Attributes: ISO,FIG

Description: Execute the definition whose *execution token* is given by ‘`xt`’ . The *code field address* serves as an execution token. (It even has offset 0, but one should not assume that a *dea* is an execution token in portable code.)

See also: ‘`'`’ ‘`>CFA`’

9.18.3 NOOP

Name: `NOOP`

No stackeffect

Attributes:

Description: Do nothing. Primarily useful as a placeholder.

9.18.4 TASK

Name: `TASK`

No stackeffect

Attributes:

Description: A no-operation word which marks the boundary between the Forth system and applications.

See also: ‘`COLD`’

9.18.5 U0

Name: `U0`

Stackeffect: — addr

Attributes:

Description: A user variable, leaving the start address of the user area. This is for reference only. After initialisation user variables derive their start address from the return stack pointer. This might be used for task switching.

See also: ‘`USER`’ ‘`+ORIGIN`’

9.18.6 _

Name: _

Stackeffect: — x

Attributes:

Description: Leave an undefined value 'x'. Presumably it is to be dropped at some time, or it is a place holder.

9.19 MULTIPLYING

The original 16 bits Forth's have problems with overflow (see Section 9.21 [OPERATOR], page 119). Operators with intermediate results of double precision, mostly scaling operators, solve this and are present in the See Section 9.19 [MULTIPLYING], page 115, wordset. In this 32 bit Forth you will have less need. When you divide a 2.n digit number by an n digit number you get an n-digit quotient and an n-digit remainder. So naturally many operators in this set have a mix of double and single numbers in their stack diagram: they are *mixed magnitude* . All those have M in the name. Furthermore all operations are signed, with the exception of the ones starting with U . Those have unsigned operands exclusively. The irregularly named word *M/MOD* is a relic from the fig-forth era; preferably its alias *UDM/MOD* is to be used. Output formatting is done with *double* 's exclusively, and relies on this wordset and the 'DOUBLE' wordset. Operators with mixed magnitude and unsigned operators allow to build arbitrary precision operators from them in *high level* code.

9.19.1 */MOD

Name: */MOD

Stackeffect: n1 n2 n3 — n4 n5

Attributes: ISO,FIG

Description: Leave the quotient 'n5' and remainder 'n4' of the operation 'n1*n2/n3' (using *symmetric division*). A double precision intermediate product is used giving correct results, unless 'n4' overflows. Then ciforth will give a floating point exception.

See also: '*/' '/MOD'

9.19.2 */

Name: */

Stackeffect: n1 n2 n3 — n4

Attributes: ISO,FIG

Description: Leave the ratio 'n4 = n1*n2/n3' where all are signed numbers(using *symmetric division*). A double precision intermediate product is used giving correct results, unless 'n4' overflows. Then ciforth will give a floating point exception.

See also: '*/MOD' '/MOD'

9.19.3 FM/MOD

Name: FM/MOD

Stackeffect: d n1 — n2 n3

Attributes: ISO

Description: A mixed magnitude math operator which leaves the signed remainder 'n2' and signed quotient 'n3' from a double number dividend 'd' and divisor 'n1'. This is floored division, i.e. the remainder takes its sign from the divisor.

See also: 'SM/REM' 'UDM/MOD' '/' 'M*'

9.19.4 M*

Name: `M*`

Stackeffect: n1 n2 — d

Attributes: ISO,FIG

Description: A mixed magnitude math operation which leaves the double number '`d`' : the signed product of two signed number '`n1`' and '`n2`' .

See also: '`UDM/MOD`' '`SM/REM`' '`*`'

9.19.5 M/MOD

Name: `M/MOD`

Stackeffect: ud1 u2 — u3 ud4

Attributes: CIF,FIG

Description: This is an irregarly named alias of *UDM/MOD* .

See also: '`UDM/MOD`'

9.19.6 SM/REM

Name: `SM/REM`

Stackeffect: d n1 — n2 n3

Attributes: ISO

Description: A mixed magnitude math operator which leaves the signed remainder '`n2`' and signed quotient '`n3`' from a double number dividend '`d`' and divisor '`n1`'. This is a symmetric division, i.e. the remainder takes its sign from the dividend.

See also: '`UDM/MOD`' '`FM/MOD`' '`/`' '`M*`'

9.19.7 UDM/MOD

Name: `UDM/MOD`

Stackeffect: ud1 u2 — u3 ud4

Attributes: CIF,FIG

Description: An unsigned mixed magnitude math operation which leaves a double quotient '`ud4`' and remainder '`u3`' , from a double dividend '`ud1`' and single divisor '`u2`'. Unlike with *UM/MOD* the quotient can not overflow. The traditional alias forthcdefi(M/MOD) dates from the fig-forth era.

See also: '`UM/MOD`' '`SM/REM`' '`M*`'

9.19.8 UM*

Name: `UM*`

Stackeffect: u1 u2 — ud

Attributes: ISO

Description: A mixed magnitude math operation which leaves the double number '`ud`' : the unsigned product of two unsigned numbers '`u1`' and '`u2`' .

See also: '`UM/MOD`' '`M*`' '`*`'

9.19.9 UM/MOD

Name: `UM/MOD`

Stackeffect: ud u1 — u2 u3

Attributes: ISO

Description: Leave the unsigned remainder '`u2`' and unsigned quotient '`u3`' from the unsigned double dividend '`ud`' and unsigned divisor '`u1`' .

See also: '`UM*`' '`SM/REM`' '`/`'

9.20 OPERATINGSYSTEM

The wordset '`OPERATINGSYSTEM`' contains words that call the underlying operating system or functions available in the BIOS-rom.

9.20.1 ARGS

Name: `ARGS`

Stackeffect: — addr

Attributes:

Description: Return the addr of `ARGS` a user variable that contains a system dependant pointer to any arguments that are passed from the operating system to ciforth during startup.

In this ciforth it points to an area with the argument count, followed by a null ended array of arguments c-strings, then by a null ended array of environment c-strings. C-string are chars followed by a zero byte and no preceeding count.

See also: '`SYSTEM`'

9.20.2 BYE

Name: `BYE`

Stackeffect: —

Attributes: ISO

Description: Return to the host environment Linux.

