ciforth manual

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

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

- version 1.0
- 16-bits
- standalone
- contains security words
- kernel contains the full ISO CORE set

Albert van der Horst
Dutch Forth Workshop

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 9, Section Portability.

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

ciforth consists of three files:

- A booting floppy or hard disk : the program
- '`ciforth.ps`' '`ciforth.html`' : the documentation
- '`forth.lab`' : source library for auxiliary programs

These files are generated together by a generic system from the file '`ci68.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 7).

These are the features:

All ciforth's are *case sensitive* . This is version 1.0 . It is a booting standalone Forth. It is a standalone Forth in rom. A number has a precision of 16 bits. It has compiler security, sacrificing some bizarre ISO compatibity. It doesn't use `>IN` exactly in the way prescribed by ISO. It contains the full ISO CORE in the kernel, more than is needed to make it self contained. 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:
'`rom`'
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:

```
[After booting]
_SBC_([After loading])

6809 ciforth beta $RCSfile: ci86.gnr,v $ $Revision: 1.51 $

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.

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 may get ' `constant ? ciforth ERROR # 12 : NOT RECOGNIZED` '. That is because you didn't type in what I said. rom is case sensitive. If you want to change that consult the section "Common problems". (see Chapter 6 [Errors], page 35).

This 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' .

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

```
: TEST 1 2 + 7 * ;      12 .
12 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 HCC FIG Holland. This Forth is called ciforth and is made available by the D.F.W.. All publications of the D.F.W. are available under GPL, the GNU public license. The file '`COPYING`' containing the legal expression of these lines must accompagny it.

Because Forth is "programming by extending the language" the GPL could be construed to mean that systems based on ciforth always are legally obliged to make the source available. But we consider this "fair use in the Forth sense" as expressed by the following statement.

In addition to the GPL HCC FIG Holland 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 fair use of the 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.

3.2 Rationale

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

3.3 Source

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

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

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

The latest version of rom 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 acknowledgements 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 dozen's 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 and 32 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 '`rom`' 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
```

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 rom. As soon as you leave rom, this command is gone.

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 6 [Errors], page 35, 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 rom you never have to worry about 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 rom'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 rom does.

rom uses the *blocks* as a library by addition of the word `REQUIRED` and a convention. Starting with '`rom -r`' or most any option you have this facility available. If you are already in rom, 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" REQUIRED
OK
DO-DEBUG

S[ ] OK 1

S[ 1 ] OK
```

You can turn DO-DEBUG off with NO-DEBUG .

If you try to INCLUDE a program, you may get errors like 'TUCK? ciforth ERROR # 12 : NOT RECOGNIZED'. See Chapter 6 [Errors], page 35, Section Explanations. Apparently, rom doesn't know about a forth word named TUCK , but after '"TUCK" REQUIRED' maybe it does. You may try again.

The convention about the way the library file must be organized for REQUIRED to find something is simple. It is divided into blocks of 16 lines. The first line is the header of the block. If the word we are looking for is mentioned in the header, that block is compiled.

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

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

4.1.3 Development.

If you want to try things out, or write a program – as opposed to just running a ready made program – you best do '5 LOAD' immediately. You can configure this screen 5 to suit your particular needs.
You will have available:

1. REQUIRED and REQUIRE . 'REQUIRE xxx' is equivalent to '"xxx" REQUIRED' , but it is more convenient.
2. DH. H. B. DUMP FARDUMP

   For showing numbers in hex and parts of memory.
3. CRACK

   To analyse words, showing the source code of compiled words.

   .

4.1.4 Finding things out.

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

```
REQUIRE TUCK
LOCATE TUCK
```

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

```
REQUIRE TUCK
CRACK 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 Configuring

For configuring your rom, without enlarging the dictionary, you may use the following sequence

```
 S" myforth.lab" BLOCK-FILE $! \ Or any config
1 LOAD
REQUIRE SAVE-SYSTEM
: DOIT
    'CONTAINS 'FORTH FORGET-VOC
    'CONTAINS >NFA  DP !
    "newforth" SAVE-SYSTEM BYE ;
DOIT
```

Here '`DOIT`' trims the dictionary just before the saving your system into a new file. You can use a similar sequence for expanding the system without building in the `SAVE-SYSTEM` command as well.

`FAR-DP` allows have a disposable part of the dictionary. This may be occasionally useful, but make sure to `FORGET` always the disposed off words.

4.3 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 8.6 [DICTIONARY], page 59). 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 vocabulary, with the exception of some fine points regarding *denotation*'s. Apart from the name a word contains data and executable code, (interpreted or not) and linking information (see Section 8.4.8 [VOCABULARY], page 56).

The concept word list is part of the ISO standard, but we will use *vocabulary* . A vocabulary is much more convenient, being a word list with a name, created by `VOCABULARY` . ISO merely knows *word list identifier* 's, a kind of handle, abbreviated as *WID* . A new word list is created by the use of `VOCABULARY` , and by executing the vocabulary 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 vocabulary. If needed you can always derive the WID from the DEA of a vocabulary.

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 '`W`'. It then jumps to the address pointed to by the address pointed to by '`W`'. '`W`' points to the code field of a definition which contains the address of the code which executes for that definition. 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* could 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* . 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 8.4 [DEFINING], page 54). 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 an Pentium processor this is 32 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 16, 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 are most accessible on the stack and if stored in memory, it is highest.

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

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

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

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)
```

4.4 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. However 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 allows. 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
2. It is not possible to catch the following words : ABORT" ABORT QUIT .
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 doesn't use `>IN` exactly in the way prescribed by ISO. The `>IN` that is there is a fake, that can only be read, not changed.
5. Vocabularies are wordlists with a name. However they push the wordlist to the search order, instead of replacing the topmost one. In this sense `FORTH` and `ASSEMBLER` words are not strictly conforming.
6. This is not strictly non-conforming, but worth mentioning here. Despite the fact that rom contains only one state-smart word besides `SLITERAL` (that word is `."`) which is about politically correct, as a consequence all denotations and some other words like `[CHAR]` and `[']`

   (but not `CHAR` and `'`) turn out to be state smart as a design consequence. 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, so don't complain about non-portability. The libraries tend to rely on ciforth-specific and rom-specific – but hopefully documented – behaviour anyway.

Here we will explain how you must read the glossary of rom, 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 "rom", 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. 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 rom 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 bit clean. This means that it runs on 16 and 32 bits systems.

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 of the 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.

Of course, whether you have a hosted system or a booted system like this one, it is clear that some system-dependant information goes into accomplishing this.

In the following we use the naming convention of ISO about cells. A cell is the fundamental unit of storage for the Forth engine. Here it is 16 bits (2 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 $40 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 rom to a new file using `PUT-FILE` . The difficult part is to get the system specific header right. Add to the header the area from `BM` to `HERE` .

This has all been sorted out for you. Just use `SAVE-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
- Stacks and terminal input buffer.

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

The disk block buffers are allocated in the dictionary, because otherwise they would not be accessible to the BIOS

The program as residing on disk may contain startup code, but that is of no concern for the usage.

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

Instead 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 could support 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 extends from '`0 +ORIGIN`' and has initial value for all of the user area. This is the image for the *user area* .

So in ciforth the bootup parameters are more or less the data field of the `+ORIGIN` word. Executing '`0 +ORIGIN`' leaves a pointer in this area.

4.6.2 Installation Dependent Code

`KEY EMIT KEY? CR` and `R/W` 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.

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.

Again code words are perfectly normal dictionary entries.

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 will result in deviations from the standard, so you probably want to leave this area alone. Again these words are perfectly normal dictionary entries. 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 formating. Some of these are fixed by the ISO standard too. These definitions are placed above the standard definitions to facilitate modification. That is you may `FORGET` part of the high-level and re-compile altered definitions from disc. 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).

Again these words are perfectly normal dictionary entries. A number of entries that could easily be made loadable are integrated in the assembler source of this ciforth version. You can forget them, and load your own version from files or blocks.

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. An assembler is not part of ciforth as delivered, but it is available after '`REQUIRE ASSEMBLERi86`'. It automatically loads the proper 16-bits version. You can load a more elaborate assembler. See Chapter 5 [Assembler], page 21, Section Overview. They are among the first candidates to be integrated into your system by `SAVE-SYSTEM` .

See Chapter 4 [Manual], page 9, Section Getting Started.

386 and a 8086 Forth assemblers are available in '`forth.lab`'. They are loaded in accordance with the system that is run.

It is essential that you regard rom 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. Later you can rewrite them as code words.

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, the computation and return stacks, the user area, and the terminal 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.

There is no longer a reason to put up with a 16-bit system less than 64K.

The boundary between this area and the previous one is pretty sharp, it is where `DP` points. The other areas are not clearly separated at all. But even this boundary constantly changes as you add and forget definitions.

4.7 Specific layouts

4.7.1 The layout of a dictionary entry

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

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

They go from lowest in memory to highest:

1. The code field. This is one cell. A pointer to such a field is called a *code field address* . It contains the address of the code to be executed for this word.
2. The data field, of the DEA, not in the ISO sense. This is one cell. A pointer to such a field is called a *data field address* . It contains a pointer to an area owned by this definition.
3. The flag field. This is one cell. A pointer to such a field is called a *flag field address* . For the meaning of the bits of the flag field sea below.
4. The link field. This is one cell. A pointer to such a field is called a *link field address* . It contains the dictionary entry address of the last word that was defined in the same *word list* before this one.
5. The name field. This is one cell. This contains a pointer to a string. A pointer to such a field is called a *name field address* . The name itself is stored outside of the dictionary header in a regular string, i.e. a one cell count followed by as many characters. Unfortunately, *name field address* is sometimes used, where a *dictionary entry address* would be more correct, especially in older documentation. This came about because the name was lowest in memory. That they happen to be the same is no reason to confuse two completely different concepts. In this Forth the code field address and the dictionary address are the same, but not accidentally so.
6. 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.

Note that the entries are not only in alphabetic order, they are in order of essentiality. They are accessed by `>CFA >DFA >FFA >LFA >NFA >SFA` .

Note *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.

The most important flag bits currently defined are:

- The INVISIBLE bit = 1 when *smudge* d, and 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 vocabulary. This indicates that it should not be executed.
- The DENOTATION bit =1 for a word from the DENOTATION word list. 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 letter word, but not necessarily.

`(CREATE)` takes care to generate this data structure; it is called by all defining words.

For all *colon definition* 's the code field 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`'.

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 machine code. The code field of this word points to it too.
- For a word defined by `VARIABLE` , `USER` , or `CONSTANT` it has a width of one cell, and contains data.
- For all *colon definition* 's the data field address contains a pointer to an area with a variable length. It contains the compiled high level code, a sequence of *code field address* address es.
- 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.

A *dictionary entry address* can be turned into any of these fields by words that are in the vocabulary '`DICTIONARY`'. See Section 8.6 [DICTIONARY], page 59,, for those field words. They customarily start with `>`.

A dictionary falls apart into the

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 in fact the data area owned by `FIRST` . 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 CELL+ CELL+` bytes.

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

The user area is configured at $40 cells, most of it unused. There is no word to add user variables. The user area is at the upper bound of RAM memory. So it ends at `EM` .

The terminal input buffer and the return stack share an area configured at a size of $0100 bytes. The lower half is intended for the terminal 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 8.11.1 [+ORIGIN], page 75, for the bootup area. But take care. You probably need to study the source for how and when they take effect.

If you need multi-tasking you have to allocate a separate user area 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 terminal input buffer.

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 standalone system the need for this is high, because there is no redirection.) For this reason character output `CR` , `EMIT` and `TYPE` all go through a common word that can be changed. . 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 EXPECT are subject to correction until <RET> has been pressed.

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 REQUIRED takes a string and loads the first screen where that name occurs in the index line. For convenience also REQUIRE is there that looks ahead in the input stream. These words are not in the kernel but are in screen 17, that corresponds to the ‘-r’ option.

