

ciforth manual

A close-to-ISO/computer-intelligence forth for the ARM.

This is a standard-ISO Forth (mostly, see the section portability) for the configuration called lina64:

- version 5.5.1
- 64-bits protected mode
- running under Linux
- contains security words
- the full ISO CORE set is present, possibly after loading
- headers with source fields

Albert van der Horst

Dutch Forth Workshop

Copyright © 2000,2024 Albert van der Horst

Permission is granted to copy with attribution. Program is protected by the GNU Public License.

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 "lina64" once it is installed on your system.

ciforth consists of three files:

- lina64 : 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 `ciarm.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 5.5.1. It is running in protected mode. It is running under Linux Blocks are allocated in files. A number has a precision of 64 bits. It calls linux system from Forth directly. It has compiler security, sacrificing some ISO compatibility. 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:
'lina64'

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 > lina64

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. `lina64` is case sensitive. If you want to change that consult the section "Common problems". (see Chapter 7 [Errors], page 37).

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/Frans van der Markt. This Forth is called ciforth and is made available by the D.F.W.. This publication 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/Frans van der Markt grants the following rights in writing:

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.

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 Aarch 64 . 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 lina64 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 `'lina64'` 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 `lina64`. As soon as you leave `lina64`, 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 `'lina64 -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 `lina64 -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 37, 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 `lina64` 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 `lina64`'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 `lina64` does.

`lina64` uses the *blocks* as a library by addition of the word `WANTED` and a convention. Starting with `'lina64 -w'` or most any option you have this facility available. If you are already in `lina64`, 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 37, Section Explanations. Apparently, `lina64` doesn't know about a forth word named `TUCK`, but after `"TUCK" WANTED` maybe it does. You may try again.

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 The library and different hardware.

The library is used to load different source depending on hardware and the operating system. Mostly for the operating system differences, this can be found out loading **CONFIG** . For hardware difference this cannot be made transparent. All low level hardware sources are turned off by default, by **AR1 ?AR2** etc. for 32 bits and **?ARa ?ARb** etc. for 64 bits.

?AR1 means Raspberrypi 1

?ARa means OrangePi One plus

?ARb means OrangePi 800

You have to specify ‘**TRUE CONFIG ARx**’ in the electives screen (screen 5) in order to use facilities like **gpio-on** and **TICKS** . Alternatively you can specify it in the source file.

4.1.4 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 **lina64** by ‘**lina64 -e**’. That is **e** for elective. ‘**lina64 -e**’ instructs **lina64** 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 `lina64` in a window, and use a separate window to start your editor. Try out things in `lina64`. 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 `lina64` 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 `lina64`. But at least you need not set up your Forth again, until your testing causes your Forth to crash.

4.1.5 Finding things out.

If you want to find things out you must start up `lina64` again by `'lina64 -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 66). 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 61). 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 wordlists 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 chosen 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 60). 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 64, 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 be 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 containing 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
ODEAD
$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 `lina64` 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 `lina64`-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 `lina64`, 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 “`lina64`”, 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 `lina64` 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 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 lina64 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 lina64, 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 lina64, 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 `_HOSTED_MSDOS_` (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 64 bits (8 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 `lina64` 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.

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 revectorred especially those for output. Output is revectorred 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 facilities 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 the kernel as delivered, but it is available after 'WANT ASSEMBLERi86'. Beware! The assembler can only be loaded on top of a CASE-SENSITIVE system.

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 - )| lina64 '
```

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 lina64 and load it with `'INCLUDE mysrc.frt'`

A Pentium 32 and a 8086 i86 Forth assembler are available in `forth.lab`. It must in due course be replaced by an ARM assembler. Screens are loaded in accordance with the system that is run, in other words an ARM system will simply refuse to load this assembler. The registers used by lina64 are called HIP, SPO, RPO and WOR. The mapping on actual processor registers is documented in the source.

It is essential that you regard lina64 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 critical 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 forth 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 see 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 0x100000 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 84, 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.

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.) `_HOSTED_MSDOS_` ((On a MSDOS system the need for this should be low, because of the redirection facilities available. However they are buggy.))

For this reason character output `CR` , `EMIT` and `TYPE` all go through a common word that can be changed. For `lina64` 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 `lina64` `ACCEPT` is left to the operating system, such that inputting to `lina64` 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 `'-1'` 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 `lina64` 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 omitted 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 `'-1'` 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 are 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 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 `lina64` 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 `lina64` 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 assembler is described in this manual, because it is not feasible to use it from the description in the source. The assembler is a general framework, with plugins for different CPU's. It is a separate project called ciasdis. The i86 lina contains in its library an assembler that can assemble source code that has been debugged using ciasdis. Such a facility is not yet present for the ARM CPU. The information in this chapter is relevant because all assembler plugins just work on the ARM lina. The 8080 is a nice example because it is small, but the details about the Pentium plug in are only present in the i86 lina manual. This chapter is about the assembler itself, not about how it is used in relation with ciforth.

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 lina64 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.

There is no assembler yet for 64 bit lina

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 : SUB , #1 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.

De **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 skimmed on write up. I didn't skip 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 (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
```

Contrarily here we adapt 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| Z| [BX]
```

This is completely unambiguous.

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. The way 'SIB' is handled is so clever, that sometimes I don't understand it myself. 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 a 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. And there is information there to the point that it allows to make a disassembler that is independant of the instruction information, one that will work for the 8086, look at the pop family. First I had

```
38 C1 02 4 1FAMILY, POP -- PUSH RST      ( B'| )
```

(cause I started from an existing assembler.) But of course `RST` (the restart instruction) has nothing to do with registers, so it gets separated out. Then the exception, represented by the hole '--' disappears. 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

Using `test.mak` (on a linux computer in `lina`) you can generate opcode sheets by `"make 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 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 37, 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 `SHOW:` you can see what completions of your opcode are legal.

- `'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.

- `'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 `B|` (byte size) with a `LEA`, opcode or with a `DI|` operand.

- `'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. For example `B|` (byte size) with a `LEA`, opcode or with a `DI|` operand.

- `'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

You may wonder why an optimizer for a computer language would be considered an AI application. This optimizer is not so much for a particular language as well related to a Computer Intelligence that has insight in her own code.

Different types of optimisations interfere and finding ones way through this certainly requires some heuristics. The bottom line is that an optimiser qualifies as an AI application.

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 contest, mostly with respect to everything.

6.1.3 Notations

In the following we will denote a stack effects as $\langle N - M \rangle$. This means that N items are popped and replaced by M new items. So 2DROP has the effect of $\langle 2 - 0 \rangle$. 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 FETCHPLUS(SPO,)'s and MINSTORE(SPO,)'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 its 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 preceding 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 applicable 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 than 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.

6.3 Concerning ARM

Optimisation for the ARM has to await an implementation of the PostIt FixUp assembler for the ARM.

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 up 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 `lina64` 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.

```
"/usr/lib/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.

7.2.4 Error -13 while trying I/O

If you get an error -13 this means you have insufficient privilege for an operation. An unexpected situation that provokes this error may be any access to io ports like general io *gpio*

or reading the clock. For those accesses one has to be root.

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 lina64, but maybe by other version of ciforth or even a differently configured lina64. 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- . You can compile even combination of DO and BRANCH controls after WANT -tricky-control-
- ‘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 .
- ‘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 123,, 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 American 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 philosophy 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* 64 bits is meant. Wherever it says *double* or “double precision number” a 128 bits number is meant.

The first line of each entry shows a symbolic description of the action of the procedure 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 ’	128 bit signed double integer: most significant portion with sign on top of stack
‘ dea ’	A <i>dictionary entry address</i> , the basic address of a Forth word from which all its properties can be found.
‘ f ’	logical <i>flag</i> : zero is interpreted as false, non-zero as true
‘ faraddr ’	a <selector:address> pair
‘ ff ’	<i>Forth flag</i> , a well-formed logical flag, 0=false, -1=true.
‘ false ’	a false <i>Forth flag</i> : 0
‘ n ’	64 bit signed integer number; it is also used for a 64-bit entity where it is irrelevant what number it represents
‘ sc ’	a <i>string constant</i> , 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 <i>Forth flag</i> : -1.
‘ u ’	64-bit unsigned integer, also used whenever a cell is considered as a bitset.
‘ ud ’	128-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 64-bit signed integers. For 128-bit signed numbers, the most significant part (with the sign) is on top.

All arithmetic is implicitly 64-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 *wordsets*, 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** , i.e. 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 may be changed and is used by the **BLOCK-INIT** command.