See also: '`COLD`' '`EXIT-CODE`'

9.20.3 EXIT-CODE

Name: `EXIT-CODE`

Stackeffect: addr —

Attributes:

Description: Return '`addr`' the address of a variable with the exit code. Its content is passed to the host environment while going `BYE` . It is custom to return zero if there are no errors. Linux allows only single byte return codes.

See also: '`BYE`'

9.20.4 FORK

Name: `FORK`

Stackeffect: — pid/0/err

Attributes:

Description: Fork the process. We then run two processes. The return value is 0 for the child process, and '`pid`' for the mother process. If it is negative, it is an error.

See also: '`SYSTEM`'

9.20.5 MS

Name: `MS`

Stackeffect: n —

Attributes: ISO

Description: Wait for '`n`' milliseconds.

See also: '`KEY?`'

9.20.6 SHELL

Name: `SHELL`

Stackeffect: —addr

Attributes:

Description: Leave the address '`addr`' of a string variable, the name of a file that contains the command interpreter, or shell. This name may be changed and is used by the `SYSTEM` command. The name may contain a path and be at most 252 characters long. The default name is **/bin/sh** .

See also: '`SYSTEM`' '`$@`'

9.20.7 SYSTEM

Name: `SYSTEM`

Stackeffect: sc —

Attributes: ISO

Description: Have the operating system execute the command contained in the string '`sc`'.

See also: '`?ERRUR`' '`SHELL`'

9.20.8 XOS5

Name: `XOS5`

Stackeffect: n1 n2 n3 n4 n5 n—ret

Attributes:

Description: Do a traditional Unix type system call '`n`' (man 2) with parameters '`n1 n2 n3 n4 n5`'. '`ret`' is the return value of the call. If it is negative, it is mostly an error, such as known by **errno** . This makes available **all** facilities present in Linux.

See also: '`?ERRUR`'

9.20.9 XOS

Name: `XOS`

Stackeffect: n1 n2 n3 n—ret

Attributes:

Description: Do a traditional Unix type system call '`n`' (man 2) with parameters '`n1 n2 n3`'. '`ret`' is the return value of the call. If it is negative, it is mostly an error. Note This makes available all facilities present in Linux, except those with 4 or 5 parameters that are handled by `XOS5` .

See also: '`?ERRUR`'

9.20.10 ZEN

Name: `ZEN`

Stackeffect: sc — addr

Attributes:

Description: Leaves an address that contains a zero-ended (c-type) equivalent of '`sc`'. The same buffer (in fact `RW-BUFFER`) is reused, such that this word is not reentrant. The size available is SIZE_RWBUF minus one for the zero character. Use the word immediately, its intended used is passing parameters to the operating system.

See also: '`OPEN-FILE`' '`XOS`' '`XOS5`'

9.21 OPERATOR

The wordset '`OPERATOR`' contains the familiar operators for addition, multiplication etc. The result of the operation is always an integer number, so division can't be precise. On ciforth all division operations are compatible with *symmetric division* .

The ISO standard require a Forth to choose between floored or symmetric division for its standard operations. Divisions involving negative numbers have an interpretation problem. In any case we want the combination of `/` and `MOD` (remainder) to be such that you can get the original '`n`' back from the two values left by '`n m MOD`' and '`n m /`' by performing '`m * +`' . This is true for all Forth's. On ciforth the `/` is a *symmetric division* , i.e. '`-n m /`' give the same result as '`n m /`', but negated. The foregoing rule now has the consequence that '`m MOD`' has '`2|m|-1`' possible outcomes instead of '`|m|`' . This is very worrisome for mathematicians, who stick to the rule that '`m MOD`' has '`|m|`' outcomes: '`0 ... |m|-1`', or '`-|m|+1 ... 0` ' for negative numbers. (*floored division*).

9.21.1 *

Name: `*`

Stackeffect: n1 n2 — n3

Attributes: ISO,FIG

Description: Leave the signed product '`n3`' of two signed numbers '`n1`' and '`n2`' .

See also: '`+`' '`-`' '`/`' '`MOD`'

9.21.2 +

Name: `+`

Stackeffect: n1 n2 — sum

Attributes: ISO,FIG

Description: Leave the sum of '`n1`' and '`n2`' .

See also: '`-`' '`*`' '`/`' '`MOD`'

9.21.3 -

Name: `-`

Stackeffect: n1 n2 — diff

Attributes: ISO,FIG

Description: Leave the difference of '`n1`' and '`n2`' .

See also: '`NEGATE`' '`+`' '`*`' '`/`' '`MOD`'

9.21.4 /MOD

Name: `/MOD`

Stackeffect: n1 n2 — rem quot

Attributes: ISO,FIG

Description: Leave the remainder and signed quotient of '`n1`' and '`n2`' . The remainder has the sign of the dividend (i.e. *symmetric division*).

See also: '`*/MOD`' '`*/`' '`SM/REM`'

9.21.5 /

Name: `/`

Stackeffect: n1 n2 — quot

Attributes: ISO,FIG

Description: Leave the signed quotient of '`n1`' and '`n2`' . (using *symmetric division*).

See also: '`+`' '`-`' '`*`' '`MOD`' '`*/MOD`'

9.21.6 ABS

Name: `ABS`

Stackeffect: n — u

Attributes: ISO,FIG

Description: Leave the absolute value of '`n`' as '`u`' .

See also: '`DABS`'

9.21.7 LSHIFT

Name: `LSHIFT`

Stackeffect: u1 n — u2

Attributes: ISO

Description: Perform a **logical shift** of the bits of '`u1`' to the left by '`n`' places. It is an ambiguous condition if '`u1`' is greater than or equal to the number of bits in a cell. Put zero into the places uncovered by the shift.

See also: '`RSHIFT`' '`2*`'

9.21.8 MAX

Name: `MAX`

Stackeffect: n1 n2 — max

Attributes: ISO,FIG

Description: Leave the greater of two numbers.

See also: '`MIN`'

9.21.9 MIN

Name: `MIN`

Stackeffect: n1 n2 — min

Attributes: ISO,FIG

Description: Leave the smaller of two numbers.

