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 yourforth:

- version snapshot 6.55
- 32-bits protected mode
- running under Linux
- the full ISO CORE set is present, possibly after loading

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 7, Section Portability.

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

ciforth consists of three files:

- '`lina`' : 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 '`fig86.gnr`' . The documentation applies to the ciforth with which it goes.

If your Forth doesn't fit the description below get a new version. The information below allows an expert to reconstruct how to generate a corresponding version (see Chapter 3 [Rationale & legalese], page 5).

These are the features:

All ciforth's are *case sensitive* . This is version snapshot 6.55. It is running in protected mode. It is running under Linux Blocks are allocated in files. A number has a precision of 32 bits. It calls linux system from Forth directly. It use `IN` instead of the ISO `>IN` 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:
'`yourforth`'
Like in a Disk Operating System a *word* is executed by typing its name, but unlike in a DOS several programs can be specified on the same line, interspersed with numbers. Also names can be anything, as long as they don't contain spaces.

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

For example:

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

80386 ciforth beta $RCSfile: ci86.gnr,v $ $Revision: 3.275 $

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. yourforth is case sensitive. If you want to change that consult the section "Common problems". (see Chapter 6 [Errors], page 29).

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

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

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

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

3 Rationale & legalese

3.1 Legalese

This application currently is copyright by Albert van der Horst. This Forth is called ciforth and is made available by the D.F.W.. 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 accompany 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 "normal use in the Forth sense" as expressed by the following statement.

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

The GPL is interpreted in the sense that a system based on ciforth and intended to serve a particular purpose, that purpose not being a "general purpose Forth system", is considered normal use of the compilation system, even if it could accomplish everything ciforth could, under the condition that the ciforth it is based on is available in accordance to the GPL rules, and this is made known to the user of the derived system.

So if you base an application program on ciforth, you need not make its source available, even if it would be a derived system in a strict interpretation of the GPL.

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 yourforth is found at
'`http://home.hccnet.nl/a.w.m.van.der.horst/ciforth.html`'.

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

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

Important:
If you just want to use a Forth, you most certainly do not want the generic system. Great effort is expended in making sure that this manual contains all that you need, and nothing that might confuse you. The generic system on the contrary contains lots that you don't need, and is confusing as hell.

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

3.4 The Generic System this Forth is based on.

The source and executable of this ciforth was generated, out of at least dozens of possibilities, by a generic system. You can configure the operating system, memory sizes, file names and minor issues like security policy. You can select between a 16 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 '`yourforth`' to get into your interactive Forth system. You will see a signon message. While sitting in your interactive Forth doing a "hello world" is easy:

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

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

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

This means you type the command '`HELLO`' while you are in yourforth. As soon as you leave yourforth, the new command is gone.

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

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

This file can be `INCLUDED` inorder to add the command '`HELLO`' to your Forth environment, like so:

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

During development you probably have started with '`yourforth -e`', so you need just type

```
INCLUDE hello.frt
```

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

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

If that failed, or anything else fails, you will get a message with at least ' `ciforth ERROR ###`' and hopefully some more or less helpful text as well. The '`###`' is an error number. See Chapter 6 [Errors], page 29, 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 yourforth you never have to worry about the life time of those quoted strings, they are allocated in the dictionary and are permanent.

4.1.2 The library.

If you want to run a program written on some other Forth, it may use facilities that are not available in yourforth'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 yourforth does.

yourforth uses the *blocks* as a library by addition of the word `WANTED` and a convention. Starting with '`yourforth -w`' or most any option you have this facility available. If you are already in yourforth, you can type '`1 LOAD`'.

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

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

S[ ] OK 1

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 29, Section Explanations. Apparently, yourforth doesn't know about a forth word named `TUCK` , but after '`"TUCK" WANTED`' maybe it does. You may try again.

The convention about the way the library file must be organized for `WANTED` to find something is simple. It is divided into blocks of 16 lines. The first line is the header of the block. If the word we are looking for is mentioned in the header, that block is compiled. This continues until the word has been defined, or the end of the search area is reached. This is marked by a screen with an empty index line. I tell you this not because you need to know, but to show that there is nothing to it.

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

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

4.1.3 Development.

If you want to try things out, or write a program – as opposed to just running a ready made program – you best start up yourforth by '`yourforth -e`'. That is `e` for elective. That name means that you can configure this screen 5 to suit your particular needs.
You will have available:

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

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

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

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

   . . .

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

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

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

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

4.1.4 Finding things out.

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

```
WANT TUCK
LOCATE TUCK
```

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

```
WANT TUCK
SEE TUCK
```

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

4.2 Configuring

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

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

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

`FAR-DP` allows to have a disposable part of the dictionary. This may be occasionally useful, but make sure to `FORGET` always the disposed off words. The '`-c`' option uses this to avoid having source files as part of an executable image.

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 8.5 [DICTIONARY], page 47). Each word owns a part of the dictionary, starting with its name and ending where the name of the next word starts. This structure is called a *dictionary entry* . Its address is called a *dictionary entry address* or *DEA* . In ciforth's this address is used for external reference in a consistent way. For example it is used as the *execution token* of a word in the ISO sense. In building a word the boundary between the dictionary and the free space shifts up. This process is called *allocating* , and the boundary is marked by a *dictionary pointer* called `DP` . A word can be executed by typing its name. Each word in the dictionary belongs to precisely one *word list* , or as we will say here namespace. Apart from the name, a word contains data and executable code, (interpreted or not) and linking information (see ⟨undefined⟩ [NAMESPACE], page ⟨undefined⟩).

The concept word list is part of the ISO standard, but we will use *namespace* . A namespace is much more convenient, being a word list with a name, created by `NAMESPACE` . ISO merely knows *word list identifier* 's, a kind of handle, abbreviated as *WID* . A new word list is created by the use of `NAMESPACE` , and by executing the namespace word the associated word list is pushed to the front of the search order. In fact in ciforth's every DEA can serve as a WID.

It defines a wordlist consisting of itself and all earlier words in the same namespace. You can derive the WID from the DEA of a namespace by `>WID` .

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

A high level word, one defined by `:` , is little more than a sequence of addresses of other words. The *inner interpreter* takes care to execute these words in order. It acts by fetching the address pointed by '`HIP`' , storing this value in register '`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 8.3 [DEFINING], page 43). The new word is created in the `CURRENT` word list .

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

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

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

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

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

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

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

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

4.4 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 as given by strerror.
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 use IN instead of the ISO >IN

   The >IN that is there is a fake, that can only be read, not changed.
5. Counting in do loops do not wrap through the boundary between negative and positive numbers. This is not useful on Forths of 32 bits and higher; for compatibility among ciforths 16 bit ciforths don't wrap either.
6. 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.
7. This is not strictly non-conforming, but worth mentioning here. In fact yourforth contains only one state-smart word besides SLITERAL (that word is ."). All denotations are state-smart, but the result is correct ISO behaviour for numbers. Knowledge of this is used freely

in the libraries of ciforth; it is the right of a system developer to do so. The library is not a supposedly ISO-conforming program. It tends to rely on ciforth-specific and yourforth-specific – but hopefully documented – behaviour. Understanding it requires some study of non-portable facilities.

Here we will explain how you must read the glossary of yourforth, 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 "yourforth", 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 yourforth 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 Compatibility with yourforth 4.0.x

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

What the legacy items are can be seen from the screen that has `_legacy_` in its index line. In particular `REQUIRE REQUIRED PRESENT? VOCABULARY WORD FIND (WORD) (PARSE) SAVE-INPUT RESTORE-INPUT` WORD FIND (WORD) (PARSE) SAVE-INPUT RESTORE-INPUT are to be found in that screen. Note that by using legacy items your code may be in conflict with upcoming standards.

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

4.6 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.

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

This has all been sorted out for you. Just use `SAVE-SYSTEM` . This accepts a string, the name you want the program-file to have.

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

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

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

4.7 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, disk block buffers and terminal input buffer.

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

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

4.7.1 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.

4.7.2 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.

4.7.3 Machine Code Definitions

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

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

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

Note: there is a separate document for if you want to change this ciforth's assembler source to fit your needs, although I strove to make it self contained. It includes assembling and linking instructions.

4.7.4 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.

Again standard definitions words are perfectly normal dictionary entries.

4.7.5 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.

4.7.6 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 '`WANT ASSEMBLERi86`'. Beware! The assembler can only be loaded on top of a `CASE-SENSITIVE` system. It automatically loads the proper 32-bits version. You can load a more elaborate assembler. See Chapter 5 [Assembler], page 23, Section Overview. They are among the first candidates to be integrated into your system by `SAVE-SYSTEM` .

