ciforth Manual

A system to generate a ciforth together with its documentation.

Albert van der Horst
Dutch Forth Workshop

1 Overview

'`ci86.gnr`' is a system to generate ciforth in all its configuration's. This is a configurators manual. For each ciforth there is a corresponding documentation; there is however just this one documentation for the generic system. It is assumed that you are familiar with Forth and with ciforth in particular. Linux is used for a development system, and the main tool is `m4` , the macro preprocessor. This generates an assembler source file and a raw documentation file out of the single generic source, controlled by a configuration file. In addition there is a file with blocks, that is common to all Forth's.

For further processing you need an assembler, such as `nasm` and one or more documentation tools, such as `info`. The raw documentation file can be ordered only by a more sophisticated tool than the usual `sort`, called `ssort`. The particulars of all this depend on the actual configuration chosen. It contains in itself some provision such that it can be loaded on any of the ciforth systems, independently of whether they are 16 or 32 bits or whether they can or cannot handle direct access of the video memory.

2 Non-technical background.

2.1 Legalese

The Forth's called ciforth are made available through 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.

This '`ci86.gnr`' system is protected by GPL. This applies to the generic source, the macro files and the Forth source in the block file.

2.1.1 Copyright of the ciforth's generated by this tool.

A ciforth generated by ci86.gnr is probably not a derived work (a thesis written in TeX is not a derived work from TeX). So D.F.W. separately claims copyright for the different versions of ciforth generated by her using this tool.

The following is present in all documentation of ciforth's:

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

In addition to the GPL the D.F.W. states the following:

> The GPL is interpreted in the sense that a system based on ciforth and intended to serve a particular purpose, that purpose not being a "general purpose Forth system", is fair use of the system, even if it could accomplish everything ciforth could, under the condition that the ciforth it is based on is available in accordance to the GPL rules, and this is made known to the user of the derived system. Consequently, for these systems the obligation to make the source available does not apply.

2.2 Legal matters

My extensions are GPL-ed or library GPL-ed. I transferred the copyright to the Dutch Forth Workshop, a foundation that supports Forth and defends the GPL. The original fig-Forth is public domain and is still available.

2.3 Rationale

What you find here is a Forth for the Intel 86. It is an ISO as of old. Complying in detail with ISO for the CORE wordset at least.

This has bee split off of a similar generic Forth, that is intended to be the last of the fig-Forth's. This generic system is no longer maintained, but the last fig-Forth is, in the sense that if there ever should be found a bug, it can be fixed. It also features the Fig glossary, which is available in electronic from for the first time in history. I shamelessly copied from it. Apart from being ISO, the Forth you have here is similar in many respects to fig-Forth. The motivation for having this type of Forth available follows from its characteristics. It is available as an assembler source, and it is an indirect threaded Forth. An assembler source has distinct advantages for getting started from nothing. An engineer might balk at the description of how to use a meta compiler, but feels at ease with a (much larger) assembler manual. (The popularity of eForth proves how popular assembler source is. I do not think that system has much merit, besides that.)

Although speed is currently in fashion, using subroutine threaded Forth's with optimizers, indirect threading is the preferred choice for some applications. I did this work, because I

needed it. I have also the firm belief that an optimizer on an indirect threaded system has more information to work with and can ultimately outperform any other system in speed.

2.4 History

From the introduction to the fig-Forth installation manual:

> The fig-Forth implementation project occurred because a key group of Forth fanciers wished to make this valuable tool available on a personal computing level. In June of 1978, we gathered a team of nine systems level programmers, each with a particular target computer. The charter of the group was to translate a common model of Forth into assembly language listings for each computer. It was agreed that the group's work would be distributed in the public domain by FIG.
>
> We intend that our primary recipients of the Implementation Project be computer users groups, libraries, and commercial vendors. We expect that each will further customize for particular computers and redistribute. No restrictions are placed on cost, but we expect faithfulness to the model. FIG does not intend to distribute machine readable versions, as that entails customization, revision, and customer support better reserved for commercial vendors.
>
> Of course, another broad group of recipients of the work is the community of personal computer users. We hope that our publications will aid in the use of Forth and increase the user expectation of the performance of high level computer languages.

2.4.1 Deviations of the FIG model

The first version of ciforth complied faithfully to the fig model, at least as faithfully as is customary. Now it is ISO, which means a lot of details are changed about how words work. In the following we will discuss not those details, but the changes made to the general build up of the Forth. (In fact some details are discussed. They are of little interest and they will move to an appendix in the next version.)

The rigid subdivision in 7 area's was never adhered to. In particular the boot up parameters are not up front as CP/M and MS-DOS require a 100H byte reserved area there. There is mention of '`(KEY)`' being "implementation dependent code" but these were not often present in fig implementations. This was based on the idea that there was some EPROM with console commands. This has been replaces by calls to an operating system, that do not comply with a simple function that could be called. Here the code definitions for `KEY` itself become implementation dependent code, but often it can written in high level.

All documentation is now accurate but only claims to describe ciforth. The `RUBOUT` key is a bona-fide `USER` variable and now has a name.

DR0 and DR1 are removed. There is only one consecutive mass storage area, be it a disk or a file. The assumption in using `OFFSET` was that you have two identical floppy drives and no hard disk. That is nowadays extremely unlikely. Instead I put `OFFSET` to good use to screen off a part of the floppy that must not be used (such as an MS-DOS directory or the hard disk part that contains the forth system.)