See also: '`MAX`'

9.21.10 MOD

Name: `MOD`

Stackeffect: n1 n2 — mod

Attributes: ISO,FIG

Description: Leave the remainder of '`n1`' divided by '`n2`' , with the same sign as '`n1`' (i.e. *symmetric division*).

See also: '`+`' '`-`' '`*`' '`/`' '`MOD`' '`*/MOD`'

9.21.11 NEGATE

Name: `NEGATE`

Stackeffect: n1 — n2

Attributes: ISO,FIG

Description: Leave the two's complement of a number, i.e. '`n2`' is '`-n1`'

See also: '`-`'

9.21.12 RSHIFT

Name: `RSHIFT`

Stackeffect: u1 n — u2

Attributes: ISO

Description: Perform a **logical shift** of the bits of '`u1`' to the right by '`n`' places. It is an ambiguous condition if '`u1`' is greater than or equal to the number of bits in a cell. Put zero into the places uncovered by the shift.

See also: '`LSHIFT`' '`2/`'

9.22 OUTPUT

The wordset '`OUTPUT`' contains words to output to the terminal and such. See Section 9.10 [FILES], page 93, for disk I/O. See Section 9.1 [BLOCKS], page 60, for blocks.

9.22.1 (D.)

Name: `(D.)`

Stackeffect: d —sc

Attributes:

Description: Format a signed double number '`d`' field to the string '`sc`'. This a temporary string.

See also: '`OUT`' '`D.`' '`D.R`'

9.22.2 (D.R)

Name: `(D.R)`

Stackeffect: d n —sc

Attributes: ISO,FIG

Description: Format a signed double number '`d`' right aligned in a field '`n`' characters wide to the string '`sc`'. Enlarge the field, if needed. So a field length of 0 results effectively in free format.

See also: '`OUT`' '`D.`' '`D.R`' '`D.`'

9.22.3 ."

Name: `."`

No stackeffect

Attributes: ISO,FIG,I

Description: Used in the form: '`." cccc"`' In a definition it compiles an in-line string '`cccc`' (as if the denotation `"cccc"` was used) followed by `TYPE` . In ciforth `."` behaves the same way in interpret mode. In ciforth the number of characters has no limit. In ciforth `."` always has an effect on `HERE` during interpretation. In ISO programs you may only use this word during compilation. We recommend that '`." cccc"`' is replaced by '`"cccc" TYPE`'.

See also: '`OUT`'

9.22.4 .(

Name: `.(`

No stackeffect

Attributes: I

Description: In ciforth this is an alias for `."` , except that the string is closed with) instead of parsed as per `"` . In ISO programs you may only use this word while interpreting. We recommend that '`.( cccc)`' is replaced by '`"cccc" TYPE`'.

See also: '`OUT`' '`."`'

9.22.5 .R

Name: `.R`

Stackeffect: n1 n2 —

Attributes:

Description: Print a signed number '`n1`' right aligned in a field '`n2`' characters wide. Enlarge the field, if needed. So a field length of 0 results effectively in free format.

See also: '`OUT`' '`.`' '`(D.R)`'

9.22.6 .

Name: `.`

Stackeffect: n —

Attributes: ISO,FIG

Description: Print the number '`n1`' observing the current `BASE` , followed by a blank.

See also: '`OUT`' '`U.`' '`.R`' '`D.R`' '`D.`' '`(D.R)`'

9.22.7 ?

Name: `?`

Stackeffect: addr —

Attributes: ISO,FIG

Description: Print the value contained at the address '`addr`' observing the current `BASE` , followed by a blank.

See also: '`OUT`' '`.`'

9.22.8 CR

Name: `CR`

No stackeffect

Attributes: ISO,FIG

Description: Transmit character(s) to the terminal, that result in a "carriage return" and a "line feed". This means that the cursor is positioned at the start of the next line, if needed the display is scrolled.

See also: '`OUT`'

9.22.9 D.R

Name: `D.R`

Stackeffect: d n —

Attributes: ISO,FIG

Description: Print a signed double number '`d`' right aligned in a field '`n`' characters wide. Enlarge the field, if needed. So a field length of 0 results effectively in free format.

See also: '`OUT`' '`D.`' '`(D.R)`'

9.22.10 D.

Name: `D.`

Stackeffect: d —

Attributes: ISO,FIG

Description: Print the signed double number '`d`', observing the current `BASE` , followed by a blank.

See also: '`OUT`' '`.`' '`(D.)`' '`D.R`' '`(D.R)`'

9.22.11 EMIT

Name: `EMIT`

Stackeffect: c —

Attributes: ISO,FIG

Description: Transmit ASCII character '`c`' to the output device. For this ciforth all terminal I/O goes through `TYPE` . In this ciforth `EMIT` maintains `OUT` .

See also: '`TYPE`' '`OUT`'

9.22.12 ETYPE

Name: `ETYPE`

Stackeffect: addr count —

Attributes:

Description: Transmit '`count`' characters from '`addr`' to the standard error device. It is high level so error output can be redirected, by *revectoring* it. Or you may use redirection by Linux. In this ciforth strings may contain embedded *`LF`* 's with the effect of a new line at that point in the output.

See also: '`EMIT`' '`TYPE`'

9.22.13 OUT

Name: `OUT`

Stackeffect: — addr

Attributes: U

Description: A user variable that reflects the position at the current line of the output device where the next character transmitted will appear. The first position is zero. Only an explicit `CR` will reset `OUT` , not an *LF* embedded in a string that is `TYPE` d.

See also: '`EMIT`' '`TYPE`' '`CR`'

9.22.14 SPACES

Name: `SPACES`

Stackeffect: n —

Attributes: ISO,FIG

Description: If '`n`' is greater or equal to zero, display as much spaces.

See also: '`SPACE`' '`OUT`'

9.22.15 SPACE

Name: `SPACE`

No stackeffect

Attributes: ISO,FIG

Description: Transmit an ASCII blank to the output device.

See also: '`EMIT`' '`OUT`'

9.22.16 TYPE

Name: `TYPE`

Stackeffect: addr count —

Attributes: ISO,FIG

Description: Transmit '`count`' characters from '`addr`' to the output device. All terminal I/O goes through this word. It is high level so terminal I/O can be redirected, by *revectoring* it and the usual redirection or '`tee`'-ing by Linux. In this ciforth strings may contain embedded *LF* 's with the effect of a new line at that point in the output, however in that case `OUT` is not observed.