Screen 0 and screen 1 to 31 are reserved.

4.8.1 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 REQUIRED .

In order to customize the forth library, you have to make a copy , preferably to a lib subdirectory. Then you can start up using a ‘-l’ option, or make a customized rom. See Chapter 4 [Manual], page 9, Subsection Configuring.

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 rom in your programs can remain the same.

4.8.2 Turnkey applications.

Turnkey application are made using the word TURNKEY . They take a word, that is to be done, and a string with the file name. Mostly it is much easier to just use the ‘-c’ option. See Chapter 4 [Manual], page 9, Getting Started Subsection Hello World! A turnkey application should decide what to do with the library file that is default opened in COLD . Make sure to CATCH errors from BLOCK-EXIT and ignore them.

5 Assembler

5.1 Introduction

Via '`http://home.hccnet.nl/a.w.m.van.der.horst/ciforth.html`' you can find here a couple of assemblers, to complement the generic ciforth system. The assemblers are not part of the rom package, and must be fetched separately. They are based on the postit/fixup principle. The assembler that is present in the blocks, is based on the same principle, but is less sophisticated, especially regards error detection. If you use that one, you can still benefit from this section by the background information it gives.

On this stand alone version of ciforth, you only have the assembler in blocks, but they are not documented separately. So you have to keep up with this description. These files are not part of this distribution, maybe you but they are available at my site, see chapter 3.)

prototype : still present in the forth.lab ("blocks"), usable and cleaned up
'`ass.frt`' : the 80-line 8086 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`'
'`asi586.frt`' : 80386 assembler, requires '`asgen.frt`'
'`ps.frt`' : generate opcode sheets
'`p0.asi586.ps`' : first byte opcode for asi586 assembler
'`p0F.asi586.ps`' : two byte opcode for same that start with 0F.
'`test.mak`' : makefile, i.e. with targets for opcode sheets.

De '`asi586.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, type it in and ask for completion. 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.

Now for the bad news. The assembler in the Library Addressable by Blocks (block file) hasn't any of those feature. It is intended for incidental use, to speed up a crucial word. Worse yet, the opcodes are not always the same as used here, and the comma-ers are even mostly different. You have to resort to that old game, reading the source.

5.2 Reliability

I skimped on write up. I didn't skimp on testing, at least not for '`asi586.frt`'. It is 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, must come out 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. Bugs where revealed, yes... in the other tool.

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 Terabytes. It is also not practical for the Alpha with 32K register combinations per instruction.

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 that was not too bad and I am returning to that. (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.
The simplest example is

```
INC [BX]
```

Are we to increment the byte or the word at BX? (Intel's solution : '`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| DO [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.
- COMMA phase: `IX,` or `X,` add addresses and immediate data.

Doesn't that 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.

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 . (You bet that the postit-fixup principle pays off for the Pentium, but not for 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 a 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 asi586.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 asi586.ps MASK=FFFF PREFIX=0F`' The opcode sheets '`p0.asi586.ps`' and '`p0F.asi586.ps`' are already made 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|`' and, as of this writing, GNU's objdump gives the same disassembly for both '`IMUL`'s despite the difference between a 32*32 > 64 and a 32*32 > 32 operation: '`IMUL|AD, X| R| BX|`' or '`IMUL, AX'| R| BX|`'

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.

5.6 Details about the 80386 instructions

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

1. A word ending in `,` reserves place in the dictionary. It stand for one assembler instruction. The start of the instruction is kept and there is a bitfield (the tally) for all bits that belong to the instruction, if only mentally. These bits are put as comment in front of the instruction and they are considered filled in. They also imply the instruction length.
2. A word ending in `|` is a fixup, it `OR` s in some bits in an already assembled instruction. Again there is a mask in front of fixups and in using the fixup these bits are considered to be filled in. A fixup cannot touch data before the start of the latest instruction.
3. Families can be constructed from instructions or fixups with the same tally bit field, provided they differ by a fixed increment. If data or addresses following differ this is unwise.
4. The part before a possible `|` in an instruction – but excluding an optional trailing I – is the opcode. Opcodes define indeed a same action.
5. The part after `|` in an instruction may be considered a built in fixup where irregularity forbids to use a real fixup. A X stands for xell or natural data width. This is 16 bit for a 16 bit assembler and 32 bit for a 32 bit assembler. These can be overruled with `AS:,` applying to `DX|` and `MEM|` and with `OS:,` applying to tdata required where there is an I suffix.
6. Width fixups determine the data width : `X|`
   (xell or natural data width 16/32) or `B|` (8 bit) unless implied.
7. Instruction ending in *I* have an immediate data field after all fixups. This can be either `X,` (xell or natural data width) or `B, W, L,` (8 16 32 bit). If there are width fixups they should correspond with the data.
8. Instructions ending in '`|SEG`' builtin fixup (segments) require `SEG,` (which is always 16 bits). If `X,` cannot be used caused by width overrules, the programmer should carefully insert `W,` or `L,` whatever appropriate.
9. With r/m you can have offsets (for `DB|` and `DX|`) that must be assembled using `B,` or `X,` but mind the previous point.
10. Instruction with r/m can have a register instead of memory indicated by the normal fixups `AX|` etc.
11. If instructions with r/m have another register, that one is indicated by a prime such as `AX'|` . Or if an instruction can handle two general registers, the one that cannot be replaced by a memory reference gets a prime.
12. Unless `T| F|` (to/from) are present, a primed register is the modifiable one, else `T| F|` refer to the primed register. The primed register is the one that cannot be replaced by a memory reference.
13. At the start of an instruction the mask of the previous instruction plus fixup should add up non-overlappingly to a full field. Offsets and immediate data should have been comma-ed in in that order.
14. A fixup or instruction is mightier than an other one if its mask contains all the bits of that other one. The second fixup or instruction shall then not be used.
15. Instructions ending in ' `:,` ' are prefixes and are considered in their own right. They have no fixups.
16. The Scaled Index Byte is handled in the following way: The fixup `SIB|` closes the previous instruction (i.e. fill up its bit field), but possible immediate data and offsets are kept. Then `SIB,` starts a new instruction.
17. The `SET,` instruction unfortunately requires a duplicate of the O| etc. fixups of the `J,` and `J|X,` instructions.

18. Some single byte instructions require `X'|` and `B'|` instead of `X|` and `B|` that are used for the ubiquitous instructions with r/m.

Hand disassembling can be done as follows.

1. Find the mightiest instruction that agrees with the data at the program counter. Tally the bits. The instructions length follows from the instruction. As does the presence of address offsets and immediate data. The dictionary may be organized such that the mightiest instruction is always found first.
2. Find the mightiest fixup that agrees with untallied bits.
3. If not all bits have been tallied go to 2
4. Disassemble the address offsets and immediate data, in accordance with the instruction. Length is determined from fixups and prefix bytes. The result must agree with the instruction in the first place.

5.7 Using 16 bits code in the 32 bit assembler

In general X refers to Xell. So in 16 bit mode or with a 16 bit prefix AX is to mean the Intel `AX` instead of what is normal: `EAX` . It is thus possible to insert a patch of 16 bit code in 32 bit code all with the 32 bit assembler. This can be necessary in system programming. Just use '`MOV, AX'| R| BX|`' and in 16 bit mode it refers to 16 bit registers.

If an address overwrite suffix applies, the indexing fixups ending in a prime must be used, e.g. '`[BX+SI]'`' instead of `[AX]` for code running in a default 32 bit environment. (Otherwise use the 8086 assembler) But during system programming only the programmer knows what is going on, so some error messages are suppressed.

While using 16 bits code, whenever you get error messages and you are sure you know better than the assembler, put `!TALLY` before the word that gives the error messages, and they will be suppressed.

5.8 This assembler is not yet integrated in the generic Forth

In the generic Forth automatically a 32 bit assembler is loaded if the Forth itself is 32 bits and a 16 bit assembler for the 16 bit forths. Adding the extra complexity to run the 16 bit assembler on a 32 bit system would be the drip that overflows the bucket. This is no restriction for what code can be generated. **The built in assembler has no error checking and may have bugs the very extensively tested** '`asi586.frt`' **has not.** ;

5.9 A rant about redundancy

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

And yes '`ES: OS: MOV, X| T| DI'| SIB|, DX| [BP +8* AX] FFFFF800 X,`' though being correct, and in a logical order, looks still bad, because it **is** bad in the sense that the Pentium design got overboard in complication. (This example is from the built-in assembler, the one in '`asi586.frt`' redefines `[BP` c.s. to get rid of the `SIB|,` instruction.)
First remark: lets assume this is 32 bit code,(because otherwise there would not be a `SIB,` sure?)
There are 3 sizes involved :

- The size of the data transported this is always the '`X`' as in `X|` . Then the first `X|` changes its meaning to 16 bit, because of the `OS:` prefix.
- The `X` in `DX|` and in `X,` must agree and are 32 bits because you are in a 32 bits segment and this cannot be overridden.
- The offset (in '`+AX]`') is counted in 64 bits, a strange array for fetching the '`DI`' but anyway.

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

5.10 Reference opcodes

Table one contains all the opcodes used in '`asi586.frt`' in alphabetic order, with `|` sorted before any letter. All opcodes on the first position are the same as Intel opcodes.

You can use it in two ways.

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

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

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

Table 1. Opcode cross reference.

AAA,

AAD,

AAM,

AAS,

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

AS:,

BOUND,

BSF,

BSR,
BT, *BTI*,
BTC, *BTCI*,
BTR, *BTRI*,
BTS, *BTSI*,
CALL, *CALLFAR*, *CALLFAROI*, *CALLO*,
CBW,
CLC,
CLD,
CLI,
CLTS,
CMC,
CMP, *CMPI*, *CMPI*|*A*,
CMPS, *CMPSI*,
CPUID,
CS:,
CWD,
DAA,
DAS,
DEC, *DEC*|*X*,
DIV|*AD*,
DS:,
ENTER,
ES:,
FS:,
GS:,
HLT,
IDIV|*AD*,
IMUL, *IMUL*|*AD*, *IMULI*, *IMULSI*,
INC, *INC*|*X*,
INS,
INT, *INT3*, *INTO*,
IN|*D*, *IN*|*P*,
IRET,
J, *J*|*X*, *(Intel Jcc)*
JCXZ,
JMP, *JMPFAR*, *JMPFAROI*, *JMPO*, *JMPS*,
LAHF,
LAR,

LDS,
LEA,
LEAVE,
LES,
LFS,
LGDT,
LGS,
LIDT,
LLDT,
LMSW,
LOCK,
LODS,
LOOP, *LOOPNZ*, *LOOPZ*,
LSL,
LSS,
LTR,
MOV, *MOV*|*CD*, *MOV*|*FA*, *MOV*|*SG*, *MOV*|*TA*,
MOVI, *MOVI*|*BR*, *MOVI*|*XR*,
MOVS,
MOVSX|*B*, *MOVSX*|*W*,
MOVZX|*B*, *MOVZX*|*W*,
MUL|*AD*,
NEG,
NOT,
OR, *ORI*, *ORI*|*A*,
OS:,
OUTS,
OUT|*D*, *OUT*|*P*,
POP, *POP*|*ALL*, *POP*|*DS*, *POP*|*ES*, *POP*|*FS*, *POP*|*GS*, *POP*|*SS*, *POP*|*X*,
POPF,
PUSH, *PUSH*|*ALL*, *PUSH*|*CS*, *PUSH*|*DS*, *PUSH*|*ES*, *PUSH*|*FS*, *PUSH*|*GS*, *PUSH*|*SS*,
PUSH|*X*,
PUSHF,
PUSHI|*B*, *PUSHI*|*X*,
RCL,
RCR,
REPNZ,
REPZ,

RET+, *RET*, *RETFAR+*, *RETFAR*,
ROL,
ROR,
SAHF,
SAR,
SBB, *SBBI*, *SBBI|A*, *SBBSI*,
SCAS,
SET, *(Intel SETcc)*
SGDT,
SHL,
SHLD|C, *SHLDI*,
SHR,
SHRD|C, *SHRDI*,
SIDT,
SLDT,
SMSW,
SS:,
STC,
STD,
STI,
STOS,
STR,
SUB, *SUBI*, *SUBI|A*, *SUBSI*,
TEST, *TESTI*, *TESTI|A*,
VERR,
VERW,
WAIT,
XCHG,
XCHG|AX,
XLAT,
XOR, *XORI*, *XORI|A*,
~SIB,

Table 2 Suffixes

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

;

5.11 The dreaded SIB byte

If you ask for the operands of a memory instruction (one of the simple one is LGDT,) instead of all the sib (*scaled index byte*) possibilities you see. '`LGDT, DB| ~SIB| 14 SIB,, 18, B,`' This loads the general description table from an address described by a sib-byte of 14.