'`MOVE MON BLOCK-READ BLOCK-WRITE DLIST`' are not present. Altering OUT to influence formatting doesn't work here, nor on any fig-Forth I know off.

`+ORIGIN` now points to the first user variable instead to some not well defined start of the boot image. The layout has not changed, so the negative offsets can be used for the other data traditionally there. Their indices have not changed, but there is now a boot-up parameter for each and every user variable. A system with other boot parameters can now be generated in an even more portable fashion by using phrase like '`7 CELLS +ORIGIN`'.

Where possible installation dependent code is using a generic call to the operating system in particular MS-DOS BIOS or LINOS. This mostly results in the installation dependent code to be high level.

The false urban legend that one could `FORGET TASK` has been replaced by an accurate description of `TASK`. The following words have been documented for the first time: `FLUSH CURRENT 2DUP RP@ U..`

Some non-substantial deviation of the original FIG source have been made for good reasons.

The FIG philosophy is that sectors, blocks and screens must be compatible, but may be all different. The original 8086 FIG had one sector for a block. I changed that in having one block for a screen. This is a boon for those wanting to ISO-fy the sources.

The way I coded the character I/O points ahead to vectoring `TYPE` and `EXPECT` rather than `EMIT` and `KEY` . This way I can have the host system handle the rub out key.

I added generic words for accessing system resources `BIOS` , `BDOS` and `LINOS` . (See subsection The joy of genericity.)

Some real errors were fixed:

1. The redefine forthsample(NULL) bug is fixed. It is no longer possible to redefine this word, that handles the refill of the `TIB` , by typing a <ret> immediately after a defining word.
2. Forgetting part of a vocabulary, other than the `FORTH` vocabulary no longer crashes.
3. Loading a screen with characters having an 8th bit set, no longer crashes.

2.5 Evolution of ciforth

The first version of ciforth was in fact the fig-Forth for the 8086 that was put in the framework of this manual. By adding a 32 bits macro file, programming I/O for Linux, programming I/O with non-obsolete MS-DOS calls and a way to switch to protected mode, this fig-Forth came available in all ciforth configurations. The RCS version numbers of the generic file fig86.gnr are in the 2-branch and the latest version is available still. (The 1-branch was experimental). This version has however an manual not split between a generic and a user part. But the user part of the manual **is** generated from the generic source. This version 2 can be seen as a 32-bit figForth.

The third and fourth versions of ciforth (RCS branch 3 and 4) are generated according to this manual. As you see there is little pertinent information about these Forth's in this manual. All the information you need to use it is in the user manual, generated with that version. Branch 3 evolvestowards an ISO compatible system. Version 4 is a stable maintained ISO compatible system, a balance between technical criteria and compatibility issues.

2.6 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 file comprising it must be kept together and in particular the reference section with the World Wide Web sites.

This Forth builds on fig-Forth, for its source see the next section. The maintainer can be reached at forthmail(ciforth@spenarnc.xs4all.nl)

2.7 Acknowledgment

ciforth is based on the fig-Forth of Charlie Krajewski and Thomas Newman, Hayward, Ca. This fig-Forth (as are all fig-Forth's) is public domain. It is still available via taygeta. And of course kudos to FIG.

'`ftp://ftp.forth.org/pub/Forth/compilers/native/dos`'

This original version is public domain according to the following statement:

> All publications of the Forth Interest Group are public domain. They may be further reproduced and distributed by inclusion of this credit notice:
>
> This publication has been made available by the Forth Interest Group,
>
> P. O. Box 1105, San Carlos, Ca 94070

I also want to thank J. E. Smith, Philadelphia for another fig Intel86 implementation also still obtainable from '`http://www.simtel.net/pub/simtelnet/msdos/forth/fig86.zip`'

This is a fairly good documented FIG Forth for IBM PC, but its "Seattle Computer 8086 assembler" format makes it less practical.

3 Background.

If you are a Unix and a Forth guru, you can skip this chapter. If you think you are, you can read this chapter and discover you are not. This chapter is about pervading concepts and how tools are used, conceptually.

3.1 Orthogonality

The concept of orthogonality is central to this effort. Orthogonality means that different aspects of configuration (in this case) are made independent of each other. For example, ciforth can be bootable or started by MSDOS, it can be assembled by `nasm` or by `MASM.EXE` . These two choices can be made independently from each other, and every combination ought to work. Each choice is associated with file with macros for `m4` , so ideally if you need to modify how `nasm` assembly source is generated to you only need to change the file '`nasm.m4`'.

This is, of course, as far as it goes. Try as you may to separate all information about header layout in the '`header.m4`' configuration file, a change to the order of the fields in a header will certainly have it impact at certain places in the source.

3.2 Metacompilation is outdated

Meta compilation, the generation of a new version of a Forth system by "similar tools as compilation", was invented for the cassette based computer system of the late seventies. There may be a motivation for using metacompilation to generate a similar forth for a **different** processor or system. This would properly be called cross-compilation, by the way. In a half-decent (or better) disk operating system like MSDOS the use of meta-compilation is a mistake at the management level.