An editor is not part of ciforth as delivered. Development in Linux uses the there available editors. Even without tools, code can be tested by piping it into Forth, then commanding Forth to look to the console, as follows :
' `(echo 8 LOAD; cat pascal.frt - )| yourforth` '
Primitive and preliminary as this may seem, it has been used for quite substantial developments like the 80386 assembler.
More advanced is using Your Favorite Editor, followed by including files:
' `"vim mysrc.frt" SYSTEM` '
' `"mysrc.frt" INCLUDED` '
See Chapter 4 [Manual], page 7, Section Getting Started.

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

It is essential that you regard yourforth 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.

Again words belonging to tools are perfectly normal dictionary entries.

4.7.7 RAM Workspace

The RAM workspace contains the compilation space for the dictionary, disc buffers, the computation and return stacks, the user area, and the terminal input buffer,

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

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

4.8 Specific layouts

4.8.1 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.
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 namespace. This indicates that it should not be executed.
- The DENOTATION bit =1 for a prefix word. This means that it is a short word used as a prefix that can parse all *denotation* 's (numbers) that start with that prefix, e.g. 7 or & . Usually it is a one letter word, but not necessarily. All built-in prefix words are part of the minimum search order.
- After the last letter follow zero bytes up till the next cell boundary.

(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 namespace 'DICTIONARY'. See Section 8.5 [DICTIONARY], page 47,, 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.

4.8.2 Details of memory layout

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

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

It is comprised of an integral number of buffers, each B/BUF bytes plus two cells. B/BUF is the number of bytes read from the disc in one go. 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 to contain 0x40 cells, most of it unused. User variables can be added by the word USER , but you have to keep track yourself which offset in the user area can be used. The user area is just under the disc buffers. So it ends at FIRST .

The terminal input buffer and the return stack share an area configured at a size of 0x10000 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.

4.8.3 Terminal I/O and vectoring.

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

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

A similar technique is not so useful on the input side, because keys entered during `EXPECT` are subject to correction until <RET> has been pressed. On yourforth `EXPECT` is left to the operating system, such that inputting to yourforth has the same look and feel as other input. Text can be pasted in with the mouse, etc. Consequently `RUBOUT` is not used.

4.9 Libraries and options

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

Screen 0 and screen 1 to 31 are reserved.

When a Forth is started up with a first parameter that is a one-letter option, the corresponding screen is to be executed.

4.9.1 Options

ciforth is a primitive system, and can interpret just one option on the command line. If the first argument is not starting with `-` ciforth returns with error code 3. However the option ‘`-l`’ can bootstrap it into more sophisticated behaviour.

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

- ‘`-a`’

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

- ‘`-c name`’

  Compile the file ‘`name`’ to an executable binary. The name of the binary is ‘`name`’ without the trailing ‘`.frt`’ or else ‘`a.out`’. Upon invocation of the binary the word defined latest is executed, then Forth goes `BYE` .

  Also `WANTED` is made available.

- '`-e`'

  Load the elective screen, screen 5. This contains *preferences* , the tools you want to have available running an interactive Forth. The default library file contans system wide default preferences. See the '`-l`' option if the default preferences don't suite you.

- '`-f forthcode`'

  Execute the '`forthcode`' in the order present.

- '`-g number name`'

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

- '`-h`'

  Print overview of options.

- '`-i binpath libpath [shellpath]`'

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

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

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

  For a smallish system you may expand by 0 Mbyte '`-g 0`'. If the system has no swap space, and less than 8 Mbyte of memory, use '`-g -3`', diminish from 4 to 1 Mbyte.

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

  In ciforth all options are mapped onto a letter. All options that start with `-` are mapped onto *m*. The result is the combination of -h and -v , so both help and version information is printed.

- '`-l name [more]`'

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

  Note that the default file is opened for reading only.

- '`-p`'

  Be pedantic about ISO. Redefine some words to follow the standard as closely as possible.

- '`-s script`'

  Load the file '`script`' , but ignore its first line. This is intended to be used for scripts, i.e. a piece of code to be interpreted rather than compiled.

  In a script `WANTED` is available and you can use standard in and standard out.

- '`-t sourcefile`'

  Try to load the file '`script`' automatically, by possibly unsafe means. Report facilities that were required. This is a first step in a porting activity.

- '`-v`'

  Print version and copy right information.
- '`-w`'

  Make sure `WANTED` is available.

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

4.9.2 Private libraries

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

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

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

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

4.9.3 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 7, 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

The assembler is described in this manual, because it is not feasable to use it from the description in the source. This chapter is about the assembler itself, not about how it is used in relation with ciforth.

5.1 Introduction

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

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

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

forth.lab : equivalent to asgen.frt plus asi86.frt plus asipentium.frt but without error detection.
'`ass.frt`' : the 80-line 8086 assembler (no error detection), a prototype.
'`as6809s.frt`' : a small 6809 assembler (no error detection).
'`asgen.frt`' : generic part of postit/fixup assembler
'`as80.frt`' : 8080 assembler, requires '`asgen.frt`'
'`asi86.frt`' : 8086 assembler, requires '`asgen.frt`'
'`asi386.frt`' : 80386 assembler, requires '`asgen.frt`'
'`aspentium.frt`' : general Pentium non-386 instructions, requires '`asgen.frt`'
'`asalpha.frt`' : DEC Alpha assembler, requires '`asgen.frt`'
'`asi6809.frt`' : 6809 assembler, requires '`asgen.frt`'
'`ps.frt`' : generate opcode sheets
'`p0.asi386.ps`' : first byte opcode for asi386 assembler
'`p0F.asi386.ps`' : two byte opcode for same that start with 0F.
'`test.mak`' : makefile, i.e. with targets for opcode sheets.