See also: '`EMIT`' '`OUT`' '`ETYPE`'

9.22.17 U.

Name: `U.`

Stackeffect: u —

Attributes: ISO

Description: Print the unsigned number '`u`' observing the current `BASE` , followed by a blank.

See also: '`OUT`' '`.`' '`.R`' '`D.R`' '`D.`' '`(D.R)`'

9.23 PARSING

The *outer interpreter* is responsible for parsing, i.e. it gets a word from the *current input source* and interprets or compiles it, advancing the `PP` pointer. The wordset '`PARSING`' contains the words used by this interpreter and other words that consume characters from the input source. In this way the outer interpreter need not be very smart, because its capabilities can be extended by new words based on those building blocks.

9.23.1 ?BLANK

Name: `?BLANK`

Stackeffect: c — ff

Attributes:

Description: For the character '`c`' return whether this is considered to be white space into the flag '`ff`' . At least the space, ASCII null, the tab and the carriage return and line feed characters are white space. In ciforth all control characters are considered white space.

See also: '`BL`' '`SPACE`'

9.23.2 CHAR

Name: `CHAR`

Stackeffect: — c

Attributes: ISO,I

Description: Parse a word and leave '`c`' the first non blank char of that word in the input source. If compiled the searching is done while the word being compiled is executed. Because this is so confusing, it is recommended that one never compiles or postpones `CHAR` .

See also: '`Prefix_&`' '`[CHAR]`' '`'`'

9.23.3 EVALUATE

Name: `EVALUATE`

Stackeffect: sc — ??

Attributes: ISO

Description: Interpret the content of '`sc`'. Afterwards return to the *current input source* .

See also: '`LOAD`' '`INCLUDE`' '`SET-SRC`'

9.23.4 INTERPRET

Name: `INTERPRET`

Stackeffect: ?? — ??

Attributes:

Description: Repeatedly fetch the next text word from the *current input source* and execute it (`STATE` is 0) or compile it (`STATE` is 1). A word is blank-delimited and looked up in the vocabularies of *search-order* . It can be either matched exactly, or it can match a prefix. If it is matched regularly the parse pointer remains after the blank delimiter as required by ISO. If a word matches a prefix, `INTERPRET` sets back the parse pointer to immediately after the prefix, A word that matches a prefix is called a *denotation* ; mostly this is a number. If it is not found at all, it is an `ERROR` . Prefixes are present in the *namespace* `ONLY` which comprises the minimum search order and is always present as the last wordlist in the search order. An application can add prefixes in whatever namespace is suitable. A *denotation* is a number, a double number, a character or a string etc. Denotations are handled respectively by the words `0` ... `F & "` and any other word of the `ONLY` wordlist, depending on the first characters. The denotation parsing is extendable, e.g. after loading a floating point package floating point numbers are recognized. character or characters.

A number is converted according to the current base. If a decimal point is found as part of a number, the number value that is left is a double number, otherwise it is a single number. Comma's in numbers are ignored by ciforth.

See also: '`NAME`' '`(NUMBER)`' '`NUMBER`' '`BLK`' '`DPL`'

9.23.5 NAME

Name: `NAME`

Stackeffect: — sc

Attributes: CI

Description: Parse the *current input source* for a word, i.e. blank-delimited as per `?BLANK` . Skip leading delimiters then advance the input pointer to past the next delimiter or past the end of the input source. Leave the word found as a string constant '`sc`'. As it goes with string constants, you may not alter its content, nor assume anything is appended. Note that this is more deserving of the name "WORD" than what is in the ISO standard, that can be used to parse lines. Contrast this to a `PREFIX` word that leaves the input pointer past itself.

See also: '`BLK`' '`WORD`' '`PP`'

9.23.6 PARSE

Name: `PARSE`

Stackeffect: c — sc

Attributes:

Description: Scan the *current input source* for the character '`c`' . Return '`sc`': a string from the current position in the input stream, ending before the first such character, or at the end of the current input source if it isn't there. The character is consumed. As it goes with *string constants*, you may not alter its content, nor assume anything is appended. So no leading delimiters are skipped.

See also: '`WORD`' '`NAME`'

9.23.7 PP@@

Name: `PP@@`

Stackeffect: —addr c

Attributes: CI

Description: Parse the *current input source* leaving the address '`addr`' of the next character '`c`' . If at the end of the input source, leave a pointer past the end and a zero. Advance the input pointer to the next character.

See also: '`BLK`' '`WORD`' '`PP`'

9.23.8 RESTORE-INPUT

Name: `RESTORE-INPUT`

Stackeffect: n1 n2 n3 3—f

Attributes: ISO,WANT

Description: Restore the input source stream from what was saved by SAVE-INPUT . ciforth is always able to restore the input across different input sources, as long as the input to be restored was not exhausted. This has the effect of chaining, and doesn't affect the return from nested calls be it interpreting, loading or evaluating. ciforth always returns a true into '`f`'. The input source abandoned will never be closed properly, so use should be restricted to the same input source.

See also: '`SAVE`' '`SAVE-INPUT`'

9.23.9 RESTORE

Name: `RESTORE`

Stackeffect: —

Attributes:

Description: This must follow a `SAVE` in the same definition. Restore the content of `SRC` from the return stack thus restoring the *current input source* to what it was when the `SAVE` was executed.

See also: '`SET-SRC`' '`RESTORE-INPUT`'

9.23.10 SAVE-INPUT

Name: `SAVE-INPUT`

Stackeffect: — n1 n2 n3 3

Attributes: ISO,WANT

Description: Get a complete specification of the input source stream. For ciforth this is the content of `SRC` . ciforth needs 3 cells, and is always able to `RESTORE` an input saved like this. In practice the use of `SAVE-INPUT` should be restricted to restoring input of the same stream.

See also: '`SAVE`' '`RESTORE-INPUT`'

9.23.11 SAVE

Name: `SAVE`

Stackeffect: —

Attributes:

Description: Save the content of `SRC` on the return stack to prepare for changing the *current input source* . This must be balanced by a `RESTORE` in the same definition. `CO` can be used between the two.