See also: ‘BLOCK-HANDLE’ ‘BLOCK-INIT’ ‘BLOCK-EXIT’ ‘\$@’

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’

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’

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 ciarm.lina64.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 procedure 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 block ‘n’. Multiple locks are possible, and require multiple unlocks. Probably, because 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’ ‘UNLOCK’ ‘#BUFF’

9.1.15 UNLOCK

Name: UNLOCK

Stackeffect: n —

Attributes: CI

Description: Unlock the block ‘n’. 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’ ‘#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 address ‘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’

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’ ‘SCREEN’

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 stack number by BLOCK before read or writing blocks.

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 first 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 See Section 9.6.14 [IMMEDIATE], page 69, words and numbers. They need special attention because these words in general execute during compilation forthtxref(IMMEDIATE). 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 in ciforth, the number will just remain on the stack.

See also: ‘`LITERAL`’ ‘`LIT`’

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 64 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 in ciforth, the number will just remain on the stack.

See also: ‘`LIT`’ ‘`LITERAL`’

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 64 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 just remain on the stack.

See also: ‘SLITERAL’ ‘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 cOPY FROM I86 checking.

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 or equal to ‘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’

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.

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 place 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)’ ‘?COMP’

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 OBRANCH 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’

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 (**+LOOP**) and uses ‘**addr**’ to calculate an offset to ‘**DO**’ . ‘**n2**’ 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 by means of **EXIT** or unstructured branching. In this Forth the parameters are the address after the loop .

See also: 'DO' 'LOOP' '+LOOP' '(BACK' '(FORWARD' 'EXIT'

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 OBRANCH 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 n1 — 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 OBRANCH 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 procedure 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 procedure. 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 procedure 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 procedure 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’

9.3.27 OBRANCH

Name: OBRANCH

Stackeffect: f —

Attributes: FIG,C

Description: The run-time procedure 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 procedure 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: ‘OBRANCH’ ‘(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’

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) point to 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`' '`,`'

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 created by NAMESPACE is not immediate. This may differ from VOCABULARY that is present 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 dummy dea that serves as *word list identifier* or WID in the sense of the ISO

standard. It has empty fields, except for the link field. The *link field address* contains the *dea* of the latest word of the namespace or a *nil pointer* if empty. 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' 'WORDLIST' 'VOC-LINK' '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. 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 the 'UP' is to be found at '0 +ORIGIN'. It is best to have '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 procedure, compiled by ;CODE, that fills in the code field of the most recently defined word with ‘addr’. It is used after **CREATE** instead of **DOES>** if the code following is assembler code instead of high level code.

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**. 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’. Stop compilation and terminate a new defining word ‘cccc’ by compiling (;CODE). Set **ASSEMBLER** to the top of the *search 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 procedure 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**.

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 **HEADER** initializes the *code field address* 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)**' '**LINK**' '**>CFA**' '**>DFA**' '**>FFA**' '**>LFA**' '**>NFA**' '**>SFA**' '**>XFA**'

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 '**dea**'. 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’ ‘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’ ‘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 only requires that denotators for decimal digits are present, one can always use a leading zero.

See also: ‘NUMBER’ ‘B’ ‘7’

9.5.6 Prefix_7

Name: `Prefix_7`

Stackeffect: — s/d

Attributes: I,P

Description: Implements NUMBER for numbers that start with 7 . Similar words are present for all decimal and hex digits.

See also: ‘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: ISO,FIG,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'** **'>NFA'**

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,I

Description: Used in the form:

' nnnn

It leaves the *execution token* (equivalent to the 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 **WORD** and **FIND** 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 significant cell of ‘d’ into the next available dictionary cell, advancing the *dictionary pointer* , then the least significant 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>’

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 (this is not a header *data field address*) to its ‘dea’. Where >BODY keeps working after *revectoring* , 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 8 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 **ALLLOT** .

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 string.

See also: ‘**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.

See also: ‘**PRESENT**’ ‘**CONTEXT**’ ‘**(FIND)**’ ‘**PREFIX**’ ‘**FIND**’

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.

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 *denotations* 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. 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. It is recommended that the only smart words present are prefix words.

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

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. Contrary to custom the most recent words is shown last.

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 address 'dea' return its *code field address* 'addr'. By jumping indirectly via this address the definition 'dea' is executed. This is the address that is compiled within high level definitions, so it serves as an execution token. 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 address return its *data field address* ‘addr’. This points to the code for a code word, 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 address 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 address 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 address 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 address 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 address return the *source field address* ‘addr’.

See also: ‘Prefix_’ ‘HEADER’

9.6.28 >VFA

Name: >VFA

Stackeffect: dea — cfa

Attributes:

Description: Given the dictionary entry address 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 address '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 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 .

9.6.31 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'

9.6.32 FOR-WORDS

Name: FOR-WORDS

Stackeffect: x1...xn xt wid — x1...xn

Attributes:

Description: For all words from a word list identified by 'wid' execute 'xt' with as data 'x1..xn' plus the *dea* of those words. 'xt' must have the stack diagram 'x1..xn dea --- x1..xn'. Note that you can use the dea of any word as a WID and the remainder of the word list will be searched.

See also: 'FOR-VOCS' 'EXECUTE'

9.6.33 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**’

9.6.34 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**’

9.6.35 ~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.

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 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 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 revector to give more elaborate diagnostics.

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:

Description: The run time action of **ABORT"** .

See also: 'WANT'

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 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 u — fileid ior

Attributes: ISO

Description:

Create a file with name ‘sc’ and file access privileges ‘u’. 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.

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.

See also: 'LOAD'

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.

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 73, wordset. See Section 9.19 [MULTIPLYING], page 99, 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’

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 BASEName: **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.

See also: ‘**NUMBER**’ ‘?**BLANK**’

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 input, or a *nil pointer* . During output it is free to use, e.g. to hold the output column location of a decimal point, in user generated formatting.

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, so one can access or modify the boot-up parameters. During run time user variables are fetched from the current user area, via a pointer at ‘0 +ORIGIN ’. This can be swapped to get a fresh set of user variables, for multi-asking. One can access or modify the boot-up parameters, prior to saving a customised boot image. This will affect the initialisation by COLD .

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: Initialise 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. Show signon message and restart via ABORT . May be called from the terminal to remove application programs and restart, as long as there are no new vocabularies with definitions. But it is better to say BYE to Forth and start again.

See also: ‘WARM’ ‘INIT’ ‘OPTIONS’

9.12.4 DEV-MEM

Name: DEV-MEM

Stackeffect: — addr

Attributes:

Description: A variable containing the file descriptor where the device memory is accessible. If it contains a negative number, it indicates an error, probably a lack of privilege.

See also: ‘VMA-IO’ ‘PC!’

9.12.5 INIT

Name: INIT

No stackeffect

Attributes: FIG

Description: Initialise or reinitialise the system. Reset the data stack, the wordlists, 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`’ ‘`SO`’

9.12.6 MMAP-IO

Name: `MMAP-IO`

No stackeffect

Attributes:

Description: Fills VMA-IO with the memory base address or else an error code. *DEV-MEM* has been filled at *COLD*. If that failed VMA-IO will contain a same negative number, that indicates the error.

See also: ‘`P!`’ ‘`VMA-IO`’

9.12.7 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 revector to show debugging info or give a prompt.

See also: ‘`QUIT`’ ‘`COLD`’

9.12.8 OPTIONS

Name: `OPTIONS`

Stackeffect: `ft` — `f2`

Attributes: `CI`

Description: Handle command line option. If an option is passed during startup execute the screen corresponding to the option letter and return a false flag into ‘`f2`’, otherwise leave a true flag. During startup this flag decides whether the signon message is displayed. An option can always decide to execute `.SIGNON`. By redefining `OPTIONS` as ‘`DROP 0`’

See also: ‘`COLD`’

9.12.9 PERIPHERALS

Name: `PERIPHERALS`

Stackeffect: — `addr`

Attributes:

Description: A variable containing the address offset of peripherals, like digital I/O pins and I2C interfaces reside in *DEV-MEM*

pseudo-file.

See also: ‘`P!`’ ‘`PC!`’

9.12.10 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**’

9.12.11 VMA-IO

Name: **VMA-IO**

Stackeffect: — addr

Attributes:

Description: A variable containing the address where the memory mapped IO resides.

See also: ‘**P!**’ ‘**PC!**’

9.12.12 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**’

9.13 INPUT

The wordset ‘**INPUT**’ contains words to get input from the terminal and such. See Section 9.10 [FILES], page 78, for disk I/O. See Section 9.1 [BLOCKS], page 46, 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.