We want our Forth to be able to generate standalone programs anyway. (a *turnkey* facility.) So what do we need

1. A facility to save a running system with all what is loaded on it, in the configuration it currently has.
2. A facility to remove parts of a running system, that are not needed for an application after it has been build. (E.g. the assembler.)
3. A facility to optimize some parts of a system. (Then remove the, possibly large, optimizer.).

If you have the first facility, you can build a powerful Forth from a small kernel and regular source code. If you have all of them, you can build a truly, optimal Forth from a small kernel. You need not "a similar tool as compilation", you just need "compilation" .

The `SAVE-SYSTEM` facility of course requires in depth knowledge of the operating system. This doesn't mean it is cumbersome or difficult. Under Linux we need

```
HEX
\ The magic number marking the start of an ELF header
 CREATE MAGIC 7F C, &E C, &L C, &F C,
\ Return the START of the ‘‘ELF’’ header.
 : SM BM BEGIN DUP  MAGIC  <> WHILE 1 CELLS - REPEAT ;
\ Return the VALUE of ‘‘HERE’’ when this forth started.
 : HERE-AT-STARTUP  ’ DP >DFA  +ORIGIN  ;
\ Save the system in a file with NAME .
 : SAVE-SYSTEM
\ Increment the file and dictionary sizes
  HERE HERE-AT-STARTUP - DUP SM 20 + +!      SM 44 + +!
   U0    0 +ORIGIN   40 CELLS  MOVE \ Save user variables
\ Now write it. Consume NAME here.
   SM    HERE OVER -   2SWAP   PUT-FILE ;  DECIMAL
```

3.3 How m4 is used.

The Unix macroprocessor `m4` is very powerful indeed. Testimony is that the description of its usage in here is longer that its man-pages. You know `m4` is a text substitution tool. A macro is like a function. In the macro call the text is replaced by the text present in the function. Within the text the placeholders for the parameters are replaced by the actual parameters. In `m4` the placeholders are ‘`$1`’ ... ‘`$9`’. Parameters can be passed, and any (even multiline) text can be given as a parameter, provided it is quoted. We will use ‘`{`’ and ‘`}`’ (braces) throughout. This is convenient, because they are not used in a Basic Forth system and they are special anyway (e.g. for TeX). The use of quotation is very critical at times, and the find points are not covered in the following.

3.3.1 Customization

Simple customization can be done by `m4` as follows:
‘`define({version},2.149)`’
Within the text treated the version number is substituted.

3.3.2 Selection

Selection, often one of alternatives, is in general done as follows
‘`_BITS16_(32)_BITS32_(64)_BITS64_(128)`’ ,
which gives, of course, the size of a double number.
This is accomplished by
‘`define({_BITSxx_ },{$1 })`’
for the actual bitsize and
‘`define({_BITSxx_ },)`’
for others.
Selections can be nested within other `m4` macro construct. As in

```
_VERBOSE_{(_BITS64_({The possibility to cycle through all (64-bit)
numbers by forthsamp({0 0 DO ... LOOP}) is very useful indeed.})})
```

Here you see at work, apart from '`_BITS64_`' , the macro '`_VERBOSE_`' that allows (if turned on) verbosity that can help understanding but is not always appreciated. You also see '`forthsamp`' that is in fact a markup to indicate we have a piece of Forth code there.

Selections can be used to throw out a block of word definitions and their documentation as a whole. For example words accessing I/O ports are not available in a Linux Forth, as they would only lead to privilege violations.

The braces are essential here. Without it the introduction of a comma somewhere in the text results in `m4` interpreting the remainder as a second parameter, which it will ignore.

3.3.3 A postponed markup language.

Just say '`forthcode({+LOOP })`' to indicate that you want formatting as for "code" words. Later you can decide to use
'`define({forthcode },{@code{$1}})`'
for '`texinfo`' or
'`define({forthcode },{<B>$1</B> })`'
for '`html`' .

3.3.4 Defining structures

Some macro calls must be considered to define a structure, in particular '`worddoc`' . Suppose we have a list of structures, meaning that the first person is a child of the second and third person:

parents({Alice},{Mary},{John})

parents({Fred},{Mary},{Henry})

parents({Aayilah},{Sjantil},{Bodaji})

...

With '`define({parents},{$2})`' we get a list of (you guessed) the mothers.

The usage of '`divert()`' can best be explained with an example in this context.

```
define({parents},
{{divert(3)dnl}
$2
{divert(6)dnl}
$3
})
```

will give out the mothers on channel 3 and fathers on channel 6. The output will be concatenated, but all mothers and all fathers stay together. For '`dnl`' see the `m4` man-page.

3.3.5 Defining lists

By using an extra pair of braces you can have a list in `m4` . So '`{{A},{B},{C},{D}}`' is a single parameter to a macro and can be passed to other macro's as a whole. The outer braces are removed and without special measures (reinstalling extra braces again) the macro called **sees** the comma's and concludes there are four parameters. This is put to good use in the "See also" and "Test" fields of the '`worddoc`' structure. These fields may have zero or more parts.

The "Test" field contain the tests in the odd fields, and the expected outcome in the following even fields.

3.3.6 Defining aliases

Sometimes you need aliases, i.e. other names for macro's, Although it doesn't properly belong here as a technique, I want to mention it, because the amount of brackets is hard to sort out. An alias is useful in a transition period, where you want to rename something, but where you want to be able to do that gradually on a file by file basis.