See also: '`SET-SRC`' '`SAVE-INPUT`'

9.23.12 SET-SRC

Name: `SET-SRC`

Stackeffect: sc —

Attributes:

Description: Make the *string constant* '`sc`' the *current input source* . This input is chained, i.e. exhausting it has the same effect as exhausting the input that called `SET-SRC` . In practice this word is almost always surrounded by call's to `SAVE` and `RESTORE` and then followed by a call to `INTERPRET` or some such.

See also: '`SRC`' '`SAVE`' '`EVALUATE`' '`INTERPRET`'

9.23.13 SOURCE

Name: `SOURCE`

Stackeffect: — addr n1

Attributes: ISO

Description: Return the address and length of the *current input source* .

See also: '`SRC`'

9.23.14 SRC

Name: `SRC`

Stackeffect: addr —

Attributes:

Description: Return the address '`addr`' of the *current input source* specification, allocated in the user area. It consists of three cells: the lowest address of the parse area, the non-inclusive highest address of the parse area and a pointer to the next character to be parsed. Changing '`SRC`' takes immediate effect, and must be atomic, by changing only the third cell or by e.g. using `SET-SRC` . The third cell has the alias '`PP`' . Words like '`>IN BLK SOURCE`' are secondary, and return their output by "second-guessing" '`SRC`' .

See also: '`SAVE`' '`RESTORE`' '`PP`' '`BLK`'

9.23.15 STATE

Name: `STATE`

Stackeffect: — addr

Attributes: ISO,U

Description: A user variable containing the compilation state. A non-zero value indicates compilation. In ciforth it then contains 1.

See also: '`[`' '`]`'

9.23.16 WORD

Name: `WORD`

Stackeffect: c —addr

Attributes: ISO,FIG,WANT

Description: Parse the '`current input source`' using '`c`' for a delimiter. Skip leading delimiters then advance the input pointer to past the next delimiter or past the end of the input source. Leave at '`addr`' a copy of the string, that was surrounded by '`c`' in the input source. This is an oldfashioned string to be fetched by `COUNT` , not `$@` . In ciforth the character string is positioned at the dictionary buffer `HERE` . `WORD` leaves the character count in the first byte, the characters, and ends with two or more blanks.

See also: '`NAME`' '`PARSE`' '`BLK `' '`IN`'

9.23.17 [CHAR]

Name: `[CHAR]`

Stackeffect: — c

Attributes: ISO I

Description: A compiling word. Parse a word. Add the run time behaviour: leave '`c`', the first non blank char of that word in the input source. In ciforth this word works also in interpret mode.

See also: '`Prefix_&`' '`CHAR`'

9.23.18 [

Name: `[`

No stackeffect

Attributes: ISO,FIG,I,

Description: Used in a colon-definition in form:

```
:  xxx    [  words   ]    more   ;
```

Suspend compilation. The words after `[` are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with `]`

See also: '`LITERAL` ' '`]`'

9.23.19 \

Name: `\`

No stackeffect

Attributes: ISO,I

Description: Used in the form: '`\ cccc`' Ignore a comment that will be delimited by the end of the current line. May occur during execution or in a colon-definition. Blank space after the word `\` is required.

See also: '`(`'

9.23.20]

Name: `]`

No stackeffect

Attributes: ISO,FIG

Description: Resume compilation, to the completion of a colon-definition.

See also: '`[`'

9.23.21 (

Name: `(`

No stackeffect

Attributes: ISO,FIG,I

Description: Used in the form: '`( cccc`'). Ignore a comment that will be delimited by a right parenthesis that must be in the same input source, i.e. on the same line for terminal input, or in the same string, block or file, when that is the input. It is an immediate word. So colon definitions can be commented too. A blank after the word '`(`' is required.

See also: '`\`'

9.24 SCREEN

Most of the *blocks* mass storage is used for *screen* 's that have 16 lines of 64 characters. They are used for source code and documentation. Each screen is one `BLOCK` as required by ISO. The '`SCREEN`' wordset contains facilities to view screens, and *load* them, that is compiling them and thus extending the base system. A system is customized by loading source screens, possibly one of these extension is a text editor for screens.

9.24.1 (LINE)

Name: `(LINE)`

Stackeffect: n1 n2 — sc

Attributes:

Description: Convert the line number '`n1`' and the screen '`n2`' to a string '`sc`' the content of the line (without the trailing new line).

See also: '`LINE`'

9.24.2 C/L

Name: `C/L`

Stackeffect: — c

Attributes:

Description: A constant that leaves the number of characters on a line of a standard screen: 64. The last character of each line is a *LF*.

See also: '`LIST`' '`LINE`'

9.24.3 INDEX

Name: `INDEX`

Stackeffect: from to —

Attributes:

Description: Print the first line of each screen over the inclusive range '`from`' , '`to`' . This is used to view the comment lines of an area of text on disc screens.

See also: '`LIST`'

9.24.4 LIST

Name: `LIST`

Stackeffect: n —

Attributes: ISO,FIG

Description: Display the ASCII text of *screen* '`n`'. The number of the screen is always printed in decimal. `SCR` contains the screen number during and after this process.

See also: '`BLOCK`'

9.24.5 LOAD

Name: `LOAD`

Stackeffect: n — ??

Attributes: ISO,FIG

Description: Interrupt the *current input source* in order to interpret screen '`n`' . While the screen is interpreted, it is locked, meaning that it cannot be associated with a different screen nr than '`n`' . The stack changes in according with the words executed. At the end of the screen, barring errors or forced changes, it continues with the interrupted input source.

See also: '`BLOCK`' '`LOCK`' '`#BUFF`' '`THRU`' '`QUIT`' '`EXIT`' '`-->`' '`LIST`'

9.24.6 R#

Name: `R#`

Stackeffect: — addr

Attributes: U

Description: A user variable which may contain the location of an editing cursor, or other file related function. Unused in the kernel of ciforth.

9.24.7 SCR

Name: `SCR`

Stackeffect: — addr

Attributes: U

Description: A user variable containing the screen number most recently reference by `LIST` .

See also: '`BLOCK`'

9.24.8 THRU

Name: `THRU`

Stackeffect: n1 n2 — ??