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 address, until a **RET** is received. The simple tty editing functions of Linux are observed, such that the “erase” (delete

a character) and “kill” (delete a line) characters. Typically these are the backspace key and ^U. Note that excess characters after ‘count’ are ignored. The number of characters not including the *RET* is returned into ‘n’.

See also: ‘(ACCEPT)’ ‘KEY’ ‘KEY?’ ‘(ACCEPT)’

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 characters after a *RET* are available) ‘REMAINDER 2@’ contains the part of TIB that is not yet consumed by (ACCEPT) , and outside the reach of SRC .

All system call errors result in an exception. For redirected I/O this word may generate an end-of-pipe exception.

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 termio 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: ‘SET-TERM’

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 ‘termio’.

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: $n1 \text{ --- } n1$ (if zero) / $n1 \text{ --- } n1 \ n1$ (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: $n \text{ ---}$

Attributes: ISO,FIG

Description: Drop the number from the stack.

See also: ‘**DUP**’ ‘**SWAP**’ ‘**OVER**’

9.14.7 DUP

Name: DUP

Stackeffect: $n \text{ --- } n \ n$

Attributes: ISO,FIG

Description: Duplicate the value on the stack.

See also: ‘**OVER**’ ‘**SWAP**’ ‘**DROP**’

9.14.8 NIP

Name: NIP

Stackeffect: $n1 \ n2 \text{ --- } n2$

Attributes: ISO

Description: Drop the second number from the stack.

See also: ‘**DUP**’ ‘**DROP**’ ‘**SWAP**’ ‘**OVER**’

9.14.9 OVER

Name: OVER

Stackeffect: $n1 \ n2 \text{ --- } n1 \ n2 \ n1$

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: $n1 \ n2 \ n3 \text{ --- } n2 \ n3 \ n1$

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 described into library have nothing particular in common, except that they are available in the *LAB* and not the *ciforth* kernel. They are either words from the library that are sufficiently useful to merit being described here, or legacy words that were in the kernel in earlier versions. All of them can be loaded from the library by ‘WANT’ ‘LEGACY_WORD’.

9.15.1 L_>IN

Name: L_>IN

Stackeffect: — addr

Attributes: ISO,WANT

Description: Hides and replaces the user variable PP . All parsing words use and move the value of that user variable. Return a variable that contains the offset from the start within the current input text buffer (terminal or disc) from which the next text will be accepted. The legacy >IN is not actually a variable: its content must be fetched immediately, and changing it has no effect.

See also: ‘(>IN)’

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 64 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’ if 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 64-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: n1 n2 — n3

Attributes: ISO,FIG

Description: Leave the bitwise logical exclusive or of two 64-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 64 bits of n at ‘addr’ .

See also: ‘@’ ‘C!’ ‘2!’ ‘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 64 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 64 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 64 bit contents ‘n’ of ‘addr’.

See also: ‘!’ ‘C@’ ‘2@’ ‘P@’ ‘PC@’

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 BIC

Name: BIC

Stackeffect: mask portaddr — n

Attributes: ISO,FIG

Description: ‘portaddr’ is the address of a 32-bit entity irrespective of cell-size. The bits present in ‘mask’ are cleared from ‘addr’ without affecting other bits.

See also: ‘BIS’ ‘!’ ‘@’ ‘PW@’ ‘PW!’

9.17.9 BIS

Name: BIS

Stackeffect: mask portaddr — n

Attributes: ISO,FIG

Description: ‘portaddr’ is the address of a 32-bit entity irrespective of cell-size. The bits present in ‘mask’ are set in ‘addr’ without affecting other bits.

See also: ‘BIC’ ‘!’ ‘@’ ‘PW@’ ‘PW!’

9.17.10 BLANK

Name: BLANK

Stackeffect: addr count —

Attributes: ISO

Description: This is shorthand for “BL FILL ”.

9.17.11 BM

Name: BM

Stackeffect: — addr

Attributes:

Description: A constant leaving the address of the lowest memory in use by Forth.

See also: ‘DP’ ‘EM’

9.17.12 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@’ ‘!’

9.17.13 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@’

9.17.14 CELL+

Name: CELL+

Stackeffect: n1 — n2

Attributes: ISO

Description: Advance the memory pointer ‘n1’ by one (in this case 64 bits) cell to ‘n2’. This is invaluable for writing portable code. Much of the library code of ciforth runs on both 16 and 32 bits systems, thanks to this.

9.17.15 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 both 16 and 32 bits systems, thanks to this.

See also: ‘CELL+’

9.17.16 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+’

9.17.17 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’

9.17.18 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 this ciforth byte-addressing there are no restrictions in ciforth.

See also: ‘MOVE’ ‘CHARS’

9.17.19 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. It would have been named ‘COMPARE’, if that were not taken by ISO.

9.17.20 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’

9.17.21 ERASE

Name: ERASE

Stackeffect: addr n —

Attributes: ISO

Description: This is shorthand for ‘O FILL’.

See also: ‘BLANK’ ‘FILL’

9.17.22 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’

9.17.23 L!

Name: L!

Stackeffect: n addr —

Attributes:

Description: Store the 32-bit contents of ‘n’ at ‘addr’ .

See also: ‘L@’ ‘!’ ‘C!’ ‘2!’ ‘PW!’ ‘PC!’

9.17.24 L@

Name: L@

Stackeffect: addr — n

Attributes: ISO,FIG

Description: Leave the 32 bit contents ‘n’ of ‘addr’ .

See also: ‘!’ ‘C@’ ‘2@’ ‘PW@’ ‘PC@’

9.17.25 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 this cforth is byte-addressing there are no restrictions. Because in cforth bytes (address units) and characters are the same the difference with CMOVE amounts to MOVE

being an intelligent move.

9.17.26 P!

Name: P!

Stackeffect: n addr —

Attributes:

Description: Store all 64 bits of n at io-location ‘addr’ . Actually his is the same as !

See also: ‘@’ ‘C!’ ‘2!’ ‘PW!’ ‘PC!’

9.17.27 P@

Name: P@

Stackeffect: addr — n

Attributes:

Description: Leave the 64 bit contents ‘n’ of ‘addr’ which is an io mapped peripheral.

See also: ‘!’ ‘C@’ ‘@’ ‘P!’ ‘PC@’

9.17.28 TOGGLE

Name: TOGGLE

Stackeffect: addr b —

Attributes: NFIG

Description: Complement the contents of ‘addr’ by the bit pattern ‘b’ .

See also: ‘XOR’ ‘+!’

9.17.29 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. What is taken into account by user variables is the initialisation variable at ‘0 +ORIGIN’ . 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 103). Operators with intermediate results of double precision, mostly scaling operators, solve this and are present in the ‘MULTIPLYING’ wordset. In this 64 bit Forth you will have less need. Formatting is done with *double* ’s exclusively, and relies on this wordset. Operators with mixed precision 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’ or ‘n5’ overflows. ‘n1 n2 * n3 /’ gives an incorrect answer as soon as ‘n1 n2 *’ overflows.

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.

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 numbers ‘n1’ and ‘n2’ .

See also: ‘UDM/MOD’ ‘SM/REM’ ‘*’

9.19.5 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’ ‘/’ ‘M*’

9.19.6 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’.

See also: ‘UM/MOD’ ‘SM/REM’ ‘M*’

9.19.7 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*’ ‘*’

9.19.8 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-strings 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 counted string, 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: ‘BLOCK’ ‘?ERROR’ ‘?ERRUR’

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 XOS7

Name: XOS7

Stackeffect: **n1 n2 n3 n4 n5 n6 n7 n—ret**

Attributes:

Description: Do a traditional Unix type system call ‘**n**’ (man 2) with parameters ‘**n1 n2 n3 n4 n5 n6 n7**’. ‘**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’

9.20.10 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, such as known by **errno** . This makes available all facilities present in Linux, except those with 4 or 5 parameters that are handled by XOS5 .

See also: ‘?ERRUR’

9.20.11 ZEN

Name: ZEN

Stackeffect: **sc — addr**

Attributes:

Description: Leaves an address that contains a zero-ended (c-type) equivalent of ‘**sc**’. The same buffer is reused, such that this word is not reentrant. Use the word immediately, e.g. its intended used is passing parameters to the operating system. In fact this is **RW-BUFFER** .

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 ARSHIFT

Name: ARSHIFT

Stackeffect: $n1\ n \rightarrow n2$

Attributes: ISO

Description: Perform a **arithmetic shift** of the bits of ‘ $n1$ ’ to the right by ‘ n ’ places. Put the highest bit into the places uncovered by the shift.