'`define({_OLDNAME_},{_NEWNAME_({$1},{$2})})`'

After '`_OLDNAME_`' is phased out everywhere this definition can be deleted. Note that for this to work all parameters applicable to '`_NEWNAME_`' must be taken into account, the two shown here are just an example.

3.3.7 Impress the crowd

By using macro's to define other macro's, then pass the result through `m4` another time, severe stress can be laid upon the intelligence of the everyday person. The very inconvenient way nodes must be linked in texinfo even forced me to define part of the macro in one macro and the remainder in another.

3.4 How `ssort` is used

The sorting tool `ssort` can order multiple field records, with different sorting criteria for each field. The fields can be defined by regular expressions, such that the '`worddoc`' structures can be sorted by name, or by wordset then by name, or in about any way you want. Because such a tool didn't exist, I had to write it.

3.4.1 Analyzing '`worddoc`'

`ssort` captures the structure of a '`worddoc`' as follows:

'`^worddoc({@},{@}.*\n$worddoc`'

The part between '`^`' and '`$`' matches the record. The part after the last '`$`' is for synchronization, to make sure the record doesn't end early. This would result in an error "not according to structure": the next line doesn't start with "worddoc" and so it just doesn't match the record description. The '`$`' is merely a separation, (newlines are indicated by '`\n`'). The '`.*`' matches anything, including new lines. But it isn't greedy as in ordinary regular expressions, because not being stopped by '`\n`' , it would match the whole file. Here it tries to match as little as possible. '`@}`' is shorthand for '`[^}]*}$`' so a "sequence of anything except right braces followed by a right brace". It also contains the '`$`' to mark the end of a field.

3.4.2 Sorting fields

Once we know what the fields are, '`-M 1S2S` ' sorts on the first field and within that field on the second. We just use the ordinary ASCII collating sort, indicated by '`S`' .

4 Structures and processes

4.1 The generic source file

The generic source file '`ci86.gnr`' mostly consists of Intel assembly code, with which, I assume, you are familiar. All macro's in the following are '`m4`' macro's. Words are divided in small (<20) groups of cooperating words, the *wordset* . See also "thinking Forth".

The things that differ among assemblers, are taken care of by simple macro's, e.g. '`_COMMENT`' starts a comment. Most of the time they don't have parameters.

The selection of parts that go or don't go into a particular configuration is done by multiline macro's, generally with a call on a separate line. Such as:

```
_HIGH_BUF_({
BUF1    EQU     EM-(KBBUF+2*2)*NBUF      ;{ FIRST DISK BUFFER}
STRUSA  EQU     BUF1-US          ;{ User area}
});_END_({ _HIGH_BUF_})
```

Note how comments are protected from macro expansion by quotation. The '`_END_`' is an adornment. It expands to nothing. So it doesn't show up in the output, but it helps to keep the generic source organized.

The '`worddoc`' macro defines a structure with additional information of a word. Generally it is placed in front of the word. The same word can be found several times in the input file, but only one is selected in a particular configuration. The same goes for the corresponding '`worddoc`' .

Its fields are:

1. Wordset name.
2. Word name.
3. Pronunciation. This is a pure textual and pronounceable identification of the word. It is also used in '`texinfo`' that doesn't handle special characters well.
4. Stack effect. The stack effect obeys all the conventions put forth in the user manual.
5. Properties. Properties are i.a. immediate and such, and the standards with which this word complies. Again this is described in the user manual.
6. Description.
7. References. This is a list of names of other Forth words, that can be studied to better understand this one.
8. Tests. This is a list. The first and all other odd members is a test, code that can be passed to Forth. The second and all other even members is the expected outcome of the preceding test.

'`worddoc`' are such that a structure starts with '`worddoc( `' and end with a '`})`' at the end of a line. This means that a worddoc can be simply skipped if it occurs in Forth code, by defining a word `worddoc(` that reads and ignores source up to the end sentinel.

The '`worddocchapter`' macro defines a wordset. It has the same fields as a '`worddoc`' macro, but most are left empty. It is primarily used for its "description" field, that is used as an overview description for the wordset in glossaries. These macro's can be put anywhere, but take care to exclude macro's for wordsets that are not present.

4.2 The process

The ultimate information about how a ciforth is generated are the makefile's : '`Makefile`' and '`test.mak`' .

The process of generating a program proceeds along the following steps:

1. Generated the assembler source from the generic source via a configuration file. The file suffix indicates which assembler to use.
2. Generate an object file.
3. Link the object file.

Once you have an assembler file, you can do what you want with it. Proceeding from an assembler source file to a binary is in general straightforward.

The process of generating program's documentation (TeX and info) proceeds along the following steps:

1. Generated the raw glossary documentation from the generic source via a configuration file. The file suffix is '`.rawdoc`' .
2. Sort the '`.rawdoc`' file, such that words of a wordset appear together, and are preceded by a wordset documentation. The file suffix is '`.mig`' .
3. Generate the glossary documentation from the '`.mig`' by expanding the '`worddoc`' 's into glossary entries by '`gloss.m4`' or '`glosshtml.m4`' . This, for a second time (!), takes into account the configuration file, to generate exactly fitting information.
4. Expand the "postponed markup's " in '`ciforth.mi`' by the macro's from '`manual.m4`' to generate the texinfo commands. This file include all the other '`.mi`' files with postponed markup's.