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

For the curious:

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

Instead of the comma-er we use a `~SIB,` instruction. This specifies in fact a one byte opcode with three fields examplified by '`[AX +2* SI`'] (and again you might say '`+2* SI] [AX`' with the same meaning.)

5.12 A last caveat

There is no way to communicate to the assembler whether the current instructions are supposed to be executed in 16 or 32 bit mode. This means that if you use the address overwrite prefix `AS:,` and/or primed fixups `[BX]'` and/or run your code in 16 bit mode, you must be very careful. As long as you stay away from the above, you can be sure that valid instructions are correctly assembled and executed as you specified and invalid instructions are rejected.

5.13 An incomplete and irregular guide to the instruction mnemonics.

The following is an attempted overview of the suffixes and fixup's used. It may be of some help for using the assembler because it gives some idea of some of the names. It is not checked in a long time and was inaccurate and incomplete in the first place. You may also find names that are only used in the block files and not explained in table 1. So beware!

Note that some of the instruction are Pentium and as yet not present in the '`asi586.frt`' (which should still be called '`asi386.frt`').

Never use an instruction that end in a ' (such as `[BP+IS]'`

except in case of address size overwrites **and** you know what you are doing.

Some instructions
SET : Byte Set on Condition
BT : Bit Test
BTR: Bit Test and Reset
BTS: Bit Test and Set
BTC: Bit Test and Complement
CPUID: CPU Identification
CLTS:
L : Load Full Pointer
LAR : Load Access Rights Byte
LLDT: Load Local Descriptor Table Register
LGDT: Load General Descriptor Table Register
LIDT: Load Interrupt Descriptor Table Register
LTR: Load Task Register
LMSW: Load Machine Status Word
MOV : Move
RSM:

RDTSC: Read from Time Stamp Counter
RDMSR: Read from Model Specific Register
SHLD: Double Precision Shift Left
SHRD: Double Precision Shift Right
SLDT: Store Local Descriptor Table Register
SMSW: Store Machine Status Word
VERR: Verify a Segment for Reading or Writing
WRMSR: Write to Model Specific Register

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

Items in Fixups.
Y| : Yes, Use the condition straight
N| : No, Use the condition inverted
O| : Overflow
C| : Carry
Z| : Zero
CZ| : C || Z (unsigned <=)
S| : Sign (<0)
P| : Parity (even)
L| : S != O (signed <)
LE| : L || Z (signed <=)
<AH| : As a second register is a source, different in size from the destination. T| : To (primed or special register)
F| : From (primed or special register)
V| : Variable number

OB : Obligatory byte
OW : Obligatory word (=16bits)
;

5.14 6809 specific information

The reference manual is the "MC6809-MC6809E Microprocessor Programming Manual" from Motorola. The first, mnemonic part of an instruction is the same as the Motorola opcode in the reference plus a comma `,`.

Only for direct page addressing a modifier '`|D`' is inserted before this comma. Note that the X in STX is not an inherent address. LD and ST as they are on top of the page in the reference are never used as mnemonics as such, neither is it here.

The register names are always as found in table 4.

The reference doesn't use consequent names, in particular not for CCR and DPR .

Table 5 6809 register names

A : Data register 8 bit.

B : Data register 8 bit.

D : Data register 16 bit.

X : Index register
Y : Index register
U : Index register
S : Index register
CCR : Condition code.
DPR : Direct page.

Index register names are used as such in indexed addressing expressions.

The addressing mode is a fixup such as found in table 5.

Table 5 6809 addressing modes

E| : Extended, requires '`E,`'
[] : Indexed, requires an index expression
: Direct adressing is build into the mnemonic.
A| : Inherent A
B| : Inherent B
D| : Inherent D
#| : 1 byte immediate, requires '`#,`'
##| : 2 byte immediate, requires '`##,`'

An index expression is one of the expressions found in the column "Assembler Form" in table F-2 of appendix F of the reference, OF THE TABLE? FIXME!> followed by the name of the index register, which names can be found at the bottom of the table. Where there is an '`n`' in this table, it is replaced by '`#`' for a 1-byte and by '`##`' for a 2-byte constant. They require a '`#,`' respectively '`##,`'. For example '`[n,R]`' is disambiguated to '`[#,R]`' in the following assembly instruction
'`CLR [] [#,R] X 5 #,`'
The '`n,R`' indication for a register indirect with 5-bit offset is just ommitted, or if you want, implied in the 5 bits offset. The 5 bits offset itself must be followed by '`|#,`' . For example to clear the memory 5 bytes from where X points, use
'`CLR X 5 |#,`'

5.14.1 Special Expressions

The '`EXG`' and '`TFR`' instructions require two registers. The names are such as in table 5. A difference with the reference that '`D`' is used instead of '`A:B`' . The first (source) register gets the suffix '`==`' , the second (destination) register gets the prefix '`=>`' . As per '`TFR, D== ==>X`'

The '`PULx`' and '`PSHx`' is to be followed by a set of registers in the following fashion
'`(& reg1& reg2& ... )S,`'
Register names '`reg1`' '`reg2`' are as in table 5. Specify both halves of '`D`' separately.

5.15 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 6 [Errors], page 35, 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 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 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 the error up in the section "Error explanations". More specific problems are addressed in the section "Common Problems".

6.1 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.

6.2 Common problems

6.3 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 rom, but maybe by other version of ciforth or even a differently configured rom. 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 21).

.

Here are the error explanations.

- '`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`'
- '`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 inappropiate.
- ‘ciforth ERROR # 7 : FULL STACK’
  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 : DISC ERROR !’
  An access to the Library Accessible by Block (screen aka block file) 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. If you forget, you get this message too.
- ‘ciforth ERROR # 9 : UNRESOLVED FORWARD REFERENCE’
  Not used.
- ‘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. If the DENOTATION vocabulary is on top of the search order, you get this message if you type 2SWAP . Note that hex digits must be typed in uppercase, even if "CASE-SENSITIVE" is in effect. This error 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 # 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 temporily, 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, require NEW-IF .
- ‘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)
- ‘ciforth ERROR # 20 : DEFINITION NOT FINISHED’

  This error is reported by ?CSP . It detects stack unbalance between : and ; . This means there is an error in the compiled code. It happens also if you try to use data that is put on the stack before : during compilation. 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 : DECLARE VOCABULARY’
- ‘ciforth ERROR # 25 : LIST EXPECTS DECIMAL’

  This message is used by a redefined LIST , to prevent getting the wrong screen.

See Section 8.27.2 [ASSEMBLER], page 112,, for errors generated by the assembler. These have numbers that are all higher than the general errors.

7 Documentation summary

This is copied from the FIG documentation 1978. It is probably out of date now.

The following manuals are in print:

Caltech FORTH Manual, an advanced manual with internal details of Forth. Has Some implementation peculiarities. Approx. $6.50 from the Caltech Book Store, Pasadena, CA.

Kitt Peak Forth Primer, $20.00 postpaid from the Forth Interest Group, P. O. Box 1105, San Carlos, CA 94070.

microFORTH Primer, $15.00 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

8 Glossary

Wherever it says single precision number or *cell* 16 bits is meant. Wherever it says *double* or "double precision number" a 32 bits number is meant.

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

The symbols include:

'`addr`' memory address

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

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

'`d`' 32 bit signed double integer: most significant portion with sign on top of stack

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

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

'`faraddr`'

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

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

'`n`' 16 bit signed integer number; it is also used for a 16-bit entity where it is irrelevant what number it represents

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

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

'`u`' 16-bit unsigned integer

'`ud`' 32-bit unsigned double integer: most significant portion on top of stack

The capital letters on the right show definition characteristics:

'`B`' The word is available only after loading from background storage

'`C`' May only be used within a colon definition. A digit indicates number of memory addresses used, if other than one.

'`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

'`L0`' Level Zero definition of FORTH-78

'`L1`' Level One definition of FORTH-78

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

'`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. Words marked with '`ISO,FIG,LO`' will behave identically over all but the whole spectra of Forth's.

Unless otherwise noted, all references to numbers are for 16-bit signed integers. For 32-bit signed numbers, the most significant part (with the sign) is on top.

All arithmetic is implicitly 16-bit signed integer math, with error and under-flow indication unspecified.

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

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

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

8.1 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 . Most blocks are used for the '`SCREEN`' facility, where each block contains source code.

8.1.1 RESTORE-INPUT

Name: `RESTORE-INPUT`

Stackeffect: n1 n2 n3 3—f

Attributes: ISO

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: '`RESTORE`' '`SAVE-INPUT`'

8.1.2 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: '`RESTORE-INPUT`' '`SAVE-INPUT`'

8.1.3 SAVE-INPUT

Name: `SAVE-INPUT`
Stackeffect: — n1 n2 n3 3
Attributes: ISO
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: '`RESTORE`' '`RESTORE-INPUT`'

8.1.4 SAVE

Name: `SAVE`
Stackeffect: —
Attributes:
Description: Save the content of `SRC` on the return stack prior to changing the *current input source* . This must be balanced by a RESTORE in the same definition.
See also: '`RESTORE`' '`SAVE-INPUT`'

8.2 COMPILING

The wordset '`COMPILING`' contains words that compile See Section 8.6.13 [IMMEDIATE], page 62, words and numbers. You need special precautions because these words would execute during compilation. Numbers are compiled *in line* , behind a word that fetches them.

8.2.1 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`'

8.2.2 LITERAL

Name: `LITERAL`
Stackeffect: n — n (executing) n — (compiling)
Attributes: ISO,I,C2,L0
Description: If compiling, then compile the stack value '`n`' as a 16 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`'

8.2.3 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]`'

8.2.4 [COMPILE]

Name: `[COMPILE]`

No stackeffect

Attributes: ISO,I,C

Description: Used in a colon-definition in form:

```
:  xxx    [COMPILE]   FORTH   ;
```

`[COMPILE]` will force the compilation of an immediate definitions, that would otherwise execute during compilation. The above example will select the `FORTH` vocabulary when '`xxx`' executes, rather than at compile time.

See also: '`POSTPONE`'

8.2.5 LIT

Name: `LIT`

Stackeffect: — n

Attributes: FIG,C2,L0

Description: Within a colon-definition, `LIT` is compiled followed by a 16 bit literal number given during compilation. Later execution of `LIT` causes the contents of this next dictionary cell to be pushed to the stack.

See also: '`LITERAL`'

8.2.6 SDLITERAL

Name: SDLITERAL
Stackeffect: d — s/d (executing) d — (compiling)
Attributes: I
Description: If compiling, compile a stack double number into a literal or double literal, depending on whether DPL contains a *nil pointer* or points into the input. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.
See also: 'SLITERAL' 'DLITERAL'

8.3 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.