See also: ‘LSHIFT’ ‘2/’

9.21.8 LSHIFT

Name: LSHIFT

Stackeffect: $u1\ n \rightarrow u2$

Attributes: ISO

Description: Perform a **logical shift** of the bits of ‘ $u1$ ’ to the left by ‘ n ’ places. Put zero into the places uncovered by the shift.

See also: ‘RSHIFT’ ‘2*’

9.21.9 MAX

Name: MAX

Stackeffect: $n1\ n2 \rightarrow \max$

Attributes: ISO,FIG

Description: Leave the greater of two numbers.

See also: ‘MIN’

9.21.10 MIN

Name: MIN

Stackeffect: $n1\ n2 \rightarrow \min$

Attributes: ISO,FIG

Description: Leave the smaller of two numbers.

See also: ‘MAX’

9.21.11 MOD

Name: MOD

Stackeffect: $n1\ n2 \rightarrow \text{mod}$

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’

9.21.12 NEGATE

Name: NEGATE

Stackeffect: $n1 \rightarrow n2$

Attributes: ISO,FIG

Description: Leave the two’s complement of a number, i.e. ‘ $n2$ ’ is ‘ $-n1$ ’

See also: ‘-’

9.21.13 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. 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 78, for disk I/O. See Section 9.1 [BLOCKS], page 46, 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’

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.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 cforth 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’ ‘ETypes’

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 **IN** 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 cforth 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 not 1) 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. A word that matches a prefix is called a *denotation* ; mostly this is a number. Prefixes are present in ONLY which is the last wordlist in the search order, and the minimum search order. If it is not found at all, it is an ERROR . 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 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 are ignored in ciforth.)

See also: 'WORD' '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.

See also: 'BLK' 'WORD' 'IN'

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 next character 'c' and its address 'addr' . 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 preceded by a call to SAVE and followed by a call to INTERPRET .

See also: 'SRC' '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’ . 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’ ‘#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:

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, and 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 accessible on the *return stack* . Use should be balanced with R> in the same definition.

See also: ‘R@’

9.26.3 CLS

Name: CLS

Stackeffect: $i \times -$

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: $\text{addr} -$

Attributes:

Description: Initialize the data stack pointer with ‘addr’ .

See also: ‘S0’ ‘DSP@’

9.26.6 DSP@

Name: DSP@

Stackeffect: $- \text{addr}$

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: $- \text{addr}$

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: ‘R0’ ‘RSP@’

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: ‘\$^’ ‘COR’ ‘\$@’

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: ‘\$/’ ‘COR’ ‘\$@’

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**’. This word should not be used.

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. This is the same as ‘S>D 2 FM/MOD NIP’. It is not the same as ‘2 /’ , nor is it the same as ‘1 RSHIFT’.

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 66). 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 no name, but a *word list identifier* or *WID* , 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 vocabularies are nearly the wordlist ’s of the ISO standard, the primary difference is that they have a name.

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 loses the last one. This is not counting the **ONLY** search order. You seldomly want to use this word, because in ciforth by naming a namespace you add it to the order.

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. (For ISO-compliance it would replace the top.) **ASSEMBLER** is not immediate. 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 make testthe 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 by these fields to allow **FORGET** to find all vocabularies.

See also: ‘**NAMESPACE**’ ‘**>VFA**’

Glossary Index

This index finds the glossary description of each word.

!

!	93
!CSP	116

#

#	81
#>	80
#BUFF	46
#S	80

\$

\$!	120
\$!-BD	120
\$+!	120
\$,	120
\$/	120
\$^	121
\$@	120
\$\	121
\$C+	121

,

,	66
---	----

(

(113
(+LOOP)	58
(;)	58
(;CODE)	63
(>IN)	88
(?DO)	58
(ABORT")	77
(ACCEPT)	86
(BACK)	58
(BLK)	115
(BUFFER)	49
(CREATE)	63
(D.)	106
(D.R)	106
(DO)	59
(FIND)	70
(FORWARD)	59
(LINE)	114
(NUMBER)	82

*

*	104
*/	100
*/MOD	100

+

+	104
+!	93
+BUF	50
+LOOP	52
+ORIGIN	84

,

,	66
---	----

—

-	104
-->	46
-TRAILING	121

.

.	107
."	106
.(106
.R	107
.S	117
.SIGNON	98

/

/	104
/MOD	104

;

;	60
;CODE	63

<

<	92
<#	81
<>	92

=

=	92
---	----

>

>.....	92
>BODY.....	67
>CFA.....	70
>DFA.....	71
>FFA.....	71
>LFA.....	71
>NFA.....	71
>NUMBER.....	81
>PHA.....	71
>R.....	117
>SFA.....	71
>VFA.....	72
>WID.....	72

?

?.....	107
?BLANK.....	109
?COMP.....	116
?CSP.....	116
?DELIM.....	116
?DISK-ERROR.....	46
?DO.....	53
?DUP.....	90
?ERROR.....	75
?ERRUR.....	75
?EXEC.....	116
?LOADING.....	116
?PAIRS.....	117
?STACK.....	117

[

[.....	113
['].....	70
[CHAR].....	112
[COMPILE].....	52

]

].....	113
--------	-----

-

-.....	99
_FIRST.....	50
_LIMIT.....	50
_PREV.....	51

@

@.....	94
--------	----

\.....	113
--------	-----

~

~MATCH.....	73
-------------	----

0

0.....	122
0<.....	91
0=.....	91
OBRANCH.....	59

1

1+.....	122
1-.....	122

2

2!.....	94
2*.....	122
2,.....	67
2/.....	123
2@.....	94
2DROP.....	89
2DUP.....	89
2OVER.....	89
2SWAP.....	89

A

ABORT.....	84
ABORT".....	75
ABS.....	104
ACCEPT.....	86
AGAIN.....	53
ALIGN.....	94
ALIGNED.....	94
ALLOT.....	67
ALSO.....	123
AND.....	92
ARGS.....	101
ARSHIFT.....	105
ASSEMBLER.....	123

B

B/BUF.....	47
BACK).....	59
BASE.....	81
BEGIN.....	53
BIC.....	94
BIS.....	95
BL.....	121
BLANK.....	95
BLK.....	115
BLOCK.....	48
BLOCK-EXIT.....	47
BLOCK-FILE.....	47
BLOCK-HANDLE.....	47
BLOCK-INIT.....	47
BLOCK-READ.....	48
BLOCK-SEEK.....	50
BLOCK-WRITE.....	48
BM.....	95
BODY>.....	67
BRANCH.....	59
BYE.....	101

C

C!	95
C,	67
C/L	114
C@	95
CATCH	76
CELL+	95
CELLS	96
CHAR	109
CHAR+	96
CHARS	96
CLOSE-FILE	78
CLS	118
CMOVE	96
COLD	84
colon	62
CO	54
CONSTANT	60
CONTEXT	124
CORA	96
CORE	74
COUNT	121
CPU	74
CR	107
CREATE	60
CREATE-FILE	78
CSP	117
CURRENT	124

D

D+	73
D.	107
D.R	107
DABS	73
DATA	61
DECIMAL	81
DEFINITIONS	124
DELETE-FILE	78
DEPTH	118
DEV-MEM	84
DIGIT	82
DISK-ERROR	48
DLITERAL	51
DNEGATE	74
DO	54
DOES>	61
DP	68
DPL	83
DROP	90
DSP!	118
DSP@	118
DUP	90

E

ECL	83
ELSE	54
EM.	97
EMIT	108
EMPTY-BUFFERS	48
ENVIRONMENT	124
ENVIRONMENT?	74
ERASE	97
ERROR	76
ERRSCR	76
ETYPE	108
EVALUATE	109
EXECUTE	99
EXIT	55
EXIT-CODE	102

F

FENCE	72
FILE-POSITION	78
FILL	97
FIND	68
FLD	83
FLUSH	50
FM/MOD	100
FOR-VOCS	72
FOR-WORDS	72
FORGET	68
FORGET-VOC	72
FORK	102
FORTH	125
FORWARD)	60
FOUND	68