The process for generating html has the postponed markups and the expansion into glossary entries in the same file '`glosshtml.m4`'. Only the documentation of the glossary enters into html , and the '`ciforth.html`' is generated from the intermediate file '`.mig`'.

Generating documents is made more complicated by the requirements for special tables. For '`html`' we want an extra alphabetic list of all the words where we click on to get at the glossary entry immediately.
In '`texinfo`' we need to build complicated menu structures, that refer back and forth.
This is done by separate passes over the '`.rawdoc`' or '`.mig`' files, with other macro's.

5 off we go

5.1 Introduction

What you find here is a Forth for the Intel 86. Not much more can be said for such a highly configurable system. But in this section we will try to summarize the common characteristics.

It borrows some philosophy from the old fig-Forth. It is in fact based on it, and its documentation in first draft copied from it. The Forth's are build from an assembler source, and it is (in general) an indirect threaded Forth. The motivation for having this type of Forth available follows from its characteristics. An assembler source has distinct advantages for getting started from nothing. An engineer might balk at the description of how to use a meta compiler, but feels at ease with a (much larger) assembler manual. Although speed is currently in fashion, using subroutine threaded Forth's with optimizers, indirect threading is the preferred choice for some applications. Furthermore the current trend of subroutine threaded Forth's may very well be unsuitable for 64-bits processors like the Alpha. I did this work, because I needed it for my thesis on computer intelligence.

5.1.1 32 bits

It is unusual for a forth to be configurable as 16 or 32 bit. It turned out that the addition of `CELL+` goes a long way toward allowing utilities like a decompiler to be 16/32 bit clean. In the documentation mostly reference to cells can be made. But the macro's '`_BITS_`' '`_BIT16_`' and '`_BIT32_`' can be used to signify the actual number of bits and parts to refer to 16 and 32 bits only respectively.

5.1.2 System requirements

This generic version -if suitably built- runs on industry standard hardware ("PC's") : standalone, under Linux and under MSDOS/MSWINDOWS. To build, you need a version of `nasm` , `TASM.EXE` or `MASM.EXE` on your system. I recommend `nasm` , it is an open source assembler and available on different platforms, at least MSDOS and Unix. It solves a lot of the design errors I find in the Intel ways of `MASM.EXE` . It is easier to use than the GNU `as` because it adheres more to the Intel syntax. It generates a binary without a linker. On the opposite side, e.g. Borland's `TASM.EXE` you can buy nowadays only as part of a giant C++ package. If you want to use the generic possibilities you will need a Unix system with all of its tools. I have successfully used GNU-Linux (RedHat andS Suse) to do the makes and version control on that. If you want your bootable floppies made from Linux to be MSDOS-compatible you need mtools.

5.1.3 Assembler sources

The following two assembly sources generated are supplied as a service. These are in fact just examples. You can generate different ones (see next section.) The file '`alone.asm`' can be assembled using `nasm` . It includes a boot sector such that it can boot from a standard floppy on a industry standard Intel PC. If you have the mtools set (most Linux'es have it) the Makefile shows you how to make the floppy. On MSDOS you can use `DEBUG.EXE` . If you run on Linux with '`mtools`' , '`make boot`' will do it. The resulting floppy will even be recognized by MSDOS, such that you can copy block sources to it. '`make moreboot`' will do this from Linux, then you will have '`forth.lab`' available. '`make allboot`' will do it all, but it needs a working forth on Linux for doing some calculations. Otherwise on MSDOS (I recommended version 3.3, the most stable MSDOS ever) adapt the example '`genboot.asm`' . The file '`msdos.msm`' can be assembled using `TASM.EXE` and `MASM.EXE`. The resulting Forth executable can be run off hard disk and respects the file system on it. It uses the file '`forth.lab`' .

5.1.4 A generic Forth

As was mentioned before, ciforth has one single source file: the generic '`ci86.gnr`' . All advantages of assembler source would be gone, if an engineer were confronted with conditional compilation and lots of code for other systems he doesn't want to learn or assemblers he doesn't want to use. So we proceed in two steps. First a clean assembler source is generated from the generic Forth using configuration files. Then the assembler source is processed in one of a number of ways, each way familiar to one brand of engineers.

You can customize at a number of levels.

1. Configuration files have extension '`.cfg`' , these are files with `m4` commands. They are intended to use at the highest and easiest level of configuration and contain their own simple usage instructions.
2. `m4` files have extension '`.m4`', and control one aspect of genericity, such as the header layout or the protection mode. You definitely need to know `m4` to use these.
3. Assembler files can be customized in the traditional way by adopting constants, or commenting in source lines. The assembler files are distinct from the one generic source. No `m4`, you need only cope with the directives of your assembler, and will not see any code applicable to other operating systems or I/O systems. (It is not commented out, it is just not there.)
4. You can adapt the generic source. This is difficult, but gratifying. If you manage to ISO-fy it, the result is a lot of ISO systems, not just one.

5.1.5 Level 1 customization.

This is assuming you run on Unix.

By specifying what you want in a configuration file you can generate a host of assembler listings. This is as simple as replacing "$1" with "$2" in configuration files. See the examples '`msdos.cfg`' and '`alone.cfg`' and the Makefile. You can find out what the options are by inspecting '`prelude.m4`' .