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

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

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

5.2 Reliability

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

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

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

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

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

5.3 Principle of operation

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

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

```
INC [BX]
```

Are we to increment the byte or the word at BX? Intel's solution is '`INC BYTE POINTER BX`')
Contrarily here we adapt the rule: if an instruction doesn't determine the operand size (some do, like `LEA,`), then a size fixup is needed ('`X|`' or '`B|`').
In this assembler this looks like

```
INC, B| 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. Other fixups determine registers and addressing mode.

- COMMA phase: First check whether the fixups have filled up all holes. Then add addresses (or offsets) and/or immediate data, using e.g. `IL,` or `L,`
- Check whether all commaers, requested either by postit's or fixup's are present. This check is actually executed by a postit like `MOV,`

  prior to assembling, or by `END-CODE` .

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

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

Now if you are serious, you have to study the '`asgen.frt`' and '`as80.frt`' sources. You better get your feet wet with '`as80.frt`' before you attack the Pentium. The way '`SIB`' is handled is so clever, that sometimes I don't understand it myself. It deviates somewhat from the phases explained here.

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

5.4 The 8080 assembler

The 8080 assembler doesn't take less place than Cassady's . (In the end the postit-fixup makes the Pentium assembler more compact, but not the 8080.) But... The regularities are much more apparent. It is much more difficult to make a mistake with the code for the '`ADD`' and '`ADI`' instructions. And there is information there to the point that it allows to make a disassembler that is independant of the instruction information, one that will work for the 8086, look at the pop family. First I had

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

(cause I started from an existing assembler.) But of course `RST` (the restart instruction) has nothing to do with registers, so it gets 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 asi386.ps". For the opcode sheets featuring a n-byte prefix you must pass the '`PREFIX`' to make and a '`MASK`' that covers the prefix and the byte opcode, e.g. '`make asi386.ps MASK=FFFF PREFIX=0F`' The opcode sheets '`p0.asi386.ps`' and '`p0F.asi386.ps`' are already part of the distribution and can be printed on a PostScript printer or viewed with e.g. '`gv`'.

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

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

5.6 16 and 32 bits code and segments

Note that the handling of segments and mixing thereof changed starting ciforth version 4.0.6. Contrary to the old situation, commaers are checked correctly in the presence of prefixes and you have never to overrule error-checking.

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

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

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

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

In comma-ing, you must always select the proper one, xell-wide commaers always contain either *`W`* or *`L`* .

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

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

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

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

and `L,` , or between `RW,` and `RL,` .

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

that is 16 bits in an otherwise 32 bits environment.

5.7 The built in assembler

From within ciforth one can load an assembler from the installed LAB library by the command `WANT ASSEMBLERi86` . Automatically a 32 bit assembler is loaded if the Forth itself is 32 bits and a 16 bit assembler for the 16 bit forths. This is a simplified version with no error checking and no provisions for 16/32 bit mixing. (Those are not needed, because you can mix with impunity.) In the future this assembler is now (since 4.0.6) fully compatible with the large file-based one.

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

The built in assembler has no error checking.

IMPORTANT NOTE: The 4.0.6 version and later contains assembler code in the LAB file that has not yet been converted. This code largely relates to a booting version; It will be updated as soon as I have a booting version in a binary form available.

5.8 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, BO| ~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,`'

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

Table 3 SIB-byte fixups.

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

5.9 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.

The errors whose message starts with '`AS:`' are used by the PostIt FixUp assembler in the file '`asgen.frt`'. See Chapter 6 [Errors], page 29, 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,` .

- '`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.

- ‘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.

- ‘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 error always displays the text "ciforth ERROR #" plus the error number, immediately and directly.

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

6.2.1 Error 11 or 12 caused by lower case.

If you type a standard word like `words` in lower case, it will not be recognised, resulting in error 11. Similarly `' words` results in error 12. This is because yourforth is *case sensitive* , i.e. the difference between lower and upper case is significant .

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

Case insensitivity applies to the words looked up in the dictionary, not to hex digits.

6.2.2 Error 8 or only error numbers

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

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

6.2.3 Error 8 while editing a screen

If after editing a screen, you get error 8, the screen has not been written to disk, because you have no write access for the library file. You must issue `DEVELOP` which reopens the library file in READ_WRITE mode. Normally this should be part of loading the `EDITOR` . It may be of

course that you don't have privilege to write to the file. As non-privileged user you cannot edit the system-wide library file.

You may always edit and use a private copy of the library file, by the '`-l`' option. See Chapter 4 [Manual], page 7, for how options work.

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 yourforth, but maybe by other version of ciforth or even a differently configured yourforth. 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 23).

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

.

Here are the error explanations.

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

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

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

  dnl_END_(_SECURITY_)

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

  Not used.

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

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

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

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

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

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

- '`ciforth ERROR # 6 : DISK RANGE ?`'

  Reading to the terminal input buffer failed. The message is probably inappropriate.

- '`ciforth ERROR # 7 : FULL STACK`'

  dnl_END_(_SECURITY_)

- '`ciforth ERROR # 8 : ERROR ACCESSING BLOCKS FROM MASS STORAGE`'

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

This is detected by ?DISK-ERROR called from places where a disk access has occurred. It may be that the library file has not been properly installed. Check the content of BLOCK-FILE . You may not have the right to access it. Try to view the file. Normally the library file is opened read-only. If you want to edit it make sure to do DEVELOP in order to reopen it in read/write mode. 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. This will only happen with user-defined denotations. Built-in denotations are in the ONLY namespace, that can only be accessed last, because it ends the search order. Note that hex digits must be typed in uppercase, even if "CASE-SENSITIVE" is in effect. Error 10 may be caused by using lower case where upper case is expected, suach as for ISO standard words. See the section "Common problems" in this chapter if you want to make ciforth case insensitive.

- 'ciforth ERROR # 11 : WORD IS NOT FOUND'

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

- 'ciforth ERROR # 12 : NOT RECOGNIZED'

  The string printed was not found in the dictionary, nor does it match a number, or some other denotation. This may be caused by using lower case where upper case is required for ISO standard words or for hex digits. See the section "Common problems" in this chapter if you want to make ciforth case insensitive.

- 'ciforth ERROR # 13 : ERROR, NO FURTHER INFORMATION'

  This error is used temporarily, whenever there is need for an error message but there is not yet one assigned.

- 'ciforth ERROR # 14 : SAVE/RESTORE MUST RUN FROM FLOPPY'
- 'ciforth ERROR # 15 : CANNOT FIND WORD TO BE POSTPONED'

  The word following POSTPONE must be postponed, but it can't be found in the search order.

- 'ciforth ERROR # 16 : CANNOT FIND WORD TO BE COMPILED'

  The word following [COMPILE] must be postponed, but it can't be found in the search order.

- 'ciforth ERROR # 17 : COMPILATION ONLY, USE IN DEFINITION'
- 'ciforth ERROR # 18 : EXECUTION ONLY'
- 'ciforth ERROR # 19 : CONDITIONALS NOT PAIRED'
- 'ciforth ERROR # 20 : STACK UNBALANCE, STRUCTURE UNFINISHED?'
- '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'
- 'ciforth ERROR # 23 : OFF CURRENT EDITING SCREEN'
- 'ciforth ERROR # 24 : (WARNING) NOT PRESENT, THOUGH WANTED' This error is reported by WANTED . The word you required, has been looked up in the index lines. It was not found

in the index lines, or it was a dummy item, that only marks the screen to be loaded, e.g. '`ASSEMBLER-GENERIC`'. In the latter case it can be safely ignored.

- '`ciforth ERROR # 25 : LIST EXPECTS DECIMAL`'

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

- '`ciforth ERROR # 48 : NO BUFFER COULD BE FREED, ALL LOCKED`'

  While a block is in use by `THRU` , it is *locked* , which means that it must stay in memory. In addition blocks can be locked explicitly by `LOCK` . If a free block is needed, and there is no block that can be written back to the mass storage (disk or flash), you get this error.

- '`ciforth ERROR # 49 : EXECUTION OF EXTERNAL PROGRAM FAILED`' The word `SYSTEM` detected an error while trying to execute an external program.

- '`ciforth ERROR # 50 : NOT ENOUGH MEMORY FOR ALLOCATE`' The dynamic memory allocation could not allocate a buffer of the size wanted, because there is not enough consecutive memory available. Fragmentation can cause this to happen while there is more than that size available in total. This is detected by `ALLOCATE` or `RESIZE` .

- '`ciforth ERROR # 51 : UNKNOWN FORMAT IDENTIFIER`' This error is detected by the `FORMAT` wordset. The word following `%` in a format string, is not known. This means that it is not present in the *namespace*

  `FORMAT-WID` .

See ⟨undefined⟩ [ASSEMBLER], page ⟨undefined⟩,, for errors generated by the assembler. In general these have numbers that are 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* 32 bits is meant. Wherever it says *double* or "double precision number" a 64 bits number is meant.

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

The symbols include:

'`addr`' memory address

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

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

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

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

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

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

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

'`sc`' a *string constant* , i.e. two cells, an address and a length; length characters are present at the address

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

'`u`' 32-bit unsigned integer

'`ud`' 64-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

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

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

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

‘U’ A user variable.

Where there is mention of a standard or a model, it means that the word actually complies to the standard or the model, not that some word of that name is present.

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

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

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

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

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

8.1 COMPILING

The wordset ‘COMPILING’ contains words that compile See Section 8.5.10 [IMMEDIATE], page 48, 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.

8.1.1 LITERAL

Name: LITERAL

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

Attributes: ISO,I,C2

Description: If compiling, then compile the stack value ‘n’ as a 32 bit 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: ‘LIT’ ‘LITERAL’

8.1.2 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]’

8.1.3 LIT

Name: LIT

Stackeffect: — n

Attributes: FIG,C2

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

See also: 'LITERAL'

8.2 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.

8.2.1 +LOOP

Name: +LOOP

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

Attributes: ISO,I,C2

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.

8.2.2 AGAIN

Name: AGAIN

Stackeffect: addr n — (compiling)

Attributes: ISO,FIG,I,C2

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'

8.2.3 BEGIN

Name: BEGIN

Stackeffect: — addr n (compiling)

Attributes: ISO,FIG,I

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