G

GET-FILE	78
----------	----

H

HANDLER	77
HEADER	63
HERE	68
HEX	82
HIDDEN	73
HLD	83
HOLD	82

I

I.	55
ID.	69
IF	55
IMMEDIATE	69
INCLUDE	79
INCLUDED	79
INDEX	114
INIT	84
INTERPRET	110
INVERT	92

J

J 55

K

KEY 87

KEY? 87

L

L! 97

L_>IN 91

L@ 97

LATEST 125

LEAVE 56

LINK 64

LIST 114

LIT 52

LITERAL 51

LOAD 114

LOCK 49

LOOP 56

LSHIFT 105

M

M* 100

MAX 105

MAX-USER 61

MESSAGE 76

MIN 105

MMAP-IO 85

MOD 105

MOVE 97

MS 102

N

NAME 74, 110

NAMESPACE 61

NEGATE 105

NIP 90

NOOP 99

Number_1 123

Number_2 123

NUMBER 83

O

OFFSET 50

OK 85

ONLY 125

OPEN-FILE 79

OPTIONS 85

OR 93

OUT 108

OVER 90

P

P! 98

P@ 98

PAD 69

PARSE 110

PERIPHERALS 85

POSTPONE 51

PP 87

PP@@ 110

Prefix_ " 64

Prefix_& 64

Prefix_+ 65

Prefix_- 65

Prefix_^ 66

Prefix_0 65

Prefix_7 65

Prefix_B 65

Prefix_TICK 65

PREFIX 69

PRESENT 69

PREVIOUS 125

PUT-FILE 79

Q

QUIT 86

R

R# 114

R> 118

R@ 119

RO 118

RDROP 119

READ-FILE 79

RECURSE 56

REFILL-TIB 88

REMAINDER 88

REPEAT 56

REPOSITION-FILE 80

RESTORE 111

RESTORE-INPUT 111

ROT 90

RSHIFT 106

RSP! 119

RSP@ 119

RUBOUT 87

RW-BUFFER 80

S

S" 122

S>D 74

SO 119

SAVE 111

SAVE-INPUT 111

SCR 115

SDLITERAL 52

SET-SRC 111

SET-TERM 88

SHELL 102

SIGN 82

SKIP 57

SM/REM 100

SOURCE 112
SPACE 108
SPACES 108
SRC 112
STATE 112
SUPPLIER 75
SWAP 91
SYSTEM 102

T

TASK 99
TERMIO 89
THEN 57
THROW 77
THRU 115
TIB 87
TOGGLE 98
TRIAD 115
TYPE 108

U

U. 109
U< 93
UO 99
UDM/MOD 101
UM* 101
UM/MOD 101
UNLOCK 49
UNLOOP 57
UNTIL 57

UPDATE 49
USER 62

V

VARIABLE 62
VERSION 75
VMA-IO 86
VOC-LINK 125

W

WARM 86
WARNING 77
WHERE 77
WHILE 58
WITHIN 98
WORD 112
WORDLIST 62
WORDS 70
WRITE-FILE 80

X

XOR 93
XOS 103
XOS5 102
XOS7 103

Z

ZEN 103

Forth Word Index

This index contains *all* references to a word. Use the glossary index to find the glossary description of each word.

- !
- ! 11, 93
 - !CSP 40, 116
- "
- " 8, 106, 110, 119
 - "CASE-SENSITIVE" 39
 - "cccc" 106
 - "newforth" SAVE-SYSTEM 14
- #
- # 80, 81
 - #> 80, 81, 82
 - #BUFF 46
 - #S 80, 81
- \$
- \$! 120
 - \$!-BD 120
 - \$+! 120
 - \$, 120
 - \$ / 120
 - \$^ 121
 - \$@ 57, 112, 120
 - \$\ 121
 - \$C+ 121
- &
- & 39, 110
- ,
- , 39, 66
 - , words 37
 - , ; 63
 - , ;CODE' 63
- (
- (..... 113
 - (+LOOP) 56, 58
 - (;) 58, 60
 - (;CODE) 63
 - (>IN) 88
 - (?DO) 58
 - (ABORT") 77
 - (ACCEPT) 86, 88
 - (BACK) 58, 59
 - (BLK) 115
 - (BUFFER) 48, 49
 - (CREATE) 19, 39, 63
 - (D.) 106
 - (D.R) 106
 - (DO) 54, 59
 - (FIND) 19, 70, 73
 - (FORWARD 59, 60
 - (LINE) 114
 - (NUMBER) 82
- *
- * 3, 104
 - */ 100
 - */MOD 100
- +
- + 3, 11, 104
 - +! 93
 - +BUF 50
 - +LOOP 52, 56, 57, 58
 - +ORIGIN 15, 16, 84
- ,
- , 61, 66
-
- 91, 104
 - > 40, 46
 - legacy- 14
 - traditional- 13, 14
 - TRAILING 121
 - x 22
- .
- 3, 107
 - . " 13, 106
 - . (..... 106
 - .R 107
 - .S 17, 117
 - .SIGNON 85, 98
- /
- / 103, 104
 - /MOD 104
- :
- : 4, 11, 40, 63
 - :F 39
- ;
- ; 4, 40, 60
 - ;CODE 63

<

<..... 92
 <#..... 80, 81, 82
 <>..... 92

==

=..... 91, 92

>

>..... 92
 >BODY..... 19, 67
 >CFA..... 19, 70
 >DFA..... 19, 71
 >FFA..... 19, 71
 >IN..... 1, 13, 14, 87, 88, 91
 >LFA..... 19, 71
 >NFA..... 19, 71
 >NUMBER..... 81
 >PHA..... 71
 >R..... 54, 55, 117
 >SFA..... 19, 71
 >VFA..... 72
 >WID..... 11, 72

?

?..... 107
 ??..... 28
 ?32..... 8
 ?AR1..... 9
 ?AR2..... 9
 ?ARa..... 9
 ?ARb..... 9
 ?BLANK..... 109, 110
 ?COMP..... 40, 116
 ?CSP..... 40, 116
 ?DELIM..... 116
 ?DISK-ERROR..... 39, 46
 ?DO..... 53, 56, 58
 ?DUP..... 90
 ?ENVIRONMENT..... 46
 ?ERROR..... 37, 75, 77
 ?ERRUR..... 75
 ?EXEC..... 116
 ?EXEC..... 40
 ?LOAD..... 40
 ?LOADING..... 116
 ?PAIRS..... 40, 117
 ?STACK..... 38, 117

[

[..... 4, 113
 [']..... 70
 [CHAR]..... 112
 [COMPILE]..... 52
 [DEFINED]..... 22

]

]..... 4, 113

^

^X LOAD..... 22

-

-..... 99
 _FIRST..... 20, 50
 _LIMIT..... 20, 50
 _pad..... 15
 _PREV..... 49, 50, 51

@

@..... 11, 94

\

\..... 113

~

~MATCH..... 73

0

0..... 110, 122
 0<..... 91
 0=..... 91
 0>IN..... 13
 OBRANCH..... 55, 58, 59

1

1+..... 86, 122
 1-..... 122

2

2!..... 94
 2*..... 122
 2,..... 67
 2/..... 123
 2@..... 94
 2DROP..... 88, 89
 2DUP..... 89
 2OVER..... 89
 2SWAP..... 89

A

ABORT 12, 75, 84, 86
 ABORT" 12, 75, 77
 ABS 104
 ACCEPT 21, 86, 88
 AGAIN 53, 54, 59
 ALIGN 94
 ALIGNED 94
 ALLOCATE 41
 ALLOT 39, 61, 67, 68
 ALSO 123
 AND 91, 92
 AR1 9
 ARG[] 22, 23
 ARGS 101
 ARSHIFT 105
 ASSEMBLER 13, 63, 123, 124
 AUTOLOAD 23

B

B 9
 B/BUF 20, 47
 B| 27, 28
 BACK) 59
 BAD 29
 BASE 37, 81, 82, 107, 109
 BEGIN 53, 54, 56, 57, 58
 BIC 94
 BIS 95
 BL 95, 121
 BLANK 95
 BLK 13, 115
 BLOCK 47, 48, 50, 113
 BLOCK-EXIT 47
 BLOCK-FILE 37, 39, 40, 46, 47
 BLOCK-HANDLE 47
 BLOCK-INIT 47, 50
 BLOCK-READ 16, 48
 BLOCK-SEEK 50
 BLOCK-WRITE 16, 48
 BM 15, 95
 BODY> 67
 BRANCH 40, 53, 54, 56, 59
 BYE 22, 84, 101, 102

C

C! 95
 C, 67
 C/L 114
 C@ 95
 CASE-INSENSITIVE 37
 CASE-SENSITIVE 17, 37
 CATCH 12, 24, 76, 77
 CELL+ 13, 95
 CELLS 96
 CHAR 109
 CHAR+ 96
 CHARS 96
 CLOSE-FILE 78
 CLS 118
 CMOVE 96, 97
 colon 62

CO 54, 111
 COLD 84
 COMPARE-AREA 96
 CONFIG 9
 CONSTANT 3, 11, 20, 60, 63
 CONTEXT 61, 62, 124
 CORA 96
 CORE 74
 COUNT 112, 120, 121
 CPU 74, 98
 CR 16, 21, 107, 108
 CRACK 9
 CREATE 19, 60, 61, 63, 67, 71
 CREATE-FILE 78
 CSP 116, 117
 CURRENT 11, 62, 63, 70, 123, 124, 125