Attributes: ISO,FIG

Description: Interrupt the *current input source* in order to interpret screen '`n1`' through '`n2`' (inclusive). The stack changes in according with the words executed. At the end of the screens, barring errors or forced changes, it continues with the interrupted input source.

See also: '`#BUFF`' '`BLOCK`' '`LOAD`' '`QUIT`' '`EXIT`' '`-->`'

9.24.9 TRIAD

Name: `TRIAD`

Stackeffect: scr —

Attributes: FIG,WANT

Description: Display on the selected output device the three screens which include that numbered '`scr`' , beginning with a screen evenly divisible by three. Output is suitable for source text records, and includes a reference line at the bottom taken from line 0 of the first error screen.

See also: '`MESSAGE`' '`ERRSCR` '

9.24.10 (BLK)

Name: `(BLK)`

Stackeffect: — addr

Attributes: U

Description: When the standard word `BLK` is used, this user variable reflects the state of the *current input source* . It indicates the block number being interpreted, or zero, if input is being taken from the terminal input buffer. Changing `BLK` has no effect, and its content must be fetched before the *current input source* has changed.

See also: '`BLOCK`' '`TIB`'

9.24.11 BLK

Name: `BLK`

Stackeffect: — addr

Attributes: ISO

Description: When the standard word '`BLK`' is used, the content of the user variable '`(BLK)`' is refreshed to reflect the state of the current input source and its address is returned in '`addr`' . It indicates the block number being interpreted, or zero, if input is being taken from the terminal input buffer. In ciforth changing the content at '`addr`' has no effect, which may not be ISO compliant. Its content must be fetched before the *current input source* has changed.

See also: '`BLOCK`' '`TIB`'

9.25 SECURITY

The wordset '`SECURITY`' contains words that are used by control words to abort with an error message if the control structure is not correct. Some say that this is not Forth-like. You only need to know them if you want to extend the '`CONTROL`' wordset.

9.25.1 !CSP

Name: `!CSP`

No stackeffect

Attributes:

Description: Save the stack position in `CSP` . Used as part of the compiler security.

9.25.2 ?COMP

Name: `?COMP`

No stackeffect

Attributes:

Description: Issue error message if not compiling.

See also: '`?ERROR`'

9.25.3 ?CSP

Name: `?CSP`

No stackeffect

Attributes:

Description: Issue error message if stack position differs from value saved in '`CSP`' .

9.25.4 ?DELIM

Name: `?DELIM`

No stackeffect

Attributes:

Description: Parse a character and issue error message if it is not a blank delimiter.

9.25.5 ?EXEC

Name: `?EXEC`

No stackeffect

Attributes: FIG,WANT

Description: Issue an error message if not executing.

See also: '`?ERROR`'

9.25.6 ?LOADING

Name: `?LOADING`

No stackeffect

Attributes: FIG,WANT

Description: Issue an error message if not loading

See also: '`?ERROR`'

9.25.7 ?PAIRS

Name: `?PAIRS`

Stackeffect: n1 n2 —

Attributes:

Description: Issue an error message if '`n1`' does not equal '`n2`' . The message indicates that compiled conditionals do not match.

See also: '`?ERROR`'

9.25.8 ?STACK

Name: `?STACK`

No stackeffect

Attributes:

Description: Issue an error message if the stack is out of bounds.

See also: '`?ERROR`'

9.25.9 CSP

Name: `CSP`

Stackeffect: —- addr

Attributes: U

Description: A user variable temporarily storing the stack pointer position, for compilation error checking.

9.26 STACKS

The wordset '`STACKS`' contains words related to the *data stack* and *return stack* . Words can be moved between both stacks. Stacks can be reinitialised and the value used to initialise the *stack pointer* 's can be altered.

9.26.1 .S

Name: `.S`

Stackeffect: from to —

Attributes:

Description: Print the stack, in the current base. For stack underflow print nothing.

See also: '`LIST`'

9.26.2 >R

Name: `>R`

Stackeffect: n —

Attributes: ISO,FIG,C

Description: Remove a number from the *data stack* and place as the most accessable on the *return stack* . Use should be balanced with `R>` in the same definition.

See also: '`R@`'

9.26.3 CLS

Name: `CLS`

Stackeffect: i*x –

Attributes:

Description: Clear the data stack

See also: '`DSP!`'

9.26.4 DEPTH

Name: `DEPTH`

Stackeffect: — n1

Attributes: ISO

Description: Leave into '`n1`' the number of items on the data stack, before '`n1`' was pushed.

See also: '`DSP@`'

9.26.5 DSP!

Name: `DSP!`

Stackeffect: addr —

Attributes:

Description: Initialize the data stack pointer with ‘`addr`’ .

See also: ‘`DSP@`’ ‘`S0`’

9.26.6 DSP@

Name: `DSP@`

Stackeffect: — addr

Attributes:

Description: Return the address ‘`addr`’ of the data stack position, as it was before `DSP@` was executed.

See also: ‘`S0`’ ‘`DSP!`’

9.26.7 R0

Name: `R0`

Stackeffect: — addr

Attributes: U

Description: A user variable containing the initial location of the return stack.

See also: ‘`RSP!`’

9.26.8 R>

Name: `R>`

Stackeffect: — n

Attributes: ISO,FIG

Description: Remove the top value from the *return stack* and leave it on the *data stack* .

See also: ‘`>R`’ ‘`R@`’

9.26.9 R@

Name: `R@`

Stackeffect: — n

Attributes: ISO

Description: Copy the top of the return stack to the data stack.

See also: ‘`>R`’ ‘`<R`’

9.26.10 RDROP

Name: `RDROP`

Stackeffect: —

Attributes:

Description: Remove the top value from the return stack.

See also: ‘`>R`’ ‘`R@`’ ‘`R>`’

9.26.11 RSP!

Name: `RSP!`

Stackeffect: addr —

Attributes:

Description: Initialize the return stack pointer with '`addr`'.

See also: '`RSP@`' '`R0`'

9.26.12 RSP@

Name: `RSP@`

Stackeffect: — addr

Attributes:

Description: Return the address '`addr`' of the current return stack position, i.e. pointing the current topmost value.

See also: '`R0`' '`RSP!`'

9.26.13 S0

Name: `S0`

Stackeffect: — addr

Attributes: U

Description: A user variable that contains the initial value for the data stack pointer.