8.3.1 +LOOP

Name: +LOOP
Stackeffect: n1 — (run) addr n2 — (compile)
Attributes: ISO,I,C2,L0
Description: Used in a colon-definition in the form:

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

At run-time, +LOOP selectively controls branching back to the corresponding DO based on 'n1' , the loop index and the loop limit. The signed increment 'n1' is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit ('n1>0'), or until the new index is equal to or less than the limit ('n1<0'). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO . 'n2' is used for compile time error checking.

8.3.2 ?DO

Name: ?DO
Stackeffect: n1 n2 — (execute) addr n — (compile)
Attributes: ISO,I,C2,L0
Description: Occurs in a colon-definition in form:

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

It behaves like DO , with the exception that if n1 and n2 are equal the loop body is not executed.
See also: ‘DO’ ‘I’ ‘LOOP’ ‘+LOOP’ ‘LEAVE’

8.3.3 AGAIN

Name: AGAIN
Stackeffect: addr n — (compiling)
Attributes: ISO,FIG,I,C2,L0
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’

8.3.4 BEGIN

Name: BEGIN
Stackeffect: — addr n (compiling)
Attributes: ISO,FIG,I,L0
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’

8.3.5 CO

Name: CO

No stackeffect

Attributes:

Description: Return suspend 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'

8.3.6 DO

Name: DO

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

Attributes: ISO,FIG,I,C2,L0

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'

8.3.7 ELSE

Name: ELSE

Stackeffect: addr1 n1 — addr2 n2 (compiling)

Attributes: ISO,FIG,I,C2,L0

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 emplaces BRANCH reserving a branch offset, leaves the 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'

8.3.8 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: '`(;)`'

8.3.9 IF

Name: `IF`

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

Attributes: ISO,FIG,I,C2,L0

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`'

8.3.10 I

Name: `I`

Stackeffect: — n

Attributes: ISO,FIG,C,L0

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

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

8.3.11 J

Name: J

Stackeffect: — n

Attributes: ISO,FIG,C,L0

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'

8.3.12 LEAVE

Name: LEAVE

No stackeffect

Attributes: ISO

Description: Termination 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 .

8.3.13 LOOP

Name: LOOP

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

Attributes: ISO,I,C2,L0

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'

8.3.14 RECURSE

Name: RECURSE

Stackeffect: (varies)

Attributes: ISO

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

See also: ':'

8.3.15 REPEAT

Name: `REPEAT`
Stackeffect: addr1 n1 addr2 n2— (compiling)
Attributes: ISO,FIG,I,C2
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`'

8.3.16 SKIP

Name: `SKIP`
No stackeffect
Attributes: C2,L0
Description: Skip over an area in memory, where the length is given in the next cell. This length doesn't include the length cell, so it is compatible with `$@` . Internal, used for nested compilation and compiling strings.
See also: '`BRANCH`'

8.3.17 THEN

Name: `THEN`
Stackeffect: addr n — (compile)
Attributes: ISO,FIG,I,CO,L0
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`'

8.3.18 UNLOOP

Name: UNLOOP
Stackeffect: — n
Attributes: ISO,I,C,L0
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 .
See also: 'DO' 'LOOP' '+LOOP' '(BACK' '(FORWARD' 'EXIT'

8.3.19 UNTIL

Name: UNTIL
Stackeffect: f — (run-time) addr n — (compile)
Attributes: ISO,FIG,I,C2,L0
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'

8.3.20 WHILE

Name: WHILE
Stackeffect: f — (run-time) addr1 nl — addr2 n1 addr1 n2(compile-time)
Attributes: ISO,FIG,I,C2
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'

8.3.21 (+LOOP)

Name: (+LOOP)
Stackeffect: n —
Attributes: C2

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

See also: '`+LOOP`'

8.3.22 (;)

Name: `(;)`

No stackeffect

Attributes:

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

See also: '`EXIT`'

8.3.23 (?DO)

Name: `(?DO)`

No stackeffect

Attributes: C

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

See also: '`?DO`'

8.3.24 (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`'

8.3.25 (DO)

Name: `(DO)`

No stackeffect

Attributes: C

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

See also: '`DO`'

8.3.26 (FORWARD

Name: `(FORWARD`

Stackeffect: — addr

Attributes:

Description: Start a forward branch by leaving the address that must be backpatched with an offset into '`addr`'. Usage is '`POSTPONE BRANCH (FORWARD .. FORWARD)` '

See also: '`IF`'

8.3.27 (LOOP)

Name: `(LOOP)`

No stackeffect

Attributes: C2

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

See also: '`LOOP`'

8.3.28 0BRANCH

Name: `0BRANCH`

Stackeffect: f —

Attributes: FIG,C2

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

See also: '`BRANCH`'

8.3.29 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`'

8.3.30 BRANCH

Name: `BRANCH`

No stackeffect

Attributes: FIG,C2,L0

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

See also: 'OBRANCH'

8.3.31 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'

8.4 DEFINING

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

8.4.1 :

Name: :

No stackeffect

Attributes: ISO,FIG,E,L0

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 ';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 (I) are executed rather than being compiled.

8.4.2 ;

Name: ;

No stackeffect

Attributes: ISO,FIG,I,C,L0

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

See also: ':'

8.4.3 CONSTANT

Name: `CONSTANT`

Stackeffect: n —

Attributes: ISO,FIG,L0

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

See also: '`VARIABLE`'

8.4.4 CREATE

Name: `CREATE`

No stackeffect

Attributes:

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 and the execution action can be changed. ciforth is byte aligned, so no extra measures are needed.

See also: '`DOES>`' '`;CODE`' '`ALLOT`' '`,`' '`C,`'

8.4.5 DOES>

Name: `DOES>`

No stackeffect

Attributes: ISO,FIG,L0

Description: A word which is normally use to specify the run-time action within a high-level defining word. `DOES>` modifies the behaviour of the latest word as to execute the sequence of compiled word addresses following `DOES>` . Used in combination with `CREATE` . 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, multidimensional arrays, and compiler generation.

8.4.6 USER

Name: `USER`

Stackeffect: n —

Attributes: ISO,L0

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 fixed 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 ciforth the '`UP`' is fixed.

See also: '`VARIABLE`' '`+ORIGIN`'

8.4.7 VARIABLE

Name: `VARIABLE`

No stackeffect

Attributes: ISO,E,LU

Description: A defining word used in the form: '`VARIABLE cccc`' When `VARIABLE` is executed, it creates the definition '`cccc`' with its data field pointing to a data location. When '`cccc`' is later executed, the content of its data field (containing '`n`') is left on the stack, so that a fetch or store may access this location.

See also: '`USER`' '`CONSTANT`'

8.4.8 VOCABULARY

Name: `VOCABULARY`

No stackeffect

Attributes: FIG,E,L

Description: A defining word used in the form: `VOCABULARY` '`cccc`' to create a vocabulary 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 searched first by `INTERPRET` .

A vocabulary 's data content field contains at first the dovoc pointer (like for any `DOES>` word) , then follows its body. The body contains the vocabulary link field address (*VLFA*). The VLFA points to the VLFA of the next vocabulary 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 vocabulary or a *nil pointer* if empty. Executing the vocabulary means pushing its WID on top of the `CONTEXT` order. In ciforth when there can be at most 8 word list 's in the search order, the oldest one gets lost. The vocabularies generated are `IMMEDIATE` words.

See also: '`VOC-LINK`' '`DEFINITIONS`'

8.4.9 (;CODE)

Name: `(;CODE)`

No stackeffect

Attributes: C

Description: The run-time proceedure, compiled by `;CODE` , that rewrites the code field of the most recently defined word to point to the following machine code sequence. It is used after `CREATE` instead of `DOES>` if the code following is assembler code instead of high level code.

See also: ‘ `(CREATE)`’ ‘`;CODE`’

8.4.10 (CREATE)

Name: `(CREATE)`

Stackeffect: sc —

Attributes:

Description: This is the basis for all defining words. It lays down the string ‘`sc`’ in the dictionary, then creates a dictionary entries with that string as the namefield. It is linked into the `CURRENT` word list. The code field and data field both point to the area owned by this header, i.e. immediately following the completed header as appropriate for a low level (assembler) definition. The flag field is empty , so not `HIDDEN` .

See also: ‘`CREATE`’

8.4.11 ;CODE

Name: `;CODE`

No stackeffect

Attributes: B,ISO,FIG,I,C,L0

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* order. Start assembling to machine code the following mnemonics.

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

See also: ‘`(;CODE)`’ ‘`LOAD`’ ‘`:`’

8.5 DENOTATIONS

The wordset ‘`DENOTATIONS`’ contains prefixes (mostly one letter words) that introduce a *denotation* , i.e. a generalisation of `NUMBER` . Any word starting with the prefix is considered found in the dictionary and the prefix word executed. These words parse input and leave a constant (number, char or string) on the stack, or compile such constant. They reside in a special vocabulary, called `DENOTATION` . To make a distinction with the same words in other wordlists, the names of denotations are prepended with “Prefix_” in the documentation. Actual names in the dictionary do not contain the prefix. Apart from `Prefix_0` , the vocabulary 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 cannot be postponed.

8.5.1 Prefix_"

Name: `Prefix_"`
Stackeffect: — sc
Attributes: CI
Description: Leave a `"` delimited string. A `"` can be embedded in a string by doubling it.

8.5.2 Prefix_&

Name: `Prefix_&`
Stackeffect: — c
Attributes: CI
Description: Leave '`c`' the non blank char that follows. Skip another blank character.

8.5.3 Prefix_+

Name: `Prefix_+`
Stackeffect: — s/d
Attributes: CI
Description: Implements `NUMBER` for numbers that start with `+` .

8.5.4 Prefix_-

Name: `Prefix_-`
Stackeffect: — s/d
Attributes: CI
Description: Implements `NUMBER` for numbers that start with `-` .

8.5.5 Prefix_0

Name: `Prefix_0`
Stackeffect: — s/d
Attributes: CI
Description: Implements `NUMBER` for numbers that start with *`0`* . Similar words are present for all decimal and hex digits. ISO compatibility would demand that denotators for all capitals are present, but one can always use a leading zero.

8.5.6 Prefix_^

Name: `Prefix_^`
Stackeffect: — b
Attributes: CI
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.

8.5.7 Prefix__TICK

Name: `Prefix__TICK`
Stackeffect: — addr
Attributes: ISO,FIG,I,L0
Description: Used in the form:

```
'nnnn
```

In interpret mode 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. In ciforth it can be used in compilation mode too, it then compiles the address as a literal. It is recommended that one never compiles or postpones it. (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. Note that if you separate by a space, the ISO-conforming version of `'` is found.
See also: '`CONTEXT`' '`[']`' '`PRESENT`' '`>CFA`' '`>DFA`' '`>FFA`' '`>LFA`' '`>NFA`' '`>SFA`'

8.6 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.

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

Name: `'`
Stackeffect: — addr
Attributes: ISO,FIG,I,L0
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`' '`EXECUTE`'

8.6.2 ,

Name: `,`
Stackeffect: n —
Attributes: ISO,FIG,L0
Description: Store '`n`' into the next available dictionary memory cell, advancing the *dictionary pointer* .
See also: '`DP`' '`C,`'

8.6.3 >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>`'

8.6.4 ALLOT

Name: `ALLOT`
Stackeffect: n —
Attributes: ISO,FIG,L0
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: '`CELL+`'

8.6.5 BODY>

Name: `BODY>`
Stackeffect: addr — dea
Attributes:
Description: Convert the data field '`addr`' of a definition created with a `CREATE` / `DOES>` construct to its '`dea`'. 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`'

8.6.6 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`' '`,`'

8.6.7 DP

Name: DP
Stackeffect: — addr
Attributes: FIG,U,L
Description: A user variable, the *dictionary pointer* , which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT .

8.6.8 FIND

Name: FIND
Stackeffect: addr —xt 1/xt -1/addr 0
Attributes: ISO
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.
See also: 'CONTEXT' 'PRESENT' '(FIND)'

8.6.9 FORGET

Name: FORGET
No stackeffect
Attributes: ISO,FIG,E,L0
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'

8.6.10 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' '(FIND)' 'VOCABULARY'

8.6.11 HERE

Name: HERE
Stackeffect: — addr
Attributes: ISO,FIG,L0
Description: Leave the address 'addr' of the next available dictionary location.
See also: 'DP'

8.6.12 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`'

8.6.13 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 unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceding it with `POSTPONE` .

8.6.14 PAD

Name: `PAD`

Stackeffect: — addr

Attributes: ISO,FIG,L0

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

8.6.15 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 a *denotation* word, the name must match '`sc`' exactly.

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

8.6.16 WORDS

Name: `WORDS`

No stackeffect

Attributes: ISO

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

See also: '`CONTEXT`'

8.6.17 [']

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. In ciforth this word is just an alias for `'` , so it can be used in interpret mode too. It is recommended that for non-portable code the *denotation* `'` is used instead, and that it is never postponed. (Use a combination of `WORD` and `FIND` instead.).
See also: '`FOUND`' '`'`' '`EXECUTE` '

8.6.18 (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 vocabulary, 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: '`FIND`' '`PRESENT`' '`>WID`'

8.6.19 (MATCH)

Name: `(MATCH)`
Stackeffect: sc dea — sc dea ff
Attributes:
Description: Intended to cooperate with `FOR-WORDS` . Compares the *string constant* '`sc`' with the dea '`dea`'. Returns into '`ff`' a flag indicating that it is a match taking into account denotations.
See also: '`FIND`'

8.6.20 >CFA

Name: `>CFA`
Stackeffect: dea — addr
Attributes:

Description: Given a dictionary entry addres ‘dea’ return its *code field address* ‘addr’. By jumping indirectly via this address the definition ‘dea’ is executed. 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: ‘’’ ‘CFA>’ ‘>DFA’ ‘>FFA’ ‘>LFA’ ‘>NFA’ ‘>SFA’ ‘>PHA’

8.6.21 >DFA

Name: >DFA

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres 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: ‘’’ ‘>BODY’ ‘>CFA’ ‘>FFA’ ‘>LFA’ ‘>NFA’ ‘>SFA’ ‘>PHA’

8.6.22 >FFA

Name: >FFA

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres return its *flag field address* ‘addr’ .

See also: ‘’’ ‘>CFA’ ‘>DFA’ ‘>LFA’ ‘>NFA’

8.6.23 >LFA

Name: >LFA

Stackeffect: dea — addr

Attributes:

Description: Given a dictionary entry addres return its *link field address* ‘addr’. It contains the DEA of the previous word.

See also: ‘’’ ‘>CFA’ ‘>DFA’ ‘>FFA’ ‘>NFA’ ‘>PHA’ ‘>SFA’

8.6.24 >NFA

Name: >NFA

Stackeffect: dea — nfa

Attributes:

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

See also: ‘’’ ‘>CFA’ ‘>DFA’ ‘>FFA’ ‘>LFA’ ‘>SFA’

8.6.25 >PHA

Name: `>PHA`

Stackeffect: dea — addr

Attributes:

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

See also: ‘’’ ‘`>CFA`’ ‘`>BODY`’

8.6.26 >VFA

Name: `>VFA`

Stackeffect: dea — cfa

Attributes:

Description: Given the dictionary entry addres of a vocabulary return the address of the link to the next vocabulary.

See also: ‘`VOCABULARY`’ ‘`>CFA`’ ‘`>WID`’

8.6.27 >WID

Name: `>WID`

Stackeffect: dea — wid

Attributes:

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

See also: ‘`VOCABULARY`’ ‘`>CFA`’ ‘`(FIND)`’

8.6.28 CFA>

Name: `CFA>`

Stackeffect: cfa — dea

Attributes:

Description: Convert the code field address of a definition to its *dictionary entry address* ‘`dea`’.

See also: ‘’’ ‘`>CFA`’

8.6.29 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` .

8.6.30 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`'

8.6.31 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`'

8.6.32 FORGET-VOC

Name: `FORGET-VOC`

Stackeffect: addr wid – addr

Attributes:

Description: Remove all words whose DEA is greater (later defined) than '`addr`' from a wordlist given by '`wid`' . Leave '`addr`' (as `FORGET-VOC` is intended to be used with `FOR-VOCS`) . If any whole vocabulary is removed, the search order is reset to '`ONLY FORTH`'. The space freed is not recovered.

See also: '`FORGET`'

8.6.33 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`'

8.7 DOUBLE

The wordset ‘DOUBLE’ contains words that manipulate *double* ’s.

8.7.1 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’ ‘+’

8.7.2 DABS

Name: DABS
Stackeffect: d — ud
Attributes: ISO,FIG
Description: Leave the absolute value ‘ud’ of a double number ‘d’ .
See also: ‘DNEGATE’ ‘ABS’

8.7.3 DNEGATE

Name: DNEGATE
Stackeffect: d1 — d2
Attributes: ISO
Description: ‘d2’ is the negation of ‘d1’.
See also: ‘D+’

8.7.4 S>D

Name: S>D
Stackeffect: n — d
Attributes: ISO
Description: Sign extend a single number to form a double number.

8.8 ENVIRONMENTS

The wordset ‘ENVIRONMENTS’ contains all words of the ENVIRONMENT vocabulary 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.

8.8.1 CORE

Name: `CORE`
Stackeffect: — ff
Attributes: ISO
Description: An environment query whether the CORE wordset is present.
See also: '`ENVIRONMENT?`'

8.8.2 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?`'

8.8.3 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` vocabulary and '`i*x`' is what is left by the word if executed.
See also: '`VOCABULARY`'

8.8.4 NAME

Name: `NAME`
Stackeffect: — sc
Attributes: CI
Description: An environment query giving the name of this Forth as a *string constant*.
See also: '`ENVIRONMENT?`'

8.8.5 SUPPLIER

Name: `SUPPLIER`
Stackeffect: — sc
Attributes: CI
Description: An environment query giving the SUPPLIER of this Forth as a *string constant*.
See also: '`ENVIRONMENT?`'

8.8.6 VERSION

Name: `VERSION`
Stackeffect: — sc
Attributes: CI
Description: An environment query giving the version of this Forth as a *string constant*.
See also: '`ENVIRONMENT?`'

8.9 ERRORS

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

8.9.1 ?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 signalled by the kernel go through this word, allowing to catch e.g. errors in accessing the block file.
See also: '`ERROR`'

8.9.2 ABORT"

Name: `ABORT"`
Stackeffect: f —
Attributes: ISO,I
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`'

8.9.3 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`'

8.9.4 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. This word is executed by THROW before restarting the interpreter and can be revectored to give more elaborate diagnostics.

See also: '?ERROR' 'WARNING'

8.9.5 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'

8.9.6 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 open, and messages are fetched from it. If it is 0, messages will be presented by number only. because the error system will call itself recursively. This may lead to a crash.

See also: 'MESSAGE' 'ERROR' 'ERRSCR'

8.9.7 WHERE

Name: WHERE

Stackeffect: — addr

Attributes: U

Description: A user variable pair which contains the start of the source and the character position of the last error that was 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 '

8.9.8 (ABORT")

Name: `(ABORT")`
Stackeffect: f —
Attributes:
Description: The run time action of `ABORT"` .

8.9.9 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`'

8.10 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 8.7 [DOUBLE], page 67, wordset. See Section 8.17 [MULTIPLYING], page 90, wordset.

8.10.1 #>

Name: `#>`
Stackeffect: d — sc
Attributes: ISO,FIG,L0
Description: Terminates numeric output conversion by dropping '`d`', leaving the formatted string '`sc`' .
See also: '`<#`'

8.10.2 #S

Name: `#S`
Stackeffect: d1 — d2
Attributes: ISO,FIG,L0
Description: Generates ASCII text in the text output buffer, by the use of `#` , until a zero double number '`d2`' results. Used between `<#` and `#>` .

8.10.3

Name: #

Stackeffect: d1 — d2

Attributes: ISO,FIG,L0

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'

8.10.4 <#

Name: <#

No stackeffect

Attributes: ISO,FIG,L0

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'

8.10.5 >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'

8.10.6 BASE

Name: BASE

Stackeffect: — addr

Attributes: ISO,FIG,U,L0

Description: A user variable containing the current number base used for input and output conversion.

See also: 'DECIMAL' 'HEX' '<#'

8.10.7 DECIMAL

Name: `DECIMAL`
No stackeffect
Attributes: ISO,FIG,L0
Description: Set the numeric conversion `BASE` for decimal input-output.
See also: '`HEX`'

8.10.8 HEX

Name: `HEX`
No stackeffect
Attributes: ISO,FIG,L0
Description: Set the numeric conversion `BASE` for hexadecimal (base 16) input-output.
See also: '`DECIMAL`'

8.10.9 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`'

8.10.10 SIGN

Name: `SIGN`
Stackeffect: n —
Attributes: ISO,FIG
Description: Stores an ASCII minus-sign `-` just before a converted numeric output string in the text output buffer when '`n`' is negative. Must be used between `<#` and `#>` .
See also: '`HOLD`'

8.10.11 (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`'

8.10.12 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 only a don't care value and a false flag.

8.10.13 DPL

Name: `DPL`

Stackeffect: — addr

Attributes: FIG,U,L0

Description: A user variable containing the address of the decimal point on double integer input, or a *nil pointer*. It may also be used to hold the output column location of a decimal point, in user generated formating.

See also: '`<#`' '`FLD`' '`HLD`'

8.10.14 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.

8.10.15 HLD

Name: `HLD`

Stackeffect: — addr

Attributes: FIG,L0

Description: A user variable that holds the address of the latest character of text during numeric output conversion.

See also: '`<#`' '`DPL`' '`FLD`'

8.10.16 NUMBER

Name: `NUMBER`

Stackeffect: — s/d

Attributes:

Description:

This word is intended to be called from single character denotation words, hence the parse pointer is decremented, to include this first character. Convert characters from the current input

source into a number, and compile or execute this number. 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.

See also: ‘BASE’ ‘(NUMBER)’

8.11 INITIALISATIONS

The wordset ‘INITIALISATIONS’ contains words to initialise, reinitialise or configure Forth.

8.11.1 +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. It will also change the initialisation by COLD .

See also: ‘USER’

8.11.2 ABORT

Name: ABORT

No stackeffect

Attributes: ISO,FIG,L0

Description: Restart the system. In addition and before the actions of QUIT , clear also the data stack and reset the search order, reset the exception mechanism and set the numeric base to default. As required by ISO it 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’

8.11.3 COLD

Name: COLD

No stackeffect

Attributes: FIG

Description: Reinitialise the system. Clear all block buffers. Reinitialise all user variables to their boot up values values, i.a. the stacks and the *dictionary pointer* . 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.

See also: ‘WARM’ ‘BLOCK’ ‘LIST’

8.11.4 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. 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 *vocabulary* . The number base is decimal. .

See also: 'WARM' 'COLD' 'FORTH' 'BLOCK-INIT' 'BASE' 'SO'

8.11.5 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 .

See also: 'QUIT' 'COLD'

8.11.6 QUIT

Name: QUIT

No stackeffect

Attributes: ISO,FIG

Description: Restart the interpreter. Clear the return stack, stop compilation, and return control to the operators terminal, or to the redirected input stream. This means ACCEPT user input to TIB , and then INTERPRET with TIB as a SOURCE . No message is given.

See also: 'ABORT'

8.11.7 RTI

Name: RTI

Stackeffect: —

Attributes:

Description: Perform a 'return from interrupt'. A high level interrupt handler must end by calling this word.