There is a division of labor between your configuration file and the '`prelude.m4`' and '`postlude.m4`' files. '`prelude.m4`' sets all variables to defaults, for sets of alternatives this is NO, waiting to be overwritten. For the other options it is the most sensible one. You must include '`prelude.m4`' first in your configuration file. Then you specify your configuration and include '`postlude.m4`'. '`postlude.m4`' will correct the options to the most, or the only, sensible ones for that configuration. It will reject some of the configurations that will not assemble, or lead to programs that do not work. Then you can after including '`postlude.m4`' overwrite some of the sensible defaults. So you can force the generation of source that is rejected by the assembler anyway. An example is the default stack size of 64K for 32 bit programs. Sensible as it is, you may want to have a 32 bit Forth that runs in 64K. You will overwrite that stack size. Be careful.

With respect to the assembler you can choose between `nasm` and `MASM.EXE` , with file extension '`.asm`' and '`.msm`' respectively. The '`.msm`' are acceptable by NASM.EXE too. You can generate an equivalent '`.s`' file, but this is experimental and doesn't lead to a working forth.

I have reported problems in generating a forth using the Intel 86, GNU tools. Whether or not the problems could have been overcome at the time, at least from kernels 2.4.x on, GNU tools should be able to generate a Forth.

With respect to the I/O (words like `EXPECT` `R\W`) you can choose between three on MSDOS. (`R\W` is what was named `R/W` in figForth, but that name is reserved by ISO now.) You can use dos '`_CLASSIC_`' in the classic way as with the original. This means that the floppy is used directly without regard for directory structures. This uses calls that are declared obsolete. You can use dos in a modern way. '`_MODERN_`'. This allocates block in the file with name '`forth.lab`'

. This name is available in the string BLOCK-FILE for you to change, also at run time. No (as of 2000) obsolete MSDOS calls are used (Checked against MS-DOS programmers reference "covers through version 6" ISBN 1-55615-546-8) You can use the BIOS '_USEBIOS_' No MSDOS interrupts are required.

With respect to I/O on Linux you can choose between c-based and native. The c-based version may be portable to other I86 unices. The native version of course not. All Linux versions have their blocks in a file. (Accessing a floppy in the classic way is perfectly possible – and implementing it would be a perfectly pointless exercise.)

With respect to the hosting you can choose between '_HOSTED_' ('_HOSTED_LINUX_' or '_HOSTED_MSDOS_') and '_BOOTED_' . ('_BOOTDF_' or '_BOOTHD_'). A hosted version relies on MSDOS or Linux to get the program started. (It may or may not use MSDOS for I/O, once started.). A '_BOOTED_' version contains a boot sector, such that you you can make a standalone version that boots from floppy or hard disk. A '_BOOTED_' version may very well be startable from plain DOS and its files visible from DOS.

Of course a 'BOOTED_' version that tries to use MSDOS I/O (or Linux) crashes immediately, so not all versions are useful.

You have a choice between 16 or 32 protected mode and real mode. Of course on Linux real mode is not an option, (but you could run the MSDOS emulator). Protected mode Forth's on MSDOS cannot be started from virtual real mode, e.g. they will not run in a "DOS box" in Windows.

If you manage to specify conflicting options the preprocessor (m4) breaks off and you can find the exit code in 'postlude.m4' . Than you can reason back why this is a conflict. For example error 1000 indicates floppy and hard disk i/o at the same time. From 'postlude.m4' you see that '_RWFD_' and '_RWHD_' are on at the same time. '_RWHD_' is turned on because you wanted to boot from hard disk or you specified it yourself in the first place. Etc.

'postlude.m4' does you another favor. It derives logical consequences, such as once you decide for a '_REAL_' mode Forth, it must be '_BITS16_' and you need not specify that_VERBOSE_((And yes, we could add a real mode 32 bits Forth, any volunteers?)). In particular '_LINUX_N_' or '_LINUX_C_' define a whole configuration.

5.1.6 Level 2 customization.

You are on your own here.

5.1.7 Level 3 customization.

So you have this assembler file, and it looks like what you want to have, but not quite. And of course it doesn't work.

5.1.7.1 My rants

The usual customization in assembler files is possible. If you use other than 3" floppy disks you have to specify the disk parameters. Parameters for a 5" HD floppy are present and can be commented in. If you do not need a DOS-compatible floppy, you can put the image immediately after the boot sector. A bootable hard disk version always works like that. You can change the default name of the BLOCK-FILE at run time. If you want to change the header layout, you will find that the way headers are done via MACRO's make it more pleasant to use the generic listing. If you may want you can use this as a starting point for generating a whole other Forth (like me). If you want to boot into your 20 Gbyte disk (like me), you probably have a version 3.0 super modern LBA BIOS. There is no file system, just 20,000,000 blocks (and yes a 16 bit system would be inconvenient). If you want to use an older system you must experiment by using the BIOS word. (You need not resort to assembler for experimenting.) Then you can adapt your assembler listing.

5.1.7.2 FIG's rants

You may want to use the assembly code of this ciforth to base a new Forth on. If this adversely affects the documentation I urge you not to do that but to use the generic system.

The following words are traditionally the only portion that need change between different installations of the same computer CPU. They cannot come close to the capabilities of the generic system, and should be used for minor modifications only.

There are five words that need adaptation:

`KEY` Push the next ASCII value (7 bits) from the terminal keystroke to the computation stack and execute NEXT. High 9 bits are zero. Do not echo this character, especially a control character.