D

D+ 73
 D 107
 D.R 107
 DABS 73
 DATA 20, 61
 DECIMAL 81
 DEFER 39
 DEFINITIONS 124
 DELETE-FILE 78
 DENOTATIONS 125
 DEPTH 118
 DEV-MEM 84
 DEVELOP 38, 39
 DH 9
 DI| 28
 DIGIT 82
 DISK-ERROR 48
 DLITERAL 51
 DNEGATE 74
 DO 40, 53, 54, 55, 56, 59
 DO-DEBUG 8
 DO-SECURITY 40
 DOES> 19, 20, 60, 61, 63, 67, 71
 DP 11, 16, 18, 67, 68
 DPL 52, 83
 DROP 76, 90
 DSP! 118
 DSP@ 118
 DUMP 9, 17
 DUP 90

E

ECL 83
 EDITOR 38
 ELSE 54, 55, 57, 59, 90
 EM 20, 97
 EMIT 16, 21, 108
 EMPTY-BUFFERS 48
 END-CODE 27
 ENVIRONMENT 74, 124
 ENVIRONMENT? 74, 124
 ERASE 97
 ERROR 75, 76, 77, 110
 ERRSCR 76

O

OFFSET 50
 OK 85
 ONLY 39, 64, 85, 110, 123, 125
 OPEN-FILE 79
 OPTIONS 85
 OR 93
 OS-IMPORT 10
 OUT 108, 109
 OVER 90
 OW, 28

P

P! 98
 P@ 98
 PAD 16, 22, 69, 81
 PARSE 110
 PERIPHERALS 85
 POSTPONE 51, 69
 PP 1, 13, 87, 88, 91
 PP@@ 110
 Prefix_" 64
 Prefix_& 64
 Prefix_+ 65
 Prefix_- 65
 Prefix_~ 66
 Prefix_0 64, 65
 Prefix_7 65
 Prefix_B 65
 Prefix_TICK 65
 PREFIX 12, 64, 69, 83
 PRESENT 69
 PREVIOUS 125
 PUT-FILE 15, 79

Q

QUIT 12, 46, 84, 86

R

R# 114
 R> 54, 55, 117, 118
 R@ 119
 RO 21, 118
 RDROP 119
 READ-FILE 79
 RECURSE 56
 REFILL 13, 14, 41
 REFILL-TIB 88
 REGRESS 41
 REMAINDER 88
 REPEAT 54, 56, 58, 59
 REPOSITION-FILE 80
 REQUIRE 14
 REQUIRE REQUIRED PRESENT? 14
 REQUIRED 14
 RESIZE 41
 RESTORE 111
 RESTORE-INPUT 111
 ROT 90
 RSHIFT 106
 RSP! 119

RSP@ 119
 RST 27
 RUBOUT 21, 87
 RW-BUFFER 80, 103

S

S" 122
 S: 41
 S>D 74
 SO 21, 119
 SAVE 111
 SAVE-INPUT 111
 SAVE-SYSTEM 14, 15, 18, 24
 SCR 114, 115
 SDLITERAL 52
 SEE 9, 10
 SET-SRC 88, 111, 112
 SET-TERM 88
 SHELL 102
 SHOW-ALL, 28
 SHOW-OPCODES 28
 SHOW: 25, 28
 SIGN 81, 82
 SKIP 57
 SM/REM 100
 SOURCE 86, 112
 SPACE 108
 SPACES 108
 SRC 88, 111, 112
 STATE 63, 64, 83, 85, 110, 112
 SUPPLIER 75
 SWAP 91
 SYSTEM 22, 41, 102

T

TASK 99
 TASK-SIZE 16
 TERMIO 88, 89
 TEST 4
 THEN 40, 54, 55, 57
 THROW 12, 37, 76, 77
 THRU 41, 115
 TIB 87, 88
 TICKS 9
 TOGGLE 98
 TRIAD 115
 TUCK 8
 TURNKEY 22, 24
 TYPE 16, 21, 106, 108

U

U 109
 U< 93
 UO 15, 99
 UDM/MOD 101
 UM* 101
 UM/MOD 101
 UNLOCK 49
 UNLOOP 57
 UNTIL 54, 57, 58, 59
 UPDATE 49, 50, 51

USER..... 20, 62, 63

V

VARIABLE..... 11, 20, 62, 63
 VERSION..... 75
 VMA-IO..... 86
 VOC-LINK..... 125
 VOCABULARY..... 14, 61

W

WANT..... 8, 9, 14, 21, 22, 23, 37, 45
 WANT -legacy-..... 14
 WANT -scripting-..... 40
 WANT -traditional-..... 13, 14
 WANT -tricky-control-..... 40
 WANT REFILL..... 13
 WANTED..... 8, 9, 14, 21, 22, 23, 24, 37, 40
 WARM..... 86
 WARNING..... 47, 76, 77

WHERE..... 76, 77
 WHILE..... 56, 58, 59
 WITHIN..... 98
 WORD FIND..... 14
 WORDLIST..... 62
 words..... 37
 WORD..... 14, 66, 70, 112
 WORDS..... 3, 70
 WRITE-FILE..... 80

X

X|..... 27, 28
 XOR..... 93
 XOS..... 103
 XOS5..... 102, 103
 XOS7..... 103

Z

ZEN..... 103

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.

!

! 98

—

-traditional- 14

A

aligned 94
 allocating 10
 ambiguous condition 13

B

B/BUF 49
 blocks 8, 11, 46, 47

C

case sensitive 1, 37
 cell 11, 45, 93
 ciforth specific behaviour 13
 code definitions 20
 code field 20
 code field address 63
 code field address 19, 20, 70, 99
 code word 17
 COLD 85
 colon definition 11, 20
 compilation mode 4
 compilation word list 124
 computation stack 10
 crash 13
 current input source .. 49, 82, 109, 110, 111, 112, 114,
 115

D

data 67
 data field 19, 60, 61
 data field address 19, 20, 60, 61, 62, 63, 67, 71
 data stack 10, 89, 117, 118
 dea 20, 62, 66, 67, 70, 72, 99
 DEA 10, 18, 70
 defining word 3, 11, 39, 60
 denotation .. 12, 20, 39, 64, 66, 68, 69, 70, 110, 119, 125
 denotations 69
 DEV-MEM 85
 DFA 60
 dictionary 10
 dictionary entry 10, 18, 124, 125
 dictionary entry address 10, 18, 19, 20, 45, 66, 67
 dictionary pointer 10, 67, 68
 double 11, 45, 73, 80, 93, 99

E

execution token 10, 19, 66, 70, 99

F

family of instructions 27
 field address 19
 flag 45, 91
 flag field address 19, 71
 floored division 103
 Forth flag 45, 91

G

gpio 38

H

high level 11, 17, 20, 99

I

immediate 63
 immediate bit 19
 in line 51
 index line 21
 index lines 23
 indexline 8
 inner interpreter 11, 17, 20

L

LAB 91
 library 8, 21
 Library Addressable by Block 11, 17
 library file 37, 47
 link field address 19, 62, 71
 load 11, 17, 113
 locked 41
 logical and 92
 logical not 92
 logical or 93
 logical xor 93

M

mnemonic message 37

N

name field address..... 19, 20, 71
 name token..... 19
 namespace..... 11, 41, 85
 nesting..... 11
 next..... 19
 nil pointer..... 46, 52, 61, 69, 83
 number..... 12
 number base..... 80

O

old fashioned string..... 68, 119, 120
 outer interpreter..... 109

P

past header..... 19
 past header address..... 19, 71
 preferences..... 22
 prefix..... 69

R

return stack..... 11, 117, 118
 revectoring..... 21, 67, 108, 109

S

screen..... 11, 113, 114
 search order..... 11, 63, 70, 124, 125
 search-order..... 110
 signal an error..... 46, 75
 smart..... 69
 smudge..... 19
 source field address..... 71
 stack..... 10
 stack pointer..... 10, 117
 string constant..... 45, 73, 119
 string variable..... 119
 stringconstant..... 64
 symmetric division..... 100, 103, 104, 105

T

turnkey application..... 24
 turnkey system..... 15

U

user area..... 16
 user variable..... 81

V

vectoring..... 21
 VLFA..... 61, 62

W

WID..... 11, 61, 62, 70, 72, 123, 125
 word..... 3
 word list..... 11, 19, 61, 123, 124, 125
 word list associated with..... 61
 word list identifier..... 11, 61, 62, 123
 wordset..... 46