See also: 'ASSEMBLER'

8.11.8 VOC>RAM

Name: `VOC>RAM`
Stackeffect: dea —
Attributes:
Description: For the '`dea`' of a vocabulary copy its data to ram. Its *data field address* is already pointing to this place in RAM. This is executed during a cold start for code in ROM only.
See also: '`COLD`'

8.11.9 WARM

Name: `WARM`
No stackeffect
Attributes: FIG L0
Description: Perform a so called "warm" start. In addition and before the actions of `ABORT` , discard blocks; they may not be written back to mass storage. Display the sign on message.
See also: '`ABORT`'

8.12 INPUT

The wordset '`INPUT`' contains words to get input from the terminal and such.
See Section 8.1 [BLOCKS], page 42, for access of blocks.

8.12.1 (ACCEPT)

Name: `(ACCEPT)`
Stackeffect: — sc
Attributes:
Description: Input a line until a ***`RET`*** from the input terminal, and leave it as a *string constant*. It doesn't contain a trailing ***`LF`***. But possibly a trailing ***`RET`*** ($0D). This is lighter on the system and sometimes easier to use than `ACCEPT`

Text is probably present in the buffer address contained in `TIB @` , but not necessarily at the beginning. Editing is done locally observing `RUBOUT` .
See also: '`ACCEPT`' '`EVALUATE`'

8.12.2 >IN

Name: `>IN`
Stackeffect: — addr
Attributes: ISO
Description: 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. All parsing words use and move the value of `IN` . The variable `>IN` is not actually used and its content must be fetched immediately.
See also: '`(>IN)`' '`IN`'

8.12.3 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 backspace key on the standard PC keyboard layout is used to delete characters, without involvement of Forth, so changing RUBOUT has no effect. Other editing keys such as *F3* and cursor keys probably work as usual with a console application. 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)'

8.12.4 IN

Name: IN
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 IN .
See also: '>IN' 'WORD' '(WORD)' 'NUMBER' '(PARSE)' 'IN[]'

8.12.5 KEY?

Name: KEY?
Stackeffect: — ff
Attributes: ISO
Description: Perform a test of the terminal keyboard for a break request. Any key pressed is interpreted as such and the key is not consumed. A true flag indicates actuation.
See also: 'KEY' 'ACCEPT'

8.12.6 KEY

Name: KEY
Stackeffect: — c
Attributes: ISO,FIG,L0
Description: Leave the ASCII value of the next terminal key struck.

8.12.7 RUBOUT

Name: RUBOUT
Stackeffect: — c
Attributes:
Description: A user variable, leaving the key code that must delete the last character from the input buffer.
See also: 'USER'

8.12.8 TIB

Name: `TIB`

Stackeffect: — addr

Attributes: ISO,FIG,U

Description: A user variable containing the address of the terminal input buffer.

See also: '`QUIT`'

8.12.9 (>IN)

Name: `(>IN)`

Stackeffect: — addr

Attributes:

Description: If the standard word `>IN` is used, this variable contains the offset from the start 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 `IN` .

See also: '`>IN`' '`IN`'

8.12.10 REMAINDER

Name: `REMAINDER`

Stackeffect: — addr

Attributes:

Description: A pointer to a constant string that contains the balance of characters fetched into the input buffer, but not yet consumed. Used as in '`REMAINDER 2@`' .

See also: '`REFILL-TIB`'

8.13 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's. Design your word such that you need them as little as possible, because they are confusing.

8.13.1 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`'

8.13.2 2DUP

Name: `2DUP`
Stackeffect: d — d d
Attributes: ISO
Description: Duplicate the double number on the stack.
See also: '`OVER`'

8.13.3 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: '`2DUP`'

8.13.4 2SWAP

Name: `2SWAP`
Stackeffect: d1 d2 — d2 d1
Attributes: ISO
Description: Exchange the top doubles on the stack.
See also: '`ROT`'

8.13.5 ?DUP

Name: `?DUP`
Stackeffect: n1 — n1 (if zero) / n1 — n1 n1 (non-zero)
Attributes: ISO,FIG,L0
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`' '`_`'

8.13.6 DROP

Name: `DROP`
Stackeffect: n —
Attributes: ISO,FIG,L0
Description: Drop the number from the stack.
See also: '`DUP`'

8.13.7 DUP

Name: `DUP`
Stackeffect: n — n n
Attributes: ISO,FIG,L0
Description: Duplicate the value on the stack.
See also: '`OVER`'

8.13.8 OVER

Name: `OVER`
Stackeffect: nl n2 — nl n2 n1
Attributes: ISO,FIG,L0
Description: Copy the second stack value, placing it as the new top.
See also: '`DUP`'

8.13.9 ROT

Name: `ROT`
Stackeffect: nl n2 n3 — n2 n3 nl
Attributes: ISO,FIG,L0
Description: Rotate the top three values on the stack, bringing the third to the top.
See also: '`SWAP`'

8.13.10 SWAP

Name: `SWAP`
Stackeffect: n1 n2 — n2 n1
Attributes: ISO,FIG,L0
Description: Exchange the top two values on the stack.
See also: '`ROT`'

8.14 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 16 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.

8.14.1 0<

Name: 0<
Stackeffect: n — ff
Attributes: ISO,FIG,L0
Description: Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.
See also: '<'

8.14.2 0=

Name: 0=
Stackeffect: n — ff
Attributes: ISO,FIG,L0
Description: Leave a true flag 'ff' is the number 'n' is equal to zero, otherwise leave a false flag.
See also: '='

8.14.3 <>

Name: <>
Stackeffect: n1 n2 — ff
Attributes: ISO,L0
Description: Leave a true flag if 'n1' is not equal than 'n2' ; otherwise leave a false flag.
See also: '>' '=' '0<'

8.14.4 <

Name: <
Stackeffect: n1 n2 — ff
Attributes: L0,ISO
Description: Leave a true flag if 'n1' is less than 'n2' ;otherwise leave a false flag.
See also: '=' '>' '0<'

8.14.5 =

Name: =
Stackeffect: n1 n2 — ff
Attributes: ISO,FIG,L0
Description: Leave a true flag if 'n1=n2' ; otherwise leave a false flag.
See also: '<' '>' '0=' '-'

8.14.6 >

Name: >
Stackeffect: n1 n2 — ff
Attributes: ISO,L0
Description: Leave a true flag if ‘n1’ is greater than ‘n2’ ; otherwise leave a false flag.
See also: ‘<’ ‘=’ ‘0<’

8.14.7 AND

Name: AND
Stackeffect: n1 n2 — n3
Attributes: ISO,FIG,L0
Description: Leave the bitwise logical and of ‘n1’ and ‘n2’ as ‘n3’ .
See also: ‘XOR’ ‘OR’

8.14.8 INVERT

Name: INVERT
Stackeffect: n1 — n2
Attributes: ISO,L1
Description: Invert all bits of ‘n1’ leaving ‘n2’ . For pure flags (0 or -1) this is the *logical not* operator.
See also: ‘AND’ ‘OR’

8.14.9 OR

Name: OR
Stackeffect: n1 n2 — n3
Attributes: ISO,FIG,L0
Description: Leave the bit-wise logical or of two 16-bit values.
See also: ‘AND’ ‘XOR’

8.14.10 U<

Name: U<
Stackeffect: u1 u2 — ff
Attributes: ISO,L0
Description: Leave a true flag if ‘u1’ is less than ‘u2’ ; otherwise leave a false flag.(Interpreted as unsigned numbers).
See also: ‘<’

8.14.11 XOR

Name: `XOR`

Stackeffect: nl n2 — n3

Attributes: L1

Description: Leave the bitwise logical exclusive or of two 16-bit values.

See also: '`AND`' '`OR`'

8.15 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.

8.15.1 !

Name: `!`

Stackeffect: n addr —

Attributes: ISO,FIG,L0

Description: Store all 16 bits of n at '`addr`' .

See also: '`@`' '`C!`' '`2!`'

8.15.2 +!

Name: `+!`

Stackeffect: n addr —

Attributes: ISO,FIG,L0

Description: Add '`n`' to the value '`addr`'.

See also: '`TOGGLE`' '`!`'

8.15.3 2!

Name: `2!`

Stackeffect: addr— x1 x2

Attributes: ISO

Description: Store a pair of 16 bits values '`x1`' '`x2`' to consecutive cells at '`addr`' . '`x2`' is stored at the lowest address.

See also: '`2@`' '`!`' '`C!`'

8.15.4 2@

Name: 2@

Stackeffect: addr— x1 x2

Attributes: ISO

Description: Leave a pair of 16 bits values ‘x1’ ‘x2’ from consequitive cells at ‘addr’ . ‘x2’ is fetched from the lowest address.

See also: ‘2!’ ‘@’ ‘C@’

8.15.5 @

Name: @

Stackeffect: addr — n

Attributes: ISO,FIG,L0

Description: Leave the 16 bit contents ‘n’ of ‘addr’ .

See also: ‘!’ ‘C@’ ‘2@’

8.15.6 ALIGNED

Name: ALIGNED

Stackeffect: addr1 — addr2

Attributes: ISO

Description: Make sure that ‘addr1’ is *aligned* by advancing it if necessary to ‘addr2’. In this ciforth this is a NOOP.

See also: ‘ALIGN’

8.15.7 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. In this ciforth this is a NOOP.

See also: ‘ALIGNED’

8.15.8 BLANK

Name: BLANK

Stackeffect: addr count —

Attributes: ISO

Description: This is shorthand for “BL FILL ”.

8.15.9 BM

Name: `BM`
Stackeffect: — addr
Attributes:
Description: A constant leaving the address of the lowest memory in use by Forth.
See also: '`DP`' '`EM`'

8.15.10 C!

Name: `C!`
Stackeffect: b addr —
Attributes: ISO
Description: Store 8 bits of '`b`' at '`addr`' . In ciforth , running on the Intel architectures there are no restrictions regarding byte addressing.
See also: '`C@`' '`!`'

8.15.11 C@

Name: `C@`
Stackeffect: addr — b
Attributes: ISO
Description: Leave the 8 bit contents of memory address. In ciforth , running on the Intel architectures there are no restrictions regarding byte addressing.
See also: '`C!`' '`@`' '`2@`'

8.15.12 CELL+

Name: `CELL+`
Stackeffect: n1 — n2
Attributes: ISO
Description: Advance the memory pointer '`n1`' by one (in this case 16 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.

8.15.13 CELLS

Name: `CELLS`
Stackeffect: n1 — n2
Attributes: ISO
Description: Return the equivalent of '`n1`' cells in bytes: '`n2`'. This is invaluable for writing portable code. Much of the library code of ciforth runs on both 16 and 32 bits systems, thanks to this.
See also: '`CELL+`'

8.15.14 CHAR+

Name: `CHAR+`
Stackeffect: n1 — n2
Attributes: ISO
Description: Advance the memory pointer '`n1`' by one character to '`n2`'. In ciforth this means one byte. Bytes are the address units ISO is talking about. Unfortunately the ISO standard has no way to address bytes.
See also: '`CELL+`'

8.15.15 CHARS

Name: `CHARS`
Stackeffect: n1 — n2
Attributes: ISO
Description: Return the equivalent of '`n1`' chars in bytes: '`n2`'. In ciforth this is a NOOP. Unfortunately the ISO standard has no way to address bytes.
See also: '`CELLS`'

8.15.16 CMOVE

Name: `CMOVE`
Stackeffect: from to count —
Attributes:
Description: Move the specified quantity of characters beginning at address '`from`' to address '`to`' . The contents of address from is moved first proceeding toward high memory, such that memory propagation occurs. As the 6809 is an 8-bit processor byte-addressing has no restrictions in ciforth.

8.15.17 CORA

Name: `CORA`
Stackeffect: addr1 addr2 len —n
Attributes: CIF
Description: Compare the memory areas at '`addr1`' and '`addr2`' over a length '`len`' . For the first bytes that differ, return -1 if the byte from '`addr1`' is less (unsigned) than the one from '`addr2`', and 1 if it is greater. If all '`len`' bytes are equal, return zero. This is an abbreviation of `COMPARE-AREA` . It would have been named '`COMPARE`' , if that were not taken by ISO.