```
BEGIN ... UNTIL
```

```
BEGIN ... AGAIN
```

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

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

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

See also: '(BACK'

8.2.4 DO

Name: DO

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

Attributes: ISO,FIG,I,C2

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'

8.2.5 ELSE

Name: ELSE

Stackeffect: addr1 n1 — addr2 n2 (compiling)

Attributes: ISO,FIG,I,C2

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'

8.2.6 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: '(;)'

8.2.7 IF

Name: IF

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

Attributes: ISO,FIG,I,C2

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'

8.2.8 I

Name: I

Stackeffect: — n

Attributes: ISO,FIG,C

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

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

8.2.9 J

Name: J

Stackeffect: — n

Attributes: ISO,FIG,C

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

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

8.2.10 LOOP

Name: LOOP

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

Attributes: ISO,I,C2

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'

8.2.11 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'

8.2.12 THEN

Name: THEN

Stackeffect: addr n — (compile)

Attributes: ISO,FIG,I,CO

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'

8.2.13 UNTIL

Name: UNTIL

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

Attributes: ISO,FIG,I,C2

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'

8.2.14 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'

8.2.15 (+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'

8.2.16 (BACK

Name: (BACK

Stackeffect: — addr

Attributes:

Description: Start a backward branch by leaving the target address HERE into 'addr'.

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

8.2.17 (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'

8.2.18 (FORWARD

Name: (FORWARD

Stackeffect: — addr

Attributes:

Description: Start a forward branch by leaving the address that must be backpatched with an offset into 'addr'.

See also: 'IF'

8.2.19 (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'

8.2.20 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'

8.2.21 BACK)

Name: BACK)

Stackeffect: addr —

Attributes:

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

See also: 'LOOP' 'UNTIL'

8.2.22 BRANCH

Name: BRANCH

No stackeffect

Attributes: FIG,C2

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'

8.2.23 FORWARD)

Name: FORWARD)

Stackeffect: addr —

Attributes:

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

See also: 'LOOP' 'UNTIL'

8.3 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> .

8.3.1 :

Name: :

No stackeffect

Attributes: ISO,FIG,E

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

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

Creates a dictionary entry defining 'cccc' as equivalent to the following sequence of Forth word definitions '...' until the next ';' or ';CODE' . The word is added as the latest into the CONTEXT 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.

8.3.2 ;

Name: ;

No stackeffect

Attributes: ISO,FIG,I,C

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

See also: ':'

8.3.3 CONSTANT

Name: CONSTANT

Stackeffect: n —

Attributes: ISO,FIG

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

See also: 'VARIABLE' '>DFA'

8.3.4 CREATE

Name: CREATE

No stackeffect

Attributes: ISO

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

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

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

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

ciforth is byte aligned, so no extra measures are needed.

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

8.3.5 DOES>

Name: DOES>

No stackeffect

Attributes: ISO,FIG

Description: A word which is used in combination with CREATE

to specify the run-time action within a high-level defining word. DOES> modifies the default behaviour of the created word so as to execute the sequence of compiled word addresses following DOES> . When the DOES> part executes it begins with the address of the data field of the word on the stack. This allows interpretation using this area or its contents.

8.3.6 VARIABLE

Name: VARIABLE

No stackeffect

Attributes: ISO,FIG

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

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

8.3.7 (CREATE)

Name: (CREATE)

Stackeffect: sc —

Attributes:

Description: This is the basis for all defining words, including : and CONSTANT . It creates a header with name 'sc' in the dictionary and links it into the CONTEXT word list.

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

8.3.8 HEADER

Name: HEADER

Stackeffect: sc — dea

Attributes:

Description: Create a dictionary entry structure for the word 'sc' and returns its address into 'dea'.

The *code field addres* and *data field address* addres contain a same pointer, to the area owned by this header, i.e. immediately following the completed header . The flag and link fields are initialised to zero . The name 'sc' is layed down in the dictionary before the header and filled in into the name field.

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

8.3.9 LINK

Name: LINK

Stackeffect: dea wid —

Attributes:

Description: Links the Forth word represented by 'dea' into the wordlist represented by 'wid' as the latest entry.

See also: 'HEADER' '(CREATE)'

8.4 DENOTATIONS

The wordset 'DENOTATIONS' contains prefixes (mostly one letter words) that introduce a *denotation* , i.e. a generalisation of NUMBER . PREFIX turns the latest definition into a prefix, similar to IMMEDIATE . If a word starting with the prefix is looked up in the dictionary, the prefix is found and executed. Prefix words parse input and leave a constant (number, char or string) on the stack, or compile such constant, depending on STATE . For a kernel system it is guaranteed that they reside in the minimum search order wordlist, associated with the namespace ONLY . To make a distinction with the same words in other wordlists, the names of denotations are prepended with "Prefix_" in the documentation. Actual names consists of the one character following "Prefix". Apart from Prefix_0 , ONLY

contains entries for all hex digits 1...9 and A...F. Like NUMBER always did, all denotations behave identical in interpret and compile mode and they are not supposed to be postponed.

8.4.1 Prefix_"

Name: Prefix_"

Stackeffect: — sc

Attributes: CI

Description: Parse a " delimited string and leave it on the stack. A " can be embedded in a string by doubling it.

8.4.2 Prefix_&

Name: `Prefix_&`

Stackeffect: — c

Attributes: CI

Description: Leave '`c`' the non blank char that follows. Skip another blank character.

See also: '`^`'

8.4.3 Prefix_+

Name: `Prefix_+`

Stackeffect: — s/d

Attributes: CI

Description: Implements `NUMBER` for numbers that start with `+` .

8.4.4 Prefix_-

Name: `Prefix_-`

Stackeffect: — s/d

Attributes: CI

Description: Implements `NUMBER` for numbers that start with `-` .

8.4.5 Prefix__TICK

Name: `Prefix__TICK`

Stackeffect: — addr

Attributes: ISO,FIG,I

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`

instead.) Furthermore it is recommended that for non-portable code `'` is used in its *denotation* form without the space. .

See also: '`CONTEXT`' '`[']`' '`PRESENT`' '`>CFA`' '`>DFA`' '`>FFA`' '`>LFA`' '`>NFA`' '`>SFA`'

8.4.6 Prefix_

Name: `Prefix_`

Stackeffect: — s/d

Attributes: CI

Description: Implements `NUMBER` for numbers that start with nothing, i.e. anything.

8.5 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.

8.5.1 ,

Name: ,

Stackeffect: n —

Attributes: ISO,FIG

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

See also: ‘DP’ ‘C,’

8.5.2 >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>’

8.5.3 ALLOT

Name: ALLOT

Stackeffect: n —

Attributes: ISO,FIG

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

See also: ‘CELL+’

8.5.4 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’ ‘,’

8.5.5 DP

Name: DP

Stackeffect: —- addr

Attributes: FIG,U,L

Description: A 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 .

8.5.6 FORGET

Name: `FORGET`

No stackeffect

Attributes: ISO,FIG,E

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

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

8.5.7 FOUND

Name: `FOUND`

Stackeffect: sc — dea

Attributes:

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

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

8.5.8 HERE

Name: `HERE`

Stackeffect: — addr

Attributes: ISO,FIG

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

See also: '`DP`'

8.5.9 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`'

8.5.10 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` .

8.5.11 PAD

Name: `PAD`

Stackeffect: — addr

Attributes: ISO,FIG

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

8.5.12 PREFIX

Name: `PREFIX`

No stackeffect

Attributes:

Description: Mark the most recently made definition a *prefix* . If searching the wordlists for a name that starts with the prefix, the prefix is a match for that name. This method allows to define numbers, and other *denotations* such as strings, in a modular and extensible fashion. A prefix word finds the interpreter pointer pointing to the remainder of the name (or number) sought for, and must compile that remainder. Prefix words are mostly both immediate and *smart* , i.e. they behave differently when compiled, than interpreted. The result is that the compiled code looks the same and behaves the same than the interpreted code.

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

8.5.13 PRESENT

Name: `PRESENT`

Stackeffect: sc — dea

Attributes:

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