Short Contents

1	Overview	1
2	Gentle introduction	3
3	Rationale & legalese	5
4	Manual	7
5	Assembler	25
6	Optimiser	31
7	Errors	37
8	Documentation summary	43
9	Glossary	45
	Glossary Index	127
	Forth Word Index	133
	Concept Index	139

Table of Contents

1	Overview	1
2	Gentle introduction	3
3	Rationale & legalese	5
3.1	Legalese	5
3.2	Rationale	5
3.3	Source	5
3.4	The Generic System this Forth is based on.....	6
4	Manual	7
4.1	Getting started	7
4.1.1	Hello world!.....	7
4.1.2	The library.....	8
4.1.3	The library and different hardware.....	9
4.1.4	Development.....	9
4.1.5	Finding things out.....	10
4.2	Concepts	10
4.3	Portability.....	12
4.3.1	REFILL	14
4.3.2	Compatibility with lina64 4.0.x.....	14
4.4	Configuring.....	14
4.5	Saving a new system.....	15
4.6	Memory organization	15
4.6.1	Boot-up Parameters.....	16
4.6.2	Installation Dependent Code.....	16
4.6.3	Machine Code Definitions	17
4.6.4	High-level Standard Definitions	17
4.6.5	User definitions.....	17
4.6.6	System Tools	17
4.6.7	RAM Workspace.....	18
4.7	Specific layouts	18
4.7.1	The layout of a dictionary entry.....	18
4.7.2	Details of memory layout	20
4.7.3	Terminal I/O and vectoring.....	21
4.8	Libraries and options	21
4.8.1	Options.....	22
4.8.2	Private libraries.....	24
4.8.3	Stand alone programs.....	24
5	Assembler	25
5.1	Introduction.....	25
5.2	Reliability	26
5.3	Principle of operation.....	26
5.4	The 8080 assembler	27
5.5	Opcode sheets	28
5.6	Assembler Errors	28

6	Optimiser	31
6.1	Introduction	31
6.1.1	Properties	31
6.1.2	Definitions	31
6.1.3	Notations	31
6.1.4	Optimisations	31
6.1.5	Data collecting	32
6.1.6	Purpose	32
6.2	Implementation	33
6.2.1	Stack effects	33
6.2.2	Optimisation classes	34
6.2.2.1	The no store bit.	34
6.2.2.2	The no fetch property.	34
6.2.2.3	The no stack effect property.	35
6.2.2.4	The no side effect property.	35
6.2.2.5	Associativity.	35
6.2.2.6	Short circuit evaluation.	35
6.2.3	Optimisation by recursive inlining	35
6.3	Concerning ARM	36
7	Errors	37
7.1	Error philosophy	37
7.2	Common problems	37
7.2.1	Error 11 or 12 caused by lower case.	37
7.2.2	Error 8 or only error numbers	37
7.2.3	Error 8 while editing a screen	38
7.2.4	Error -13 while trying I/O	38
7.3	Error explanations	38
8	Documentation summary	43
9	Glossary	45
9.1	BLOCKS	46
9.1.1	#BUFF	46
9.1.2	->	46
9.1.3	?DISK-ERROR	46
9.1.4	B/BUF	47
9.1.5	BLOCK-EXIT	47
9.1.6	BLOCK-FILE	47
9.1.7	BLOCK-HANDLE	47
9.1.8	BLOCK-INIT	47
9.1.9	BLOCK-READ	48
9.1.10	BLOCK-WRITE	48
9.1.11	BLOCK	48
9.1.12	DISK-ERROR	48
9.1.13	EMPTY-BUFFERS	48
9.1.14	LOCK	49
9.1.15	UNLOCK	49
9.1.16	UPDATE	49
9.1.17	(BUFFER)	49
9.1.18	+BUF	50
9.1.19	BLOCK-SEEK	50

9.1.20	FLUSH	50
9.1.21	OFFSET	50
9.1.22	_FIRST	50
9.1.23	_LIMIT	50
9.1.24	_PREV	51
9.2	COMPILING	51
9.2.1	DLITERAL	51
9.2.2	LITERAL	51
9.2.3	POSTPONE	51
9.2.4	[COMPILE]	52
9.2.5	LIT	52
9.2.6	SDLITERAL	52
9.3	CONTROL	52
9.3.1	+LOOP	52
9.3.2	?DO	53
9.3.3	AGAIN	53
9.3.4	BEGIN	53
9.3.5	CO	54
9.3.6	DO	54
9.3.7	ELSE	54
9.3.8	EXIT	55
9.3.9	IF	55
9.3.10	I	55
9.3.11	J	55
9.3.12	LEAVE	56
9.3.13	LOOP	56
9.3.14	RECURSE	56
9.3.15	REPEAT	56
9.3.16	SKIP	57
9.3.17	THEN	57
9.3.18	UNLOOP	57
9.3.19	UNTIL	57
9.3.20	WHILE	58
9.3.21	(+LOOP)	58
9.3.22	(;)	58
9.3.23	(?DO)	58
9.3.24	(BACK	58
9.3.25	(DO)	59
9.3.26	(FORWARD	59
9.3.27	0BRANCH	59
9.3.28	BACK)	59
9.3.29	BRANCH	59
9.3.30	FORWARD)	60
9.4	DEFINING	60
9.4.1	;	60
9.4.2	CONSTANT	60
9.4.3	CREATE	60
9.4.4	DATA	61
9.4.5	DOES>	61
9.4.6	MAX-USER	61
9.4.7	NAMESPACE	61
9.4.8	USER	62
9.4.9	VARIABLE	62
9.4.10	WORDLIST	62

9.4.11	colon	62
9.4.12	(;CODE)	63
9.4.13	(CREATE)	63
9.4.14	;CODE	63
9.4.15	HEADER	63
9.4.16	LINK	64
9.5	DENOTATIONS	64
9.5.1	Prefix_"	64
9.5.2	Prefix_&	64
9.5.3	Prefix_+	65
9.5.4	Prefix_-	65
9.5.5	Prefix_0	65
9.5.6	Prefix_7	65
9.5.7	Prefix_B	65
9.5.8	Prefix_TICK	65
9.5.9	Prefix_^	66
9.6	DICTIONARY	66
9.6.1	' (This addition because texinfo won't accept a single quote)	66
9.6.2	,	66
9.6.3	2,	67
9.6.4	>BODY	67
9.6.5	ALLOT	67
9.6.6	BODY>	67
9.6.7	C,	67
9.6.8	DP	68
9.6.9	FIND	68
9.6.10	FORGET	68
9.6.11	FOUND	68
9.6.12	HERE	68
9.6.13	ID.	69
9.6.14	IMMEDIATE	69
9.6.15	PAD	69
9.6.16	PREFIX	69
9.6.17	PRESENT	69
9.6.18	WORDS	70
9.6.19	[?]	70
9.6.20	(FIND)	70
9.6.21	>CFA	70
9.6.22	>DFA	71
9.6.23	>FFA	71
9.6.24	>LFA	71
9.6.25	>NFA	71
9.6.26	>PHA	71
9.6.27	>SFA	71
9.6.28	>VFA	72
9.6.29	>WID	72
9.6.30	FENCE	72
9.6.31	FOR-VOCS	72
9.6.32	FOR-WORDS	72
9.6.33	FORGET-VOC	72
9.6.34	HIDDEN	73
9.6.35	~MATCH	73
9.7	DOUBLE	73
9.7.1	D+	73

9.7.2	DABS	73
9.7.3	DNEGATE	74
9.7.4	S>D	74
9.8	ENVIRONMENTS	74
9.8.1	CORE	74
9.8.2	CPU	74
9.8.3	ENVIRONMENT?	74
9.8.4	NAME	74
9.8.5	SUPPLIER	75
9.8.6	VERSION	75
9.9	ERRORS	75
9.9.1	?ERROR	75
9.9.2	?ERRUR	75
9.9.3	ABORT"	75
9.9.4	CATCH	76
9.9.5	ERROR	76
9.9.6	ERRSCR	76
9.9.7	MESSAGE	76
9.9.8	THROW	77
9.9.9	WARNING	77
9.9.10	WHERE	77
9.9.11	(ABORT")	77
9.9.12	HANDLER	77
9.10	FILES	78
9.10.1	CLOSE-FILE	78
9.10.2	CREATE-FILE	78
9.10.3	DELETE-FILE	78
9.10.4	FILE-POSITION	78
9.10.5	GET-FILE	78
9.10.6	INCLUDED	79
9.10.7	INCLUDE	79
9.10.8	OPEN-FILE	79
9.10.9	PUT-FILE	79
9.10.10	READ-FILE	79
9.10.11	REPOSITION-FILE	80
9.10.12	WRITE-FILE	80
9.10.13	RW-BUFFER	80
9.11	FORMATTING	80
9.11.1	#>	80
9.11.2	#S	80
9.11.3	#	81
9.11.4	<#	81
9.11.5	>NUMBER	81
9.11.6	BASE	81
9.11.7	DECIMAL	81
9.11.8	HEX	82
9.11.9	HOLD	82
9.11.10	SIGN	82
9.11.11	(NUMBER)	82
9.11.12	DIGIT	82
9.11.13	DPL	83
9.11.14	ECL	83
9.11.15	FLD	83
9.11.16	HLD	83