8.15.18 EM

Name: `EM`
Stackeffect: — addr
Attributes:
Description: A constant leaving the address just above the highest memory in use by Forth.
See also: '`DP`' '`BM`'

8.15.19 ERASE

Name: `ERASE`
Stackeffect: addr n —
Attributes: ISO
Description: This is shorthand for '`0 FILL`'.
See also: '`BLANK`' '`FILL`'

8.15.20 FILL

Name: `FILL`
Stackeffect: addr u b —
Attributes: ISO,FIG,L0
Description: If '`u`' is not zero, store '`b`' in each of '`u`' consecutive bytes of memory beginning at '`addr`' .
See also: '`BLANK`' '`ERASE`'

8.15.21 MOVE

Name: `MOVE`
Stackeffect: from to count —
Attributes:
Description: Move the '`count`' bytes beginning at address '`from`' to address '`to`', such that the destination area contains what the source area contained, regardless of overlaps. As the 6809 is an 8-bit processor byte-addressing has no restrictions in ciforth.

8.15.22 TOGGLE

Name: `TOGGLE`
Stackeffect: addr b —
Attributes:
Description: Complement the contents of '`addr`' by the bit pattern '`b`' .
See also: '`XOR`' '`+! `'

8.15.23 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<`'

8.16 MISC

The wordset ‘MISC’ contains words that defy categorisation.

8.16.1 .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’

8.16.2 EXECUTE

Name: EXECUTE

Stackeffect: xt —

Attributes: ISO,FIG,L0

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’

8.16.3 NOOP

Name: NOOP

No stackeffect

Attributes:

Description: Do nothing. Primarily useful as a placeholder.

8.16.4 TASK

Name: TASK

No stackeffect

Attributes:

Description: A no-operation word which marks the boundary between the forth system and applications.

See also: ‘COLD’

8.16.5 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`'

8.16.6 _

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.

8.17 MULTIPLYING

The 16 bits Forth's have problems with overflow (see Section 8.19 [OPERATOR], page 92). Operators with intermediate results of double precision, mostly scaling operators, solve this and are present in the '`MULTIPLYING`' wordset. but scaling remain tricky.. 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.

8.17.1 */MOD

Name: `*/MOD`
Stackeffect: n1 n2 n3 — n4 n5
Attributes: ISO,FIG,L0
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`'

8.17.2 */

Name: `*/`
Stackeffect: n1 n2 n3 — n4
Attributes: ISO,FIG,L0
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`'

8.17.3 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`' '`M/MOD`' '`/`' '`M*`'

8.17.4 M*

Name: `M*`

Stackeffect: n1 n2 — d

Attributes: ISO,FIG,L0

Description: A mixed magnitude math operation which leaves the double number '`d`' : the signed product of two signed number '`n1`' and '`n2`' .

See also: '`M/MOD`' '`SM/REM`' '`*`'

8.17.5 M/MOD

Name: `M/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*`'

8.17.6 SM/REM

Name: `SM/REM`

Stackeffect: d n1 — n2 n3

Attributes: ISO

Description: A mixed magnitude math operator which leaves the signed remainder '`n2`' and signed quotient '`n3`' from a double number dividend '`d`' and divisor '`n1`'. This is a symmetric division, i.e. the remainder takes its sign from the dividend.

See also: '`M/MOD`' '`/`' '`M*`'

8.17.7 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*’ ‘*’

8.17.8 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’ ‘/’

8.18 OPERATINGSYSTEM

The wordset ‘OPERATINGSYSTEM’ contains words that call the underlying operating system or functions available in the BIOS-rom.

8.18.1 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.
See also: ‘SYSTEM’

8.18.2 BYE

Name: BYE
Stackeffect: —
Attributes: ISO FIG
Description: Return to the host environment MSDOS , OS/2, Windows. In fact the server program is notified to stop, by sending a ^C
See also: ‘COLD’ ‘EXIT-CODE’

8.19 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 worrysome 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*). Having a mere 30000 for the number range can easily lead to overflow in intermediate results during *scaling* : a multiplication followed by a division. For example ‘: ADD10% 110 * 100 / ;’. There are special operators to get around that. See Section 8.17 [MULTIPLYING], page 90,.

8.19.1 *

Name: *

Stackeffect: n1 n2 — n3

Attributes: ISO,FIG,L0

Description: Leave the signed product ‘n3’ of two signed numbers ‘n1’ and ‘n2’ .

See also: ‘+’ ‘-’ ‘/’ ‘MOD’

8.19.2 +

Name: +

Stackeffect: n1 n2 — sum

Attributes: ISO,FIG,L0

Description: Leave the sum of ‘n1’ and ‘n2’ .

See also: ‘-’ ‘*’ ‘/’ ‘MOD’

8.19.3 -

Name: -

Stackeffect: n1 n2 — diff

Attributes: ISO,FIG,L0

Description: Leave the difference of ‘n1’ and ‘n2’ .

See also: ‘NEGATE’ ‘+’ ‘*’ ‘/’ ‘MOD’

8.19.4 /MOD

Name: /MOD

Stackeffect: n1 n2 — rem quot

Attributes: ISO,FIG,L0

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’

8.19.5 /

Name: `/`
Stackeffect: n1 n2 — quot
Attributes: ISO,FIG,L0
Description: Leave the signed quotient of '`n1`' and '`n2`' . (using *symmetric division*).
See also: '`+`' '`-`' '`*`' '`MOD`' '`*/MOD`'

8.19.6 ABS

Name: `ABS`
Stackeffect: n — u
Attributes: ISO,FIG,L0
Description: Leave the absolute value of '`n`' as '`u`' .
See also: '`DABS`'

8.19.7 LSHIFT

Name: `LSHIFT`
Stackeffect: u1 n — u2
Attributes: ISO
Description: Perform a '`logical shift` ' of the bits of '`u1`' to the left by '`n`' places. Put zero into the places uncovered by the shift.
See also: '`RSHIFT`' '`2*`'

8.19.8 MAX

Name: `MAX`
Stackeffect: n1 n2 — max
Attributes: ISO,FIG,L0
Description: Leave the greater of two numbers.
See also: '`MIN`'

8.19.9 MIN

Name: `MIN`
Stackeffect: n1 n2 — min
Attributes: ISO,FIG,L0
Description: Leave the smaller of two numbers.
See also: '`MAX`'

8.19.10 MOD

Name: `MOD`

Stackeffect: n1 n2 — mod

Attributes: ISO,FIG,L0

Description: Leave the remainder of '`n1`' divided by '`n2`' , with the same sign as '`n1`' (i.e. *symmetric division*).

See also: '`+`' '`-`' '`*`' '`/`' '`MOD`' '`*/MOD`'

8.19.11 NEGATE

Name: `NEGATE`

Stackeffect: n1 — n2

Attributes: ISO,FIG,L0

Description: Leave the two's complement of a number, i.e. '`n2`' is '`-n1`'

See also: '`-`'

8.19.12 RSHIFT

Name: `RSHIFT`

Stackeffect: u1 n — u2

Attributes: ISO

Description: Perform a '`logical shift` ' of the bits of '`u1`' to the right by '`n`' places. Put zero into the places uncovered by the shift.

See also: '`LSHIFT`' '`2/`'

8.20 OUTPUT

The wordset '`OUTPUT`' contains words to output to the terminal and such.

See Section 8.1 [BLOCKS], page 42, for blocks.

8.20.1 (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`'

8.20.2 (EMIT)

Name: (EMIT)
Stackeffect: c —
Attributes:
Description: Transmit ASCII character ‘c’ to the terminal.
See also: ‘EMIT’

8.20.3 ."

Name: ."
No stackeffect
Attributes: ISO,FIG,I,L0
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’

8.20.4 .(

Name: .(
No stackeffect
Attributes: I,L0
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’ ‘."’

8.20.5 .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)’

8.20.6 .

Name: .
Stackeffect: n —
Attributes: ISO,FIG,L0
Description: Print the number ‘n1’ observing the current BASE , followed by a blank.
See also: ‘OUT’ ‘U.’ ‘.R’ ‘D.R’ ‘D.’ ‘(D.R)’

8.20.7 ?

Name: ?
Stackeffect: addr —
Attributes: ISO,FIG,L0
Description: Print the value contained at the address 'addr' observing the current BASE , followed by a blank.
See also: 'OUT' '.'

8.20.8 CR

Name: CR
No stackeffect
Attributes: ISO,FIG,L0
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'

8.20.9 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)'

8.20.10 D.

Name: D.
Stackeffect: d —
Attributes: ISO,FIG,L1
Description: Print the signed double number 'd', observing the current BASE , followed by a blank.
See also: 'OUT' '.' 'D.R' '(D.R)'

8.20.11 EMIT

Name: EMIT
Stackeffect: c —
Attributes: ISO FIG L0
Description: Transmit ASCII character 'c' to the terminal. All terminal I/O goes through this word. It is high level so terminal I/O can be redirected, by *revectoring* it. OUT is incremented for each character output and reset for a carriage return.
See also: 'TYPE' 'OUT'

8.20.12 ETYPE

Name: `ETYPE`

Stackeffect: addr count —

Attributes:

Description: This behaves identical to `TYPE` . However it is used for all error message, so via this word error output can be redirected, by *revectoring* it.

8.20.13 H.

Name: `H.`

Stackeffect: x —

Attributes:

Description: Transmit '`x`' to the terminal in hexadecimal.

See also: '`EMIT`'

8.20.14 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`'

8.20.15 SPACES

Name: `SPACES`

Stackeffect: n —

Attributes: ISO,FIG,L0

Description: If '`n`' is greater or equal to zero, display as much spaces.

See also: '`SPACE`' '`OUT`'

8.20.16 SPACE

Name: `SPACE`

No stackeffect

Attributes: ISO,FIG,L0

Description: Transmit an ASCII blank to the output device.

See also: '`EMIT`' '`OUT`'

8.20.17 TYPE

Name: `TYPE`

Stackeffect: addr count —

Attributes: ISO FIG L0

Description: Transmit count characters from '`addr`' to the output device. `OUT` is incremented for each character output. In this ciforth all terminal I/O goes through `EMIT` , so `OUT` is observed.

See also: '`EMIT`' '`OUT`'

8.20.18 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)`'

8.21 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.

8.21.1 (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 constant*s, you may not alter its content, nor assume anything is appended. So no leading delimiters are skipped. The difference with an ISO '`PARSE`' is that ISO considers control characters a match for a blank.

See also: '`WORD`' '`(WORD)`'

8.21.2 (WORD)

Name: `(WORD)`

Stackeffect: — sc

Attributes: CI L0

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`'

8.21.3 (

Name: `(`
No stackeffect
Attributes: ISO,FIG,I,L0
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: '\'

8.21.4 ?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.
See also: '`BL`' '`SPACE`'

8.21.5 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]`' ''''

8.21.6 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`'

8.21.7 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* . Note that the denotations at the end of the `FORTH` wordlist match numbers. 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 `DENOTATION` wordlist, depending on the first character.

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`'

8.21.8 IN[]

Name: `IN[]`

Stackeffect: —addr c

Attributes: CI L0

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`' '`IN`'

8.21.9 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 followed by a call to `INTERPRET` .

See also: '`EVALUATE`' '`INTERPRET`'

8.21.10 SOURCE

Name: `SOURCE`

Stackeffect: — addr n1

Attributes: ISO

Description: Return the address and length of the *current input source* .

See also: '`SRC`' '`SOURCE-ID`'

8.21.11 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 and non-inclusive highest 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 using 'RESTORE-INPUT', or changing only the third cell. The third cell has the alias 'IN' .

Words like '>IN BLK SOURCE SOURCE-ID' are secondary, and return their output by "second-guessing" 'SRC' .