`EMIT` Pop the computation stack (16 bit value). Display the low 7 bits on the terminal device, then execute NEXT. Control characters have their natural functions.

`?TERMINAL`
For terminals with a break key, wait till released and push to the computation stack 1 if it was found depressed; otherwise 0. Execute NEXT. If no break key is available, sense any key depression as a break (sense but don't wait for a key). If both the above are unavailable, simply push 0 and execute NEXT.

`CR` Execute a terminal carriage return and line feed. Execute NEXT.

`R\W` This colon-definition is the standard linkage to your disc. It requests the read or write of a disc block, be it raw disk or allocated in a file.

On primitive systems these may be jumps to ROM-code. But generally on i86 facilities like this are available using *INT* 's a kind of traps. These observe operating system protocols and are available as high level forth code.

5.1.7.3 FIG's rants : Ram disc simulation

If disc is not available, a simulation of `BLOCK` and `BUFFER` may be made in RAM. The following definitions setup high memory as mass storage. Referenced "screens" are then brought to the "disc buffer" area. This is a good method to test the start-up program even if disc may be available.

```
HEX
4000 CONSTANT LO ( START OF BUFFER AREA )
6800 CONSTANT HI ( 10 SCREEN EQUIVALENT )
: R\W >R ( save boolean )
    B/BUF * LO + DUP
    HI > 6 ?ERROR ( range check )
    R> IF ( read ) SWAP ENDIF
    B/BUF CMOVE ;
```

Insert the code field address of `R\W` into `BLOCK` and `BUFFER`

and proceed as if testing disc. This `R\W` simulates screens 0 thru 9, in the memory area 04000H thru 067FFH.

5.1.7.4 FIG's rants : Debugging an assembled system.

Let us assume we have an system based on an assembler listing and we want to debug it.

Here are the sequential steps:

1. Familiarize yourself with the model written in Forth, the glossary, and specific assembly listings.
2. Edit the assembly listings into your system. Set the fifth boot-up parameters (WARNING) to 0 (warning messages are shown as simple numbers).
3. Alter the terminal support code (`KEY` , `EMIT` , etc,) to match your system. Observe register protocol specific to your implementation!
4. Place a break in your debugger at the end of NEXT, just before indirectly jumping via register W to execution. '`W`' is the Forth name for the register holding a code field address. In ciforth this is the Intel 86 register AX. If your NEXT is inline code, for the moment replace it by a jump. Mostly this can be done by inactivating the macro '`_NEXT32_`' through removing the line '`_BITS32_({define({_NEXT},{_NEXT32})})`' from '`postlude.m4`'.
5. Enter the cold start at the origin. Upon the break, check that the interpretive pointer Y points within `ABORT` and W points to `SP!` . But in the source the symbolic names is used. `COLD` being a colon-definition, the Y has been initialized on the way to NEXT and your testing will begin in `COLD` . The purpose of `COLD` is to initialize Y, U, S, UP, and some user variables from the start-up parameters at the origin.
6. Continue execution one word at a time. Clever individuals could write a simple trace routine to print Y, X, U, S and the top of the stacks. Run in this single step mode until the greeting message is printed. Note that the interpretation is several hundred cycles to this stage!
7. Execution errors may be localized by observing the above pointers when a crash occurs.
8. After the word `QUIT` is executed (incrementally), and you can input a `"return"` key and get "OK" printed, remove the break. You may have some remaining errors, but a reset and examination of the above registers will again localize problems.
9. When the system is interpreting from the keyboard, execute `EMPTY-BUFFERS` to clear the disc buffer area.
10. If your disc driver differs from the assembly version, you must create your own `R\W` . You may test the disc access by typing: '`0 BLOCK 64 TYPE`' This should bring block zero from the disc to a buffer and type the first 64 characters. If `BLOCK` (and `R\W`) doesn't function–happy hunting!

5.1.8 Level 4 customization.

Contact me if you want to contribute to the wider usability of this package.

5.1.9 Programs

In the file '`forth.lab`' is available a screen editor, assembler, decompiler and tools like `DUMP`. Beware! Some of the tools handle hards disks. There are example programs and benchmarks. Everything under screen 100 you will find more or less working, but maybe not on your system. Everything loaded from 8 is used by me on a regular basis and is 16/32 bits clean. Beware! The full screen editor does not work under Linux (protection). The system doesn't load it under Linux with '`8 LOAD`' . The program `wc` is an example of how to use `lina` as a scripting language.

5.1.10 The joy of genericity

I added generic words `BIOS` , `BDOS` and `LINOS` . These allow to have high level words to handle about all "BIOS" and interrupt 21 calls. Linux is better. `LINOS` handles all Linux system calls.

Genericity is accomplished by the Unix tools `m4` . I use GNU `m4`. This is a weird tool but powerful. Forthers probably like it. Some implementation details are hidden in the file '`header.m4`' . In particular the way headers are built. I maybe want to get rid of the `WIDTH` and `TRAVERSE` peculiarities and you may want to have the headers aligned at word bounds. This is easily done by changes to '`header.m4`' . This kind of possibilities were in fact the motivation for this undertaking.