9.11.17	NUMBER.....	83
9.12	INITIALISATIONS.....	84
9.12.1	+ORIGIN.....	84
9.12.2	ABORT.....	84
9.12.3	COLD.....	84
9.12.4	DEV-MEM.....	84
9.12.5	INIT.....	84
9.12.6	MMAP-IO.....	85
9.12.7	OK.....	85
9.12.8	OPTIONS.....	85
9.12.9	PERIPHERALS.....	85
9.12.10	QUIT.....	86
9.12.11	VMA-IO.....	86
9.12.12	WARM.....	86
9.13	INPUT.....	86
9.13.1	(ACCEPT).....	86
9.13.2	ACCEPT.....	86
9.13.3	KEY?.....	87
9.13.4	KEY.....	87
9.13.5	PP.....	87
9.13.6	RUBOUT.....	87
9.13.7	TIB.....	87
9.13.8	(>IN).....	88
9.13.9	REFILL-TIB.....	88
9.13.10	REMAINDER.....	88
9.13.11	SET-TERM.....	88
9.13.12	TERMIO.....	89
9.14	JUGGLING.....	89
9.14.1	2DROP.....	89
9.14.2	2DUP.....	89
9.14.3	2OVER.....	89
9.14.4	2SWAP.....	89
9.14.5	?DUP.....	90
9.14.6	DROP.....	90
9.14.7	DUP.....	90
9.14.8	NIP.....	90
9.14.9	OVER.....	90
9.14.10	ROT.....	90
9.14.11	SWAP.....	91
9.15	LIBRARY.....	91
9.15.1	L->IN.....	91
9.16	LOGIC.....	91
9.16.1	0<.....	91
9.16.2	0=.....	91
9.16.3	<>.....	92
9.16.4	<.....	92
9.16.5	=.....	92
9.16.6	>.....	92
9.16.7	AND.....	92
9.16.8	INVERT.....	92
9.16.9	OR.....	93
9.16.10	U<.....	93
9.16.11	XOR.....	93
9.17	MEMORY.....	93

9.17.1	!	93
9.17.2	+	93
9.17.3	2!	94
9.17.4	2@	94
9.17.5	@	94
9.17.6	ALIGNED	94
9.17.7	ALIGN	94
9.17.8	BIC	94
9.17.9	BIS	95
9.17.10	BLANK	95
9.17.11	BM	95
9.17.12	C!	95
9.17.13	C@	95
9.17.14	CELL+	95
9.17.15	CELLS	96
9.17.16	CHAR+	96
9.17.17	CHARS	96
9.17.18	CMOVE	96
9.17.19	CORA	96
9.17.20	EM	97
9.17.21	ERASE	97
9.17.22	FILL	97
9.17.23	L!	97
9.17.24	L@	97
9.17.25	MOVE	97
9.17.26	P!	98
9.17.27	P@	98
9.17.28	TOGGLE	98
9.17.29	WITHIN	98
9.18	MISC	98
9.18.1	.SIGNON	98
9.18.2	EXECUTE	99
9.18.3	NOOP	99
9.18.4	TASK	99
9.18.5	U0	99
9.18.6	-	99
9.19	MULTIPLYING	99
9.19.1	*/MOD	100
9.19.2	*/	100
9.19.3	FM/MOD	100
9.19.4	M*	100
9.19.5	SM/REM	100
9.19.6	UDM/MOD	101
9.19.7	UM*	101
9.19.8	UM/MOD	101
9.20	OPERATINGSYSTEM	101
9.20.1	ARGS	101
9.20.2	BYE	101
9.20.3	EXIT-CODE	102
9.20.4	FORK	102
9.20.5	MS	102
9.20.6	SHELL	102
9.20.7	SYSTEM	102
9.20.8	XOS5	102

9.20.9	XOS7	103
9.20.10	XOS	103
9.20.11	ZEN	103
9.21	OPERATOR	103
9.21.1	*	104
9.21.2	+	104
9.21.3	-	104
9.21.4	/MOD	104
9.21.5	/	104
9.21.6	ABS	104
9.21.7	ARSHIFT	105
9.21.8	LSHIFT	105
9.21.9	MAX	105
9.21.10	MIN	105
9.21.11	MOD	105
9.21.12	NEGATE	105
9.21.13	RSHIFT	106
9.22	OUTPUT	106
9.22.1	(D.)	106
9.22.2	(D.R)	106
9.22.3	."	106
9.22.4	.(.....	106
9.22.5	.R	107
9.22.6	107
9.22.7	?	107
9.22.8	CR	107
9.22.9	D.R	107
9.22.10	D	107
9.22.11	EMIT	108
9.22.12	ETYPE	108
9.22.13	OUT	108
9.22.14	SPACES	108
9.22.15	SPACE	108
9.22.16	TYPE	108
9.22.17	U	109
9.23	PARSING	109
9.23.1	?BLANK	109
9.23.2	CHAR	109
9.23.3	EVALUATE	109
9.23.4	INTERPRET	110
9.23.5	NAME	110
9.23.6	PARSE	110
9.23.7	PP@@	110
9.23.8	RESTORE-INPUT	111
9.23.9	RESTORE	111
9.23.10	SAVE-INPUT	111
9.23.11	SAVE	111
9.23.12	SET-SRC	111
9.23.13	SOURCE	112
9.23.14	SRC	112
9.23.15	STATE	112
9.23.16	WORD	112
9.23.17	[CHAR]	112
9.23.18	[.....	113

9.23.19	\	113
9.23.20]	113
9.23.21	(.....	113
9.24	SCREEN	113
9.24.1	(LINE)	114
9.24.2	C/L	114
9.24.3	INDEX	114
9.24.4	LIST	114
9.24.5	LOAD	114
9.24.6	R#	114
9.24.7	SCR	115
9.24.8	THRU	115
9.24.9	TRIAD	115
9.24.10	(BLK)	115
9.24.11	BLK	115
9.25	SECURITY	116
9.25.1	!CSP	116
9.25.2	?COMP	116
9.25.3	?CSP	116
9.25.4	?DELIM	116
9.25.5	?EXEC	116
9.25.6	?LOADING	116
9.25.7	?PAIRS	117
9.25.8	?STACK	117
9.25.9	CSP	117
9.26	STACKS	117
9.26.1	.S	117
9.26.2	>R	117
9.26.3	CLS	118
9.26.4	DEPTH	118
9.26.5	DSP!	118
9.26.6	DSP@	118
9.26.7	R0	118
9.26.8	R>	118
9.26.9	R@	119
9.26.10	RDROP	119
9.26.11	RSP!	119
9.26.12	RSP@	119
9.26.13	S0	119
9.27	STRING	119
9.27.1	\$!-BD	120
9.27.2	\$!	120
9.27.3	\$+!	120
9.27.4	\$,	120
9.27.5	\$/	120
9.27.6	\$@	120
9.27.7	\$C+	121
9.27.8	\$\	121
9.27.9	\$^	121
9.27.10	-TRAILING	121
9.27.11	BL	121
9.27.12	COUNT	121
9.27.13	S"	122
9.28	SUPERFLUOUS	122

9.28.1	0	122
9.28.2	1+	122
9.28.3	1-	122
9.28.4	2*	122
9.28.5	2/	123
9.28.6	Number_1	123
9.28.7	Number_2	123
9.29	WORDLISTS	123
9.29.1	ALSO	123
9.29.2	ASSEMBLER	123
9.29.3	CONTEXT	124
9.29.4	CURRENT	124
9.29.5	DEFINITIONS	124
9.29.6	ENVIRONMENT	124
9.29.7	FORTH	125
9.29.8	LATEST	125
9.29.9	ONLY	125
9.29.10	PREVIOUS	125
9.29.11	VOC-LINK	125
Glossary Index		127
Forth Word Index		133
Concept Index		139