See also: 'BLK' 'SOURCE-ID'

8.21.12 STATE

Name: STATE

Stackeffect: — addr

Attributes: ISO,L0,U

Description: A user variable containing the compilation state. A non-zero value indicates compilation.

8.21.13 WORD

Name: WORD

Stackeffect: c —addr

Attributes: ISO,FIG,L0

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: '(WORD)' '(PARSE)' 'BLK ' 'IN'

8.21.14 [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'

8.21.15 [

Name: `[`

No stackeffect

Attributes: ISO,FIG,I,L1

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` ' '`]`'

8.21.16 \

Name: `\`

No stackeffect

Attributes: ISO,I,L0

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. A blank after the word `\` is required.

See also: '`(`'

8.21.17]

Name: `]`

No stackeffect

Attributes: ISO,FIG,L1

Description: Resume compilation, to the completion of a colon-definition.

See also: '`[`'

8.22 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.

8.23 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.

8.23.1 !CSP

Name: !CSP
No stackeffect
Attributes:
Description: Save the stack position in CSP . Used as part of the compiler security.

8.23.2 ?COMP

Name: ?COMP
No stackeffect
Attributes:
Description: Issue error message if not compiling.
See also: '?ERROR'

8.23.3 ?CSP

Name: ?CSP
No stackeffect
Attributes:
Description: Issue error message if stack position differs from value saved in 'CSP' .

8.23.4 ?DELIM

Name: ?DELIM
No stackeffect
Attributes:
Description: Parse a character and issue error message if it is not a blank delimiter.

8.23.5 ?EXEC

Name: ?EXEC
No stackeffect
Attributes:
Description: Issue an error message if not executing.
See also: '?ERROR'

8.23.6 ?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`'

8.23.7 ?STACK

Name: `?STACK`

No stackeffect

Attributes:

Description: Issue an error message if the stack is out of bounds.

See also: '`?ERROR`'

8.23.8 CSP

Name: `CSP`

Stackeffect: — addr

Attributes: U

Description: A user variable temporarily storing the stack pointer position, for compilation error checking.

8.24 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.

8.24.1 .S

Name: `.S`

Stackeffect: from to —

Attributes:

Description: Print the stack, in the current base.

See also: '`LIST`'

8.24.2 >R

Name: `>R`

Stackeffect: n —

Attributes: ISO,FIG,C,L0

Description: Remove a number from the *data stack* and place as the most accessable on the *return stack* . Use should be balanced with `R>` in the same definition.

See also: '`R@`'

8.24.3 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@`'

8.24.4 DSP!

Name: `DSP!`

Stackeffect: addr —

Attributes:

Description: Initialize the data stack pointer with '`addr`' .

See also: '`DSP@`'

8.24.5 DSP@

Name: `DSP@`

Stackeffect: — addr

Attributes:

Description: Return the address '`addr`' of the data stack position, as it was before `DSP@` was executed.

See also: '`S0`' '`DSP!`'

8.24.6 R0

Name: `R0`

Stackeffect: — addr

Attributes: U

Description: A user variable containing the initial location of the return stack.

See also: '`RSP!`'

8.24.7 R>

Name: `R>`
Stackeffect: — n
Attributes: ISO,FIG,L0
Description: Remove the top value from the *return stack* and leave it on the *data stack* .
See also: '`>R`' '`R@`'

8.24.8 R@

Name: `R@`
Stackeffect: — n
Attributes: ISO
Description: Copy the top of the return stack to the data stack.
See also: '`>R`' '`<R`'

8.24.9 RDROP

Name: `RDROP`
Stackeffect: —
Attributes:
Description: Remove the top value from the return stack.
See also: '`>R`' '`R@`' '`R>`'

8.24.10 RSP!

Name: `RSP!`
Stackeffect: addr —
Attributes:
Description: Initialize the return stack pointer with '`addr`'.
See also: '`RSP@`'

8.24.11 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!`'

8.24.12 S0

Name: `S0`

Stackeffect: — addr

Attributes: U

Description: A user variable that contains the initial value for the data stack pointer.

See also: '`DSP!`'

8.25 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).

8.25.1 $!-BD

Name: `$!-BD`

Stackeffect: sc addr —

Attributes:

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`' '`$!`'

8.25.2 $!

Name: `$!`

Stackeffect: sc addr —

Attributes:

Description: Store a *string constant* '`sc`' in the string variable at address '`addr`'.

See also: '`$@`' '`$+!`' '`$C+`'

8.25.3 $+!

Name: `$+!`

Stackeffect: sc addr —

Attributes:

Description: Append a *string constant* '`sc`' to the string variable at address '`addr`'.

See also: '`$@`' '`$!`' '`$C+`'

8.25.4 $,

Name: `$,`
Stackeffect: sc — addr
Attributes:
Description: Allocate and store a *string constant* '`sc`' in the dictionary and leave its address '`addr`'.
See also: '`$@`' '`$!`'

8.25.5 $@

Name: `$@`
Stackeffect: addr — sc
Attributes:
Description: From address '`addr`' fetch a *string constant* '`sc`' .
See also: '`$@`' '`$!`' '`$+!`' '`$C+`'

8.25.6 $C+

Name: `$C+`
Stackeffect: c addr —
Attributes:
Description: Append a char '`c`' to the string variable at address '`addr`'.
See also: '`$@`' '`$!`' '`$+!`'

8.25.7 $I

Name: `$I`
Stackeffect: sc c—addr
Attributes:
Description: Find the first '`c`' in the *string constant* '`sc`' and return its '`addr`' if present. Otherwise return a *nil pointer*.
See also: '`$S`' '`CORA`'

8.25.8 $S

Name: `$S`
Stackeffect: sc c—sc1 sc2
Attributes:
Description: Find the first '`c`' in the *string constant* '`sc`' and split it at that address. Return the strings after and before '`c`' into '`sc1`' and '`sc2`' respectively. If the character is not present '`sc1`' is a null string (its address is zero) and '`sc2`' is the original string. Both '`sc1`' and '`sc2`' may be empty strings (i.e. their count is zero), if '`c`' is the last or first character in '`sc`' .
See also: '`$I`' '`CORA`'

8.25.9 -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`'

8.25.10 BL

Name: `BL`

Stackeffect: — c

Attributes: ISO.FIG

Description: A constant that leaves the ASCII value for "blank".

8.25.11 COUNT

Name: `COUNT`

Stackeffect: addr1 — addr2 n

Attributes: ISO,FIG,L0

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* '`addr n`' from the brain damaged string variable at '`addr1`' .

See also: '`TYPE`'

8.25.12 S"

Name: `S"`

Stackeffect: — addr1 n

Attributes: ISO,L0

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: '`"`'

8.26 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.

8.26.1 0

Name: `0`
Stackeffect: — 0
Attributes:
Description: Leave the number 0.
See also: '`CONSTANT`'

8.26.2 1+

Name: `1+`
Stackeffect: n1 — n2
Attributes: L1
Description: This is shorthand for "'`1`' `+` ".
See also: '`CELL+`' '`1-`'

8.26.3 1-

Name: `1-`
Stackeffect: n1 — n2
Attributes: ISO
Description: This is shorthand for '`1 -`'.
See also: '`1+` '

8.26.4 2*

Name: `2*`
Stackeffect: n1 — n2
Attributes: ISO
Description: Perform an arithmetical 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/` '

8.26.5 2/

Name: `2/`
Stackeffect: n1 — n2
Attributes: ISO
Description: Perform an arithmetical 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 SWAP DROP`'. It is not the same as '`2 /`', nor is it the same as '`1 RSHIFT`'. This confusing word should never be used.
See also: '`2*` '

8.26.6 Number_1

Name: `Number_1`
Stackeffect: — 1
Attributes:
Description: Leave the number 1.
See also: '`CONSTANT`'

8.26.7 Number_2

Name: `Number_2`
Stackeffect: — 2
Attributes:
Description: Leave the number 2.
See also: '`CONSTANT`'

8.27 WORDLISTS

The dictionary is subdivided in non-overlapping subsets: the *word list* 's (see Section 8.6 [DICTIONARY], page 59). There is one exception: the DENOTATION wordlist is also part of the FORTH word list. They are created by the defining word `VOCABULARY` and filled by defining words while that vocabulary 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 vocabulary words created by the defining word `VOCABULARY` . 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.

8.27.1 ALSO

Name: `ALSO`
No stackeffect
Attributes: ISO
Description: Duplicate the topmost *WID* in the search order stack. If there were already 8 WID 's, ciforth looses the last one. This is not counting the `ONLY` search order.
See also: '`CONTEXT`' '`VOCABULARY`'

8.27.2 ASSEMBLER

Name: `ASSEMBLER`
No stackeffect
Attributes: CI
Description: The name of the `ASSEMBLER` vocabulary. The associated *word list* contains assembler instructions. This wordlist is as yet empty.
See also: '`VOCABULARY`'

8.27.3 CONTEXT

Name: `CONTEXT`

Stackeffect: — addr

Attributes: FIG,U,L0

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 8 WID 's in this ciforth, while the ISO Search-Order wordset requires a capacity of at least 8.

See also: '`PRESENT`' '`VOCABULARY`' '`CURRENT`'

8.27.4 CURRENT

Name: `CURRENT`

Stackeffect: — addr

Attributes: FIG,U,L0

Description: A user variable containing the WID of a vocabulary 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: '`VOCABULARY`' '`CONTEXT`'

8.27.5 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 vocabulary 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`' '`VOCABULARY`'

8.27.6 DENOTATION

Name: `DENOTATION`

No stackeffect

Attributes: CI

Description: The name of the `DENOTATION` vocabulary. The associated *word list* contains prefix words, called *denotation* definitions. In ciforth all words of `DENOTATION` belong to the minimum search order . If found the parse pointer is moved back to immediately after the prefix, and the corresponding denotation definition executed. This word list must be used with care as a `CONTEXT` word list; and only as a `CURRENT` word list whenever you want to add a denotation. The word `FORTH` is hidden by the word `F` . Use '`ONLY FORTH`' to regain control.

See also: '`VOCABULARY`'

8.27.7 ENVIRONMENT

Name: `ENVIRONMENT`

No stackeffect

Attributes: CI

Description: The name of the `ENVIRONMENT` vocabulary. 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: '`VOCABULARY`'

8.27.8 FORTH

Name: `FORTH`

No stackeffect

Attributes: NISO,FIG,I,L1

Description: The name of the primary vocabulary. Execution pushes the `FORTH` *WID*

to the top of the *search order* . (For ISO-compliance it would replace the top.) Until additional user *word list* 's are created, new user definitions become a part of `FORTH` . `FORTH` is immediate, so it will execute during the creation of a colon-definition, to select this *word list* at compile time.

See also: '`CONTEXT`' '`VOCABULARY`'

8.27.9 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: '`VOCABULARY`'

8.27.10 ONLY

Name: `ONLY`

No stackeffect

Attributes: NISO

Description: Make `CONTEXT` , the search order stack, empty, leaving the minimum search order , which is approximately the `DENOTATION` word list plus the word `FORTH` . By using `FORTH` one can regain control.

See also: '`VOCABULARY`'

8.27.11 PREVIOUS

Name: `PREVIOUS`

No stackeffect

Attributes: ISO

Description: Pop the topmost *WID* from the search order stack. If empty still the `ONLY` search order is left.

See also: '`CONTEXT`' '`VOCABULARY`'

8.27.12 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 `VOCABULARY` . All vocabulary names are linked by these fields to allow `FORGET` to find all vocabularies.

See also: '`VOCABULARY`'

Glossary Index

This index finds the glossary description of each word.

!

#

$

'

(

*

,

-

.

/

:

;

=

?

@

[

]

_

+

>

\

<

0

1

2

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Forth Word Index

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

=

?

@

[

]

_

~

"

+

>

\

<

0

1

2

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

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.

P

R

S

T

U

V

W

Short Contents

Table of Contents