See also: '`DSP!`'

9.27 STRING

The wordset '`STRING`' contains words that manipulate strings of characters. In ciforth strings have been given their civil rights. So they are entitled to a *denotation* (the word `"`) and have a proper fetch and store. An (address length) pair is considered a *string constant* . It may be trimmed, but the data referring to via the address must not be changed. It can be stored in a buffer, a *string variable* , that contains in its first cell the count. Formerly this was in the first byte, and these are called *old fashioned string* 's (or less flatteringly: brain-damaged).

9.27.1 $!-BD

Name: `$!-BD`

Stackeffect: sc addr —

Attributes: WANT

Description: Store a *string constant* '`sc`' in the *old fashioned string* variable at address '`addr`', i.e. it can be fetched with `COUNT` . (Where would that BD come from?)

See also: '`COUNT`' '`$!`'

9.27.2 $!

Name: `$!`

Stackeffect: sc addr —

Attributes:

Description: Store a *string constant* '`sc`' in the string variable at address '`addr`'.

See also: '`$@`' '`$+!`' '`$C+`'

9.27.3 $+!

Name: `$+!`

Stackeffect: sc addr —

Attributes:

Description: Append a *string constant* '`sc`' to the string variable at address '`addr`'.

See also: '`$@`' '`$!`' '`$C+`'

9.27.4 $,

Name: `$,`

Stackeffect: sc — addr

Attributes:

Description: Allocate and store a *string constant* '`sc`' in the dictionary and leave its address '`addr`'.

See also: '`$@`' '`$!`'

9.27.5 $/

Name: `$/`

Stackeffect: sc c — sc1 sc2

Attributes:

Description: Find the first '`c`' in the *string constant* '`sc`' and split it at that address. Return the strings after and before '`c`' into '`sc1`' and '`sc2`' respectively. If the character is not present '`sc1`' is a null string (its address is zero) and '`sc2`' is the original string. Both '`sc1`' and '`sc2`' may be empty strings (i.e. their count is zero), if '`c`' is the last or first character in '`sc`' .

See also: '`$^`' '`CORA`' '`$@`'

9.27.6 $@

Name: `$@`

Stackeffect: addr — sc

Attributes:

Description: From address '`addr`' fetch a *string constant* '`sc`' . This is similar to `COUNT` with a cell-length instead of a char-length.

See also: '`$!`' '`$+!`' '`$C+`'

9.27.7 $C+

Name: `$C+`

Stackeffect: c addr —

Attributes:

Description: Append a char '`c`' to the string variable at address '`addr`'.

See also: '`$!`' '`$+!`' '`&`' '`^`' '`CHAR`'

9.27.8 $\

Name: `$\`

Stackeffect: sc c — sc1 sc2

Attributes:

Description: Find the first '`c`' in the *string constant* '`sc`' from the back and split it at that address. Return the strings before and after '`c`' into '`sc1`' and '`sc2`' respectively. If the character is not present 'sc1' is a null string (its address is zero) and 'sc2' is the original string. Both 'sc1' and 'sc2' may be empty strings (i.e. their count is zero), if 'c' is the last or first character in 'sc' .

See also: '$^' 'CORA' '$@'

9.27.9 $^

Name: $^

Stackeffect: sc c — addr

Attributes:

Description: Find the first 'c' in the *string constant* 'sc' and return its 'addr' if present. Otherwise return a *nil pointer*. A null string (0 0) or an empty string are allowed, and result in not found.

See also: '$/' 'CORA' '$@'

9.27.10 -TRAILING

Name: -TRAILING

Stackeffect: sc1 — sc2

Attributes: ISO

Description: Trim the *string constant* 'sc1' so as not to contain trailing blank space and leave it as 'sc2' .

See also: '?BLANK'

9.27.11 BL

Name: BL

Stackeffect: — c

Attributes: ISO.FIG

Description: A constant that leaves the ASCII value for "blank".

9.27.12 COUNT

Name: COUNT

Stackeffect: addr1 — addr2 n

Attributes: ISO,FIG

Description: Leave the byte address 'addr2' and byte count 'n' of a message text beginning at address 'addr1' . It is presumed that the first byte at 'addr1' contains the text byte count and the actual text starts with the second byte. Alternatively stated, fetch a *string constant* 'addr2 n' from the brain damaged string variable at 'addr1' .

See also: 'TYPE'

9.27.13 S"

Name: S"

Stackeffect: — addr1 n

Attributes: ISO

Description: Used in the form: 'S" cccc"' Leaves an in-line string 'cccc' (delimited by the trailing ") as a constant string 'addr1 n' . In ciforth the number of characters has no limit and using 'S"' has always an effect on HERE , even during interpretation. In ciforth a " can be embedded in a string by doubling it. In non-portable code denotations are recommended.

See also: '"'

9.28 SUPERFLUOUS

The wordset 'SUPERFLUOUS' contains words that are superfluous, because they are equivalent to small sequences of code. Traditionally one hoped to speed Forth up by coding these words directly.

9.28.1 0

Name: 0

Stackeffect: — 0

Attributes:

Description: Leave the number 0.

See also: 'CONSTANT'

9.28.2 1+

Name: 1+

Stackeffect: n1 — n2

Attributes: ISO,FIG

Description: This is shorthand for "'1 +'".

See also: 'CELL+' '1-'

9.28.3 1-

Name: 1-

Stackeffect: n1 — n2

Attributes: ISO

Description: This is shorthand for '1 -'.

See also: '1+ '

9.28.4 2*

Name: 2*

Stackeffect: n1 — n2

Attributes: ISO

Description: Perform an arithmetical shift left. The bit pattern of 'n1' is shifted to the left, with a result identical to '1 LSHIFT'.

See also: '2/ '

9.28.5 2/

Name: 2/

Stackeffect: n1 — n2

Attributes: ISO

Description: Perform an arithmetical shift right. The bit pattern of 'n1' is shifted to the right, except that the left most bit ("sign bit") remains the same, called "arithmetic shift". On this 2-complement machine this is the same as '2 /' .

See also: '2* '

9.28.6 Number_1

Name: `Number_1`

Stackeffect: — 1

Attributes:

Description: Leave the number 1.