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

8.5.14 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`'

8.5.15 >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. .

See also: '`'`' '`CFA>`' '`>DFA`' '`>FFA`' '`>LFA`' '`>NFA`' '`>SFA`' '`>PHA`'

8.5.16 >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`’

8.5.17 >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`’

8.5.18 >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`’

8.5.19 >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`’

8.5.20 >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`’

8.5.21 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.

See also: ‘`IMMEDIATE`’ ‘`RECURSE`’

8.5.22 ~MATCH

Name: ~MATCH

Stackeffect: sc dea — sc dea n

Attributes:

Description: Intended to cooperate with (FIND) . Compares the *string constant* 'sc' with the dea 'dea'. Returns into 'n' the difference between the first characters that compare unequal, or zero if the strings are the same up to the smallest length. It is required that the dea has a non-zero name field.

See also: 'FIND' 'CORA'

8.6 DOUBLE

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

8.7 ERRORS

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

8.7.1 ?ERROR

Name: ?ERROR

Stackeffect: f n —

Attributes:

Description:

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

See also: 'ERROR' '?ERRUR'

8.7.2 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. Since the stack depth is known, the application may DROP those items.

See also: 'THROW' 'QUIT' 'HANDLER'

8.7.3 ERROR

Name: ERROR

Stackeffect: n —

Attributes:

Description: Notify the user that an uncaught exception or error with number 'n' has occurred. The word that caused it is found using WHERE and displayed . Also 'n' is passed to MESSAGE in order to give a description of the error, clamped to the range [-256, 63]. This word is executed by THROW before restarting the interpreter and can be revectored to give more elaborate diagnostics.

See also: 'MESSAGE' '?ERROR' 'WARNING'

8.7.4 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'

8.7.5 WHERE

Name: WHERE

Stackeffect: — addr

Attributes: U

Description: A variable pair which contains the start of the source and the position after the last character parsed when an error thrown by ?ERROR . The contents of WHERE is interpreted by ERROR if the corresponding exception was never caught.

See also: 'THROW' 'CATCH '

8.7.6 HANDLER

Name: HANDLER

Stackeffect: — addr

Attributes:

Description: A 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'

8.8 FILES

The wordset 'FILES' contains words to input and output to files, or load words from files. They are underlying the 'BLOCKS' facilities.

8.8.1 CLOSE-FILE

Name: CLOSE-FILE

Stackeffect: fileid — ior

Attributes: ISO

Description: Close the file with file handle in 'fileid'. Return a result code into 'ior'. The latter is the Linux error code negated, to be inspected using MESSAGE .

See also: 'OPEN-FILE' 'READ-FILE' 'WRITE-FILE' 'CREATE-FILE' 'DELETE-FILE'

8.8.2 CREATE-FILE

Name: CREATE-FILE

Stackeffect: sc u — fileid ior

Attributes: NISO

Description:

. Create a file with name ‘sc’ and file access privileges ‘u’ If the file already exists, it is truncated to zero length. Return a file handle into ‘fileid’ and a result code into ‘ior’. The latter is the ‘errno’ negated, to be inspected using MESSAGE . The handle is open for READ_WRITE.

See also: ‘OPEN-FILE’ ‘READ-FILE’ ‘WRITE-FILE’ ‘CREATE-FILE’ ‘DELETE-FILE’

8.8.3 DELETE-FILE

Name: DELETE-FILE

Stackeffect: sc — ior

Attributes: ISO

Description: Delete the file with name ‘sc’. Return a result code into ‘ior’. The latter is the ‘errno’ negated, to be inspected using MESSAGE .

See also: ‘OPEN-FILE’ ‘READ-FILE’ ‘WRITE-FILE’ ‘CREATE-FILE’ ‘DELETE-FILE’

8.8.4 GET-FILE

Name: GET-FILE

Stackeffect: sc1 — sc2

Attributes:

Description: Get the content of the file with name ‘sc1’; leave it as a string ‘sc2’. Any errors are thrown.

See also: ‘PUT-FILE’ ‘OPEN-FILE’ ‘THROW’

8.8.5 INCLUDE

Name: INCLUDE

Stackeffect: "" — i*x

Attributes: ISO

Description: Interpret the content of the file with name from the input stream as if it was typed from the console, leaving result ‘i*x’.

See also: ‘LOAD’

8.8.6 OPEN-FILE

Name: OPEN-FILE

Stackeffect: sc fam — fileid ior

Attributes: ISO

Description: Open the file with name ‘sc’ and file access method ‘fam’. Return a file handle into ‘fileid’ and a result code into ‘ior’. The latter is the ‘errno’ error code negated, to be inspected using MESSAGE . ‘fam’ is one of 0=READ_ONLY, 1=WRITE_ONLY, 2=READ_WRITE.

See also: ‘READ-FILE’ ‘WRITE-FILE’ ‘CREATE-FILE’ ‘DELETE-FILE’

8.8.7 PUT-FILE

Name: PUT-FILE

Stackeffect: sc1 sc2 —

Attributes:

Description: Save the *string constant* ‘sc1’ to a file with the name ‘sc2’. Any errors are thrown.

See also: ‘GET-FILE’ ‘OPEN-FILE’ ‘THROW’

8.8.8 READ-FILE

Name: READ-FILE

Stackeffect: addr n1 fd — n2 ior

Attributes: ISO

Description: Read 'n' characters to 'addr' from current position of the file that is open at 'fd' . 'n2' is the number of characters successfully read, this may be zero. 'ior' is 0 for success, or otherwise 'errno' error code negated, to be inspected using MESSAGE .

See also: 'OPEN-FILE' 'WRITE-FILE' 'REPOSITION-FILE' 'BLOCK-READ'

8.8.9 WRITE-FILE

Name: WRITE-FILE

Stackeffect: addr n fd — u1

Attributes: ISO

Description: Write 'n' characters from 'addr' to the file that is open at 'fd' , starting at its current position. 'u1' is 0 for success, or otherwise 'errno' error code negated, to be inspected using MESSAGE .

See also: 'OPEN-FILE' 'READ-FILE' 'REPOSITION-FILE' 'BLOCK-WRITE'

8.8.10 RW-BUFFER

Name: RW-BUFFER

Stackeffect: — addr

Attributes:

Description: A constant that leaves the address of a disk buffer used by file i/o words.

See also: 'READ-FILE' 'OPEN-FILE'

8.9 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 8.6 [DOUBLE], page 51, wordset. See Section 8.16 [MULTIPLYING], page 67, wordset.

8.9.1 %>

Name: %>

Stackeffect: d — sc

Attributes: ISO,FIG

Description: Terminates numeric output conversion by dropping 'd', leaving the formatted string 'sc' .

See also: '<%'

8.9.2 %S

Name: %S

Stackeffect: n1 — n2

Attributes: ISO,FIG

Description: Generates ASCII text in the text output buffer, by the use of % , until a zero number 'n2' results. Used between <% and %> .

8.9.3 %

Name: %

Stackeffect: n1 — n2

Attributes: ISO,FIG

Description: Generate from a number ‘n1’, the next ASCII character which is placed in an output string. Result ‘n2’ is the quotient after division by BASE , and is maintained for further processing. Used between <% and %> .

See also: ‘%S’

8.9.4 <%

Name: <%

No stackeffect

Attributes: ISO,FIG

Description: Setup for pictured numeric output formatting using the words: <% % %S SIGN %> The conversion is done on a double number producing text growing down from PAD .

See also:

8.9.5 BASE

Name: BASE

Stackeffect: — addr

Attributes: ISO,FIG,U

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

See also: ‘DECIMAL’ ‘HEX’ ‘<%’

8.9.6 DECIMAL

Name: DECIMAL

No stackeffect

Attributes: ISO,FIG

Description: Set the numeric conversion BASE for decimal input-output.

See also: ‘HEX’

8.9.7 HEX

Name: HEX

No stackeffect

Attributes: ISO,FIG

Description: Set the numeric conversion BASE for hexadecimal (base 16) input-output.

See also: ‘DECIMAL’

8.9.8 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’

8.9.9 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'

8.9.10 DIGIT

Name: DIGIT

Stackeffect: c — n2 false (ok) c — x true (bad)

Attributes:

Description: Converts the ASCII character 'c' (using the current BASE) 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.

8.9.11 HLD

Name: HLD

Stackeffect: — addr

Attributes: FIG

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

See also: '<%'

8.9.12 NUMBER

Name: NUMBER

Stackeffect: — n1

Attributes:

Description: Convert the ASCII text at the *current input source* with regard to BASE . The new value is accumulated into double number 'n1' , 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 INITIALISATIONS

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

8.10.1 ABORT

Name: ABORT

No stackeffect

Attributes: ISO,FIG

Description: Restart the interpreter. This is a patch point to replace the actions of QUIT , into a turnkey.

See also: 'QUIT'

8.10.2 CLS

Name: CLS

No stackeffect

Attributes: ISO,FIG

Description:

Clear the data stack.

See also: 'WARM' 'INIT'

8.10.3 COLD

Name: COLD

No stackeffect

Attributes: FIG

Description: Initialise the stacks Initialise the system as per INIT . Handle command line options, if any. Show signon message and restart via ABORT . May be called from the terminal to remove application programs and restart, as long as there are no new vocabularies with definitions. But it is better to say BYE to Forth and start again.

See also: 'WARM' 'INIT' 'OPTIONS'

8.10.4 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. .

See also: 'WARM' 'COLD' 'FORTH' 'BLOCK-INIT' 'BASE' 'DSP0'

8.10.5 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'

8.10.6 QUIT

Name: QUIT

No stackeffect

Attributes: ISO,FIG

Description: Restart the interpreter. Clear the return stack, stop compilation, and return control to the operators terminal, or to the redirected input stream. This means (ACCEPT) user input to somewhere in the terminal input buffer, and then INTERPRET with that as a SOURCE . that is still busy or that has aborted. Reset the exception mechanism.

No message is given.

See also: 'CIB' 'ABORT'

8.11 INPUT

The wordset 'INPUT' contains words to get input from the terminal and such. See Section 8.8 [FILES], page 52, for disk I/O. See ⟨undefined⟩ [BLOCKS], page ⟨undefined⟩, for access of blocks.

8.11.1 (ACCEPT)

Name: (ACCEPT)

Stackeffect: — sc

Attributes:

Description: Accept characters from the terminal, until a *RET* is received and return the result as a constant string 'sc'. It doesn't contain any line ending, but the buffer still does and after 1+ the string ends in a *LF*. The editing functions are the same as with ACCEPT .

See also: 'KEY' 'KEY?' 'ACCEPT'

8.11.2 ACCEPT

Name: ACCEPT

Stackeffect: addr count — n

Attributes: ISO

Description: Transfer at most 'count' characters from the terminal to address, until a *RET* is received. The simple tty editing functions of Linux are observed, i.e. the "erase" (delete a character) and "kill" (delete a line) characters. Typically these are the backspace key and ^U. Note that excess characters after 'count' are ignored. The number of characters not including the *RET* is returned into 'n'.

See also: '(ACCEPT)' 'KEY' 'KEY?' '(ACCEPT)'

8.11.3 CIB

Name: CIB

Stackeffect: — addr

Attributes: ISO,FIG,U

Description: A variable containing the address of the terminal input buffer. this was used for file i/o too

See also: 'QUIT'

8.11.4 KEY

Name: KEY

Stackeffect: — c

Attributes: ISO,FIG

Description: Leave the ASCII value of the next terminal key struck.

See also: 'ACCEPT' 'KEY?'

8.11.5 PP

Name: PP

Stackeffect: — addr

Attributes:

Description: A variable containing a pointer within the current input text buffer (terminal or disc) from which the next text will be accepted. All parsing words use and move the value of PP . This is not conforming to ISO, in the sense that clearing it doesn't reset the parse pointer to the start of the current line.

See also: '>IN' 'WORD' 'NAME' 'NUMBER' 'PARSE' 'PP@@'

8.11.6 REFILL-CIB

Name: REFILL-CIB

Stackeffect: —

Attributes:

Description: Accept characters from the terminal input stream such as to fill up CIB . Normally this means until a *RET*. It is now consumable by ACCEPT or after SET-SRC

by Forth parsing words like WORD . The editing functions are those described by ACCEPT . Immediately, after REFILL-CIB 'REMAINDER 2@' defines the characters ready in the input buffer. All characters are retained including the *RET*.

If the input is redirected (such that reading after a *RET* occurs) 'REMAINDER 2@' contains the part of CIB that is not yet consumed by (ACCEPT) , and outside the reach of SRC .

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

See also: 'ACCEPT' '(ACCEPT)'

8.11.7 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-CIB'

8.12 JUGGLING

The wordset 'JUGGLING' contains words that change order of data on the *data stack* . The necessity for this arise, because the data you want to feed to a Forth word is not directly accessible, i.e. on top of the stack. It is also possible that you need the same data twice, because you have to feed it to two different words. Design your word such that you need them as little as possible, because they are confusing.

8.12.1 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'

8.12.2 2DUP

Name: 2DUP

Stackeffect: d — d d

Attributes: ISO

Description: Duplicate the double number on the stack.

See also: 'OVER'

8.12.3 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`'

8.12.4 2SWAP

Name: `2SWAP`

Stackeffect: d1 d2 — d2 d1

Attributes: ISO

Description: Exchange the top doubles on the stack.

See also: '`ROT`'

8.12.5 DROP

Name: `DROP`

Stackeffect: n —

Attributes: ISO,FIG

Description: Drop the number from the stack.

See also: '`DUP`'

8.12.6 DUP

Name: `DUP`

Stackeffect: n — n n

Attributes: ISO,FIG

Description: Duplicate the value on the stack.

See also: '`OVER`'

8.12.7 NIP

Name: `NIP`

Stackeffect: n1 n2—n2

Attributes: ISO

Description: Drop the second number from the stack.

See also: '`DUP`' '`DROP`'

8.12.8 OVER

Name: `OVER`

Stackeffect: nl n2 — nl n2 n1

Attributes: ISO,FIG

Description: Copy the second stack value, placing it as the new top.

See also: '`DUP`'

8.12.9 SDSWAP

Name: SDSWAP

Stackeffect: nl n2 n3 — n2 n3 nl

Attributes: ISO,FIG

Description: Rotate the top three values on the stack, bringing the third to the top.

See also: 'SWAP'

8.12.10 SWAP

Name: SWAP

Stackeffect: n1 n2 — n2 n1

Attributes: ISO,FIG

Description: Exchange the top two values on the stack.

See also: 'ROT'

8.13 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 logical operators (AND etc.) work on all 32 bits, one by one. In this way they are useful for mask operations, as well as for combining conditions represented as flag's. But beware that IF only cares whether the top of the stack is non-zero, such that - can mean non-equal to IF . Such conditions (often named just *flag* 's) cannot be directly combined using logical operators, but '0= 0=' can help.

8.13.1 0<

Name: 0<

Stackeffect: n — ff

Attributes: ISO,FIG

Description: Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

See also: '<'

8.13.2 0=

Name: 0=

Stackeffect: n — ff

Attributes: ISO,FIG

Description: Leave a true flag 'ff' is the number 'n' is equal to zero, otherwise leave a false flag.

See also: '='

8.13.3 <>

Name: <>

Stackeffect: n1 n2 — ff

Attributes: ISO

Description: Leave a true flag if 'n1' is not equal than 'n2' ; otherwise leave a false flag.

See also: '>' '=' '0<'

8.13.4 <

Name: <

Stackeffect: n1 n2 — ff

Attributes: ISO

Description: Leave a true flag if ‘n1’ is less than ‘n2’ ; otherwise leave a false flag.

See also: ‘=’ ‘>’ ‘0<’

8.13.5 =

Name: =

Stackeffect: n1 n2 — ff

Attributes: ISO,FIG

Description: Leave a true flag if ‘n1=n2’ ; otherwise leave a false flag.

See also: ‘<’ ‘>’ ‘0=’ ‘-’

8.13.6 >

Name: >

Stackeffect: n1 n2 — ff

Attributes: ISO

Description: Leave a true flag if ‘n1’ is greater than ‘n2’ ; otherwise leave a false flag.

See also: ‘<’ ‘=’ ‘0<’

8.13.7 AND

Name: AND

Stackeffect: n1 n2 — n3

Attributes: ISO,FIG

Description: Leave the bitwise logical and of ‘n1’ and ‘n2’ as ‘n3’ .

See also: ‘XOR’ ‘OR’

8.13.8 INVERT

Name: INVERT

Stackeffect: n1 — n2

Attributes: ISO

Description: Invert all bits of ‘n1’ leaving ‘n2’ . For pure flags (0 or -1) this is the *logical not* operator.

See also: ‘AND’ ‘OR’

8.13.9 NOT

Name: NOT

Stackeffect: ff — ff1

Attributes: ISO,FIG

Description: Invert the flag ‘ff’ leaving the inverted flag ‘ff1’.

See also: ‘=’

8.13.10 OR

Name: `OR`

Stackeffect: n1 n2 — n3

Attributes: ISO,FIG

Description: Leave the bit-wise logical or of two 32-bit values.

See also: '`AND`' '`XOR`'

8.13.11 XOR

Name: `XOR`

Stackeffect: nl n2 — n3

Attributes: ISO,FIG

Description: Leave the bitwise logical exclusive or of two 32-bit values.

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

8.14 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.

8.14.1 !

Name: `!`

Stackeffect: n addr —

Attributes: ISO,FIG

Description: Store all 32 bits of n at '`addr`' .

See also: '`@`' '`C!`' '`2!`' '`FAR!`' '`PW!`' '`PC!`'

8.14.2 +!

Name: `+!`

Stackeffect: n addr —

Attributes: ISO,FIG

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

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

8.14.3 2!

Name: `2!`

Stackeffect: x1 x2 addr —

Attributes: ISO

Description: Store a pair of 32 bits values '`x1`' '`x2`' to consecutive cells at '`addr`' . '`x2`' is stored at the lowest address.

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

8.14.4 2@

Name: 2@

Stackeffect: addr— x1 x2

Attributes: ISO

Description: Leave a pair of 32 bits values ‘x1’ ‘x2’ from consecutive cells at ‘addr’ . ‘x2’ is fetched from the lowest address.

See also: ‘2!’ ‘@’ ‘C@’

8.14.5 @

Name: @

Stackeffect: addr — n

Attributes: ISO,FIG

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

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

8.14.6 ALIGNED

Name: ALIGNED

Stackeffect: addr1 — addr2

Attributes: ISO

Description: Make sure that ‘addr1’ is *aligned* by advancing it if necessary to ‘addr2’.

See also: ‘ALIGN’

8.14.7 ALIGN

Name: ALIGN

Stackeffect: —

Attributes: ISO

Description: Make sure that HERE is *aligned* by advancing it if necessary. This means that data of any size can be fetched from that address efficiently.

See also: ‘ALIGNED’

8.14.8 BM

Name: BM

Stackeffect: —- addr

Attributes:

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

See also: ‘DP’ ‘EM’

8.14.9 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@’ ‘!’

8.14.10 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@'

8.14.11 CELL+

Name: CELL+

Stackeffect: n1 — n2

Attributes: ISO

Description: Advance the memory pointer 'n1' by one (in this case 32 bits) cell to 'n2'.

8.14.12 CELLS

Name: CELLS

Stackeffect: n1 — n2

Attributes: ISO

Description: Return the equivalent of 'n1' cells in bytes: 'n2'.

See also: 'CELL+'

8.14.13 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.

8.14.14 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'

8.14.15 FILL

Name: FILL

Stackeffect: addr u b —

Attributes: ISO,FIG

Description: If 'u' is not zero, store 'b' in each of 'u' consecutive bytes of memory beginning at 'addr' .

See also: 'BLANK' 'ERASE'

8.14.16 MOVE

Name: `MOVE`

Stackeffect: from to count —

Attributes:

Description: Move '`count`' bytes beginning at address '`from`' to address '`to`', such that the destination area contains what the source area contained, regardless of overlaps. As the Intel 86-family is byte-addressing there are no restrictions.

8.14.17 TOGGLE

Name: `TOGGLE`

Stackeffect: addr b —

Attributes: NFIG

Description: Complement the contents of '`addr`' by the bit pattern '`b`' .

See also: '`XOR`' '`+! `'

8.15 MISC

The wordset '`MISC`' contains words that defy categorisation.

8.15.1 EXECUTE

Name: `EXECUTE`

Stackeffect: xt —

Attributes: ISO,FIG

Description: Execute the definition whose *execution token* is given by '`xt`' .

See also: '`'`' '`>CFA`'

8.15.2 NOOP

Name: `NOOP`

No stackeffect

Attributes:

Description: Do nothing. Primarily useful as a placeholder.

8.15.3 TASK

Name: `TASK`

No stackeffect

Attributes:

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

See also: '`COLD`'

8.15.4 _

Name: `_`

Stackeffect: — x

Attributes:

Description: Leave an undefined value '`x`'.

8.16 MULTIPLYING