See also: '`CONSTANT`'

9.28.7 Number_2

Name: `Number_2`

Stackeffect: — 2

Attributes:

Description: Leave the number 2.

See also: '`CONSTANT`'

9.29 WORDLISTS

The dictionary is subdivided in non-overlapping subsets: the *word list* 's (see Section 9.6 [DICTIONARY], page 81). They are created by the defining word `NAMESPACE` and filled by defining words while that *namespace* is `CURRENT` . They regulate how words are found; different vocabularies can have words with the same names.

A word list in the ISO sense has a *word list identifier* or *WID* , but no name which is inconvenient. We use namespace words created by the defining word `NAMESPACE` . They are used to manipulate the word list's that are associated with them. So namespaces are nearly the wordlist 's of the ISO standard, the primary difference is that they have a name. In most Forth literature namespaces are referred to as vocabularies, and a word `VOCABULARY` is used to create them.

9.29.1 ALSO

Name: `ALSO`

No stackeffect

Attributes: ISO

Description: Duplicate the topmost *WID* in the search order stack. If there were already 16 WID 's, ciforth looses the last one, i.e. the first added to the search order probably `FORTH` . This is probably fatal. `ONLY` remains in the search order and '`ONLY FORTH`' can be used to recover. Using namespaces you will not need to use this word, because in ciforth by executing a namespace you add it to the order. This is different than the words defined by the customary `VOCABULARY` that replaces the topmost wordlist.

See also: '`CONTEXT`' '`NAMESPACE`'

9.29.2 ASSEMBLER

Name: `ASSEMBLER`

No stackeffect

Attributes: WANT,NISO,FIG

Description: The name of the namespace that contains machine code definitions. In ciforth execution it pushes the associated word list to the top of the `CONTEXT` stack. (A traditional `VOCABULARY`

would replace the top.) This word makes only sense in combination with the words that belong to it. So it is present in loadable form in the screens file `forth.lab` .

See also: '`NAMESPACE`' '`LOAD`'

9.29.3 CONTEXT

Name: `CONTEXT`

Stackeffect: — addr

Attributes: FIG,U

Description: The context is the address where the WID is found of the wordlist that is searched first. In ciforth '`addr`' actually points to the *search order* , a row of WID 's ending with the minimum search order WID. The corresponding wordlists are searched in that order for definitions during interpretation. This row of WID's is allocated in the user variable space allowing for compilation in threads. It may contain up to 16 WID 's in this ciforth, while the ISO Search-Order wordset requires a capacity of at least 8.

See also: '`PRESENT`' '`NAMESPACE`' '`CURRENT`' '`USER`'

9.29.4 CURRENT

Name: `CURRENT`

Stackeffect: — addr

Attributes: FIG,U

Description: A user variable containing the WID of a namespace to which new words will be added. It is the *compilation word list* in the sense of the ISO standard. The WID has the structure of a *dictionary entry* . This allows to link in a new word between the link field of the WID and the next definition.

See also: '`NAMESPACE`' '`CONTEXT`'

9.29.5 DEFINITIONS

Name: `DEFINITIONS`

No stackeffect

Attributes: ISO

Description: Used in the form: '`cccc DEFINITIONS`' Make the top most *search order* word list, (context), the compilation word list (current). In the example, executing namespace name '`cccc`' add it to the top of the *search order* and executing `DEFINITIONS` will result in new definitions added to '`cccc`' .

See also: '`CONTEXT`' '`NAMESPACE`'

9.29.6 ENVIRONMENT

Name: `ENVIRONMENT`

No stackeffect

Attributes:

Description: The name of the `ENVIRONMENT` namespace. The associated *word list* contains environment queries. The names of words present in `ENVIRONMENT` are recognized by `ENVIRONMENT?` . This word list is not intended to be used as a `CONTEXT` word list; and only as a `CURRENT` whenever you want to add an environment query.

See also: '`NAMESPACE`'

9.29.7 FORTH

Name: `FORTH`

No stackeffect

Attributes: NISO,FIG

Description: The name of the primary namespace. Execution pushes the `FORTH` *WID* to the top of the *search order* . (For ISO-compliance it would replace the top, however the phrase '`ONLY FORTH`' has the effect required by ISO.) Until additional user *word list* 's are created, new user definitions become a part of `FORTH` .

See also: '`CONTEXT`' '`NAMESPACE`'

9.29.8 LATEST

Name: `LATEST`

Stackeffect: — addr

Attributes: FIG

Description: Leave the dictionary entry address '`addr`' of the topmost word in the `CURRENT` word list.

See also: '`NAMESPACE`'

9.29.9 ONLY

Name: `ONLY`

No stackeffect

Attributes: NISO

Description: Set the minimum search order, such that only *denotation* 's (numbers etc.) can be found plus the word `FORTH` . By using `FORTH` one can regain control towards a startup search order.

`ONLY` is actually a regular `NAMESPACE` . The associated *word list* contains mainly prefix words, that scan denotations and are described in the chapter `DENOTATIONS` . If you want to add a denotation, add it to the `ONLY` wordlist preferably. This prevents masking regular words that start with the prefix.

See also: '`PREFIX`' '`PP`' '`>WID`'

9.29.10 PREVIOUS

Name: `PREVIOUS`

No stackeffect

Attributes: ISO

Description: Pop the topmost *WID* from the search order stack. If empty still the `ONLY` search order is left.

See also: '`CONTEXT`' '`NAMESPACE`'

9.29.11 VOC-LINK

Name: `VOC-LINK`

Stackeffect: — addr

Attributes: U

Description: A user variable containing the *dictionary entry address* address of the word most recently created by `NAMESPACE` . All namespace names are linked to allow `FORGET` to find all vocabularies.

See also: '`NAMESPACE`' '`>VFA`'

Glossary Index

This index finds the glossary description of each word.

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Forth Word Index

This index contains *all* references to a word. Use the glossary index to find the glossary description of each word.

.

/

:

;

<

=

>

?

[

]

^

_

@

\

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Concept Index

Mostly the first reference is where the concept is explained. But sometimes in introductory and tutorial sections an explanation sometimes was considered too distracting.

O

P

R

S

T

U

V

W

Short Contents

Table of Contents