The original 16 bits Forth's have problems with overflow (see Section 8.18 [OPERATOR], page 68). Operators with intermediate results of double precision, mostly scaling operators, solve this and are present in the '`MULTIPLYING`' wordset. In this 32 bit Forth you will have less need. Formatting is done with *double* 's exclusively, and relies on this wordset. Operators with mixed precision and unsigned operators allow to build arbitrary precision operators from them in *high level* code.

8.16.1 */MOD

Name: `*/MOD`

Stackeffect: n1 n2 n3 — n4 n5

Attributes: ISO,FIG

Description: Leave the quotient '`n5`' and remainder '`n4`' of the operation '`n1*n2/n3`' (using *symmetric division*). A double precision intermediate product is used giving correct results, unless '`n4`' or '`n5`' overflows.

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

8.16.2 */

Name: `*/`

Stackeffect: n1 n2 n3 — n4

Attributes: ISO,FIG

Description: Leave the ratio '`n4 = n1*n2/n3`' where all are signed numbers(using *symmetric division*). A double precision intermediate product is used giving correct results, unless '`n4`' overflows.

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

8.16.3 M*

Name: `M*`

Stackeffect: n1 n2 — d

Attributes: ISO,FIG

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

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

8.16.4 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

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

8.17 OPERATINGSYSTEM

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

8.17.1 XOS

Name: `XOS`

Stackeffect: n1 n2 n3 n—ret

Attributes:

Description: Do a traditional Unix type system call '`n`' (man 2) with parameters '`n1 n2 n3`'. '`ret`' is the return value of the call. If it is negative, it is mostly an error, such as known by **errno** . This makes available all facilities present in Linux, except those with 4 or 5 parameters that are handled by `XOS5` .

See also: '`?ERRUR`'

8.17.2 ZEN

Name: `ZEN`

Stackeffect: sc — addr

Attributes:

Description: Leaves an address that contains a zero-ended (c-type) equivalent of '`sc`'. The same buffer is reused, such that this word is not reentrant.

See also: '`OPEN-FILE`' '`XOS`' '`XOS5`'

8.18 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* .

Divisions involving negative numbers have an interpretation problem. In any case we want the combination of `/` and `MOD` (remainder) to be such that you can get the original '`n`' back from the two values left by '`n m MOD`' and '`n m /`' by performing '`m * +`' . This is true for all Forth's. On ciforth the `/` is a *symmetric division* , i.e. '`-n m /`' give the same result as '`n m /`', but negated. The foregoing rule now has the consequence that '`m MOD`' has '`2|m|-1`' possible outcomes instead of '`|m|`' . This is very worrisome for mathematicians, who stick to the rule that '`m MOD`' has '`|m|`' outcomes: '`0 ... |m|-1`', or '`-|m|+1 ... 0` ' for negative numbers. (*floored division*).

8.18.1 *

Name: `*`

Stackeffect: n1 n2 — n3

Attributes: ISO,FIG

Description: Leave the signed product '`n3`' of two signed numbers '`n1`' and '`n2`' .

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

8.18.2 +

Name: `+`

Stackeffect: n1 n2 — sum

Attributes: ISO,FIG

Description: Leave the sum of '`n1`' and '`n2`' .

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

8.18.3 -

Name: -

Stackeffect: n1 n2 — diff

Attributes: ISO,FIG

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

See also: ‘NEGATE’ ‘+’ ‘*’ ‘/’ ‘MOD’

8.18.4 /MOD

Name: /MOD

Stackeffect: n1 n2 — rem quot

Attributes: ISO,FIG

Description: Leave the remainder and signed quotient of ‘n1’ and ‘n2’ . The remainder has the sign of the dividend (i.e. *symmetric division*).

See also: ‘*/MOD’ ‘*/’ ‘SM/REM’

8.18.5 /

Name: /

Stackeffect: n1 n2 — quot

Attributes: ISO,FIG

Description: Leave the signed quotient of ‘n1’ and ‘n2’ . (using *symmetric division*).

See also: ‘+’ ‘-’ ‘*’ ‘MOD’ ‘*/MOD’

8.18.6 ABS

Name: ABS

Stackeffect: n — u

Attributes: ISO,FIG

Description: Leave the absolute value of ‘n’ as ‘u’ .

See also: ‘DABS’

8.18.7 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.

See also: ‘RSHIFT’ ‘2*’

8.18.8 MAX

Name: MAX

Stackeffect: n1 n2 — max

Attributes: ISO,FIG

Description: Leave the greater of two numbers.

See also: ‘MIN’

8.18.9 MIN

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

8.18.10 MOD

Name: `MOD`
Stackeffect: n1 n2 — mod
Attributes: ISO,FIG
Description: Leave the remainder of '`n1`' divided by '`n2`' , with the same sign as '`n1`' (i.e. *symmetric division*).
See also: '`+`' '`-`' '`*`' '`/`' '`MOD`' '`*/MOD`'

8.18.11 NEGATE

Name: `NEGATE`
Stackeffect: n1 — n2
Attributes: ISO,FIG
Description: Leave the two's complement of a number, i.e. '`n2`' is '`-n1`'
See also: '`-`'

8.18.12 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.
See also: '`LSHIFT`' '`2/`'

8.19 OUTPUT

The wordset '`OUTPUT`' contains words to output to the terminal and such. See Section 8.8 [FILES], page 52, for disk I/O. See ⟨undefined⟩ [BLOCKS], page ⟨undefined⟩, for blocks.

8.19.1 .

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

8.19.2 CR

Name: `CR`
No stackeffect
Attributes: ISO,FIG
Description: Transmit character(s) to the terminal, that result in a "carriage return" and a "line feed".
See also: '`OUT`'

8.19.3 EMIT

Name: EMIT

Stackeffect: c —

Attributes: ISO,FIG

Description: Transmit ASCII character 'c' to the output device. For this ciforth all terminal I/O goes through TYPE . In this ciforth EMIT maintains OUT .

See also: 'TYPE' 'OUT'

8.19.4 SPACE

Name: SPACE

No stackeffect

Attributes: ISO,FIG

Description: Transmit an ASCII blank to the output device.

See also: 'EMIT' 'OUT'

8.19.5 TYPE

Name: TYPE

Stackeffect: addr count —

Attributes: ISO,FIG

Description: Transmit 'count' characters from 'addr' to the output device. All terminal I/O goes through this word. It is high level so terminal I/O can be redirected, by *revectoring* it . In this ciforth strings may contain embedded *LF* 's with the effect of a new line at that point in the output, however in that case OUT is not observed.

See also: 'EMIT' 'OUT' 'ETYPE'

8.19.6 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)'

8.20 PARSING

The *outer interpreter* is responsible for parsing, i.e. it gets a word from the *current input source* and interprets or compiles it, advancing the PP pointer. The wordset 'PARSING' contains the words used by this interpreter and other words that consume characters from the input source. In this way the outer interpreter need not be very smart, because its capabilities can be extended by new words based on those building blocks.

8.20.1 ?BLANK

Name: ?BLANK

Stackeffect: c — ff

Attributes:

Description: For the character 'c' return whether this is considered to be white space into the flag 'ff' . At least the space, ASCII null, the tab and the carriage return and line feed characters are white space.

See also: 'BL' 'SPACE'

8.20.2 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'

8.20.3 INTERPRET

Name: INTERPRET

Stackeffect: ?? — ??

Attributes:

Description: Repeatedly fetch the next text word from the *current input source* and execute it (STATE is not 1) or compile it (STATE is 1). A word is blank-delimited and looked up in the vocabularies of *search-order* . It can be either matched exactly, or it can match a prefix. A word that matches a prefix is called a *denotation* ; mostly this is a number. Prefixes are present in ONLY which is the last wordlist in the search order, and the minimum search order. If it is not found at all, it is an ERROR . A *denotation* is a number, a double number, a character or a string etc. Denotations are handled respectively by the words 0 ... F & " and any other word of the ONLY wordlist, depending on the first character or characters.

See also: 'WORD' 'NUMBER'

8.20.4 NAME

Name: NAME

Stackeffect: — sc

Attributes: CI

Description: Parse the *current input source* for a word, i.e. blank-delimited as per ?BLANK . Skip leading delimiters then advance the input pointer to past the next delimiter or past the end of the input source. Leave the word found as a string constant 'sc'.

See also: 'WORD' 'PP'

8.20.5 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.

See also: 'WORD' 'NAME'

8.20.6 PP@@

Name: PP@@

Stackeffect: —addr c

Attributes: CI

Description: Parse the *current input source* leaving the next character 'c' and its address 'addr' . If at the end of the input source, leave a pointer past the end and a zero. Advance the input pointer to the next character.

See also: 'WORD' 'PP'

8.20.7 RESTORE

Name: RESTORE

Stackeffect: —

Attributes:

Description: This must follow a SAVE in the same definition. Restore the content of SRC from the return stack thus restoring the *current input source* to what it was when the SAVE was executed.

See also: 'SET-SRC' 'RESTORE-INPUT'

8.20.8 SAVE

Name: SAVE

Stackeffect: —

Attributes:

Description: Save the content of SRC on the return stack to prepare for changing the *current input source* . This must be balanced by a RESTORE in the same definition. CO can be used between the two.

See also: 'SET-SRC' 'SAVE-INPUT'

8.20.9 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 .

See also: 'SRC' 'EVALUATE' 'INTERPRET'

8.20.10 SRC

Name: SRC

Stackeffect: addr —

Attributes:

Description: Return the address 'addr' of the *current input source* specification. It consists of three cells: the lowest address of the parse area, and the non-inclusive highest address of the parse area and a pointer to the next character to be parsed. Changing 'SRC' takes immediate effect, and must be atomic, by changing only the third cell or by e.g. using SET-SRC . The third cell has the alias 'PP' .

See also: 'SAVE' 'RESTORE'

8.20.11 STATE

Name: STATE

Stackeffect: — addr

Attributes: ISO,U

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

8.20.12 [

Name: [

No stackeffect

Attributes: ISO,FIG,I,

Description: Used in a colon-definition in form:

```
:  xxx    [  words   ]    more   ;
```

Suspend compilation. The words after [are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with]

See also: 'LITERAL ' ']'

8.20.13 \

Name: \

No stackeffect

Attributes: ISO,I

Description: Used in the form: '\ cccc' Ignore a comment that will be delimited by the end of the current line. May occur during execution or in a colon-definition. Blank space after the word \ is required.

See also: '('

8.20.14]

Name:]

No stackeffect

Attributes: ISO,FIG

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

See also: '['

8.21 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. You only need to know them if you want to extend the 'CONTROL' wordset. This version of ciforth contains no security, so the remainder of the chapter is empty.

8.21.1 ?STACK

Name: ?STACK

No stackeffect

Attributes:

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

See also: '?ERROR'

8.22 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.

8.22.1 .S

Name: .S

Stackeffect: from to —

Attributes:

Description: Print the stack, in the current base. For stack underflow print nothing.

See also: ‘LIST’

8.22.2 >R

Name: >R

Stackeffect: n —

Attributes: ISO,FIG,C

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

See also: ‘R@’

8.22.3 DEPTH

Name: DEPTH

Stackeffect: — n1

Attributes: ISO,WANT

Description: Leave into ‘n1’ the number of items on the data stack, before ‘n1’ was pushed.

See also: ‘DSP@’

8.22.4 DSP!

Name: DSP!

Stackeffect: addr —

Attributes:

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

See also: ‘DSP@’

8.22.5 DSP0

Name: DSP0

Stackeffect: — addr

Attributes: U

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

See also: ‘DSP!’

8.22.6 DSP@

Name: DSP@

Stackeffect: — addr

Attributes:

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

See also: ‘DSP0’ ‘DSP!’

8.22.7 R>

Name: R>

Stackeffect: — n

Attributes: ISO,FIG

Description: Remove the top value from the *return stack* and leave it on the *data stack* .

See also: ‘>R’ ‘R@’

8.22.8 R@

Name: R@

Stackeffect: — n

Attributes: ISO

Description: Copy the top of the return stack to the data stack.

See also: ‘>R’ ‘<R’

8.22.9 RSP!

Name: RSP!

Stackeffect: addr —

Attributes:

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

See also: ‘RSP@’

8.22.10 RSP0

Name: RSP0

Stackeffect: — addr

Attributes: U

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

See also: ‘RSP!’

8.22.11 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: ‘RSP0’ ‘RSP!’

8.23 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 .

8.23.1 $!

Name: $!

Stackeffect: sc addr —

Attributes:

Description: Store a *string constant* ‘sc’ in the string variable at address ‘addr’.

See also: ‘$@’ ‘$+!’ ‘$C+’

8.23.2 $+!

Name: $+!

Stackeffect: sc addr —

Attributes:

Description: Append a *string constant* ‘sc’ to the string variable at address ‘addr’.

See also: ‘$@’ ‘$!’ ‘$C+’

8.23.3 $,

Name: $,

Stackeffect: sc — addr

Attributes:

Description: Allocate and store a *string constant* ‘sc’ in the dictionary and leave its address ‘addr’.

See also: ‘$@’ ‘$!’

8.23.4 $/

Name: $/

Stackeffect: sc c — sc1 sc2

Attributes:

Description: Find the first ‘c’ in the *string constant* ‘sc’ and split it at that address. Return the strings after and before ‘c’ into ‘sc1’ and ‘sc2’ respectively. If the character is not present ‘sc1’ is a null string and ‘sc2’ is the original string.

See also: ‘$^’ ‘CORA’ ‘$@’

8.23.5 $@

Name: $@

Stackeffect: addr — sc

Attributes:

Description: From address ‘addr’ fetch a *string constant* ‘sc’ .

See also: ‘$!’ ‘$+!’ ‘$C+’

8.23.6 $C+

Name: `$C+`

Stackeffect: c addr —

Attributes:

Description: Append a char '`c`' to the string variable at address '`addr`'.

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

8.23.7 $^

Name: `$^`

Stackeffect: sc c — addr

Attributes:

Description: Find the first '`c`' in the *string constant* '`sc`' and return its '`addr`' if present. Otherwise return a *nil pointer*. A null string (0 0) or an empty string are allowed, and result in not found.

See also: '`$/`' '`CORA`' '`$@`'

8.23.8 BL

Name: `BL`

Stackeffect: — c

Attributes: ISO.FIG

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

8.23.9 COUNT

Name: `COUNT`

Stackeffect: addr1 — addr2 n

Attributes: ISO,FIG

Description: Leave the byte address '`addr2`' and byte count '`n`' of a message text beginning at address '`addr1`' . It is presumed that the first byte at '`addr1`' contains the text byte count and the actual text starts with the second byte. Alternatively stated, fetch a *string constant* '`addr2 n`' from the brain damaged string variable at '`addr1`' .

See also: '`TYPE`'

8.24 SUPERFLUOUS

The wordset '`SUPERFLUOUS`' contains words that are superfluous, because they are equivalent to small sequences of code.

8.24.1 0

Name: `0`

Stackeffect: — 0

Attributes:

Description: Leave the number 0.

See also: '`CONSTANT`'

8.24.2 1+

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

8.24.3 Number_1

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

Glossary Index

This index finds the glossary description of each word.

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

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Concept Index

Mostly the first reference is where the concept is explained. But sometimes in introductory and tutorial sections an explanation sometimes was considered too distracting.

T

U

V

W

Short Contents

Table of Contents