5.1.11 Web sites and availability.

A newer or improved version may be gotten from '`http://home.hccnet.nl/a.w.m.van.der.horst/cifo`
Nasm is found at

ftp://ftp.us.kernel.org/pub/software/devel/nasm/source/

'`http://www.cryogen.com/Nasm/`'

The FIG source this is based on is at

MASM.EXE is available from IBM, at least the version 1.0 I used. Microsoft seems to distribute later versions.

The original fig documentation is obtainable via '`http://home.hccnet.nl/a.w.m.van.der.horst/figfo`
This include the pictures.

5.1.12 Linux application notes ciforthc version

The Linux forth called ciforthc has its i/o based on c. This may seem more portable but it isn't. Where c is very portable on Linux, the way assembler is linked with forth is not documented (as far as I can tell, in my version. Linux improves overnight, so this may no longer be true.) `KEY?` is implemented using a '`select`' system call. The `EXPECT` has not the `"` return if maximum reached property`"`, so it is not strictly conforming. This can be done at the expense of handling each character separately. (Use `KEY` to implement `EXPECT` as in the CLASSIC I/O model). This results in loosing interruptability. Moreover Linux knows better what the `RUBOUT` key should be, although for your convenience it is already placed in a user variable and can be easily changed. The c-approach allows signals to be handled in a familiar way. By using quit, a loop can be interrupted. So `^\` results in a warm start. A segmentation fault also results in a warm start. `^C` immediately leaves. `^S`/`^Q` can be used to hold up output and are not interpreted as a break in e.g. WORDS.

5.1.13 Linux application notes lina version

The lina version is based on a single assembler source, built without trickery and binary-portable across Linux Intel (all systems were it has been tried work : 1.2.13 .. 2.4.20). No run time c-libraries, no compile time c-libraries, no libraries at all. It is built directly on the solid rock of the system calls by ignoring a taboo c-programmers suffer from.

```
nasm -felf lina.asm
ld lina.o -s -o lina
strip lina
```

It is about 20k and the dictionary space is set at 64 Mbyte.

Blocks are allocated in a file called '`forth.lab`' This name can be changed in listing and also during run time. '`forth.lab`' can be changed into an editable file and back by '`cat forth.lab | fromblocks > blocks.frt cat blocks.frt | toblocks > forth.lab`'

The user variable '`EM`' still is the end of the memory. The '`M4_EM`' in the configuration files is such that it designates the relative size, from the relocatable start. Consequently it is not the same as the user variable. (The relocatable start is some 128 Mbyte into the memory space.).

5.1.14 Bugs

See the separate test report for an indication of which and how far versions have been tested.

1. Linux version. Once you have used a SIGQUIT to interrupt a loop, `BYE` no longer works. You can exit the program by `"0 0 0 1 LINOS"`, which is exit(0) in c-parlance or by pressing `^C`, or by killing it from some other terminal, or by just closing the window. You will not encounter this bug in version 2.148, because that version crashes immediately, due to build problems.
2. `OUT` may not be observed in all I/O models. Needs examination.
3. More a misfeature. The negative error numbers of Linux system calls can be handled by negative offset's from screen 4.

6 Now: release it!

To release the software use the command

‘`make release VERSION=NNN`’

where ‘`NNN`’ is as appropriate: the RCS version number (for ciforth 3d###) for a beta version and the minor release numbers for an official release (ciforth has major release 4, because fig 2 is taken, my latest fig is major release 3, so there.) Official releases have the `d` replaced with a `a` And RCS version numbers always run in the hundreds, official minor releases are one digit. And during started up it tells "ciforth beta ", if it is not official.

There are basically two zip files with the same contents, and a formidable content at that. They are ‘`ci86gNNNN`’ and ‘`fgNNNN.zip.`’ The ‘`fg`’ version features 8+3 file names, appropriate for ISO9660 or FAT (MSDOS) file systems.

7 Loose ends

7.1 list of safe customization in the assembler.

RTS TIB / Return stack area

The block stuff is easily the most crappy part of this system. What little advantage it had, is no longer applicable. Interestingly the fig block system is copied verbatim into ANS. 32 bits systems can enlarge the dictionary by redefining the stacks and the terminal input buffer positions. The block buffers are in the way. So I put them in the dictionary space. In fact the block buffers have become the data field of FIRST.

My changes :

1. There is a very simple `INCLUDED` to compile files. This goes a long way towards eliminating the blocks. They become a convenience rather than a necessity.
2. Blocks are always `UPDATE` 'd immediately. This makes words like `FLUSH` trivial.
3. `BLK` and its ilk are defined by second guessing. They do not dictate what is going on. This is contrary to ISO, but makes the basic system so much cleaner. Blocks that load other blocks are `LOCK` ed during that time, so that `WORD` need not constantly look whether the rug is pulled out from under it.
4.

The backspace character is also in the boot-up origin parameters. It is universally expected that `"rubout"` is the backspace.

Despite all the talk about user variables, I am not aware that a multi-user fig forth existed, ever. The above block policy is not compatible with multi-user (but the actual code is not worse than it ever was, and you can rebuilt it with 1000 block buffers if you like.)

CUSTOMIZING

The name USER reflects that more than one user could use the dictionary and users could share the background storage, provided certain precautions are taken. About this you can forget. What you can do is, store back you user variables in the boot-up parameters as follows '`USVA ' USVA +ORIGIN !`' where '`USVA`' is the user variable which value you want to keep.

Underneath the I/O model has improved. TYPE and EXPECT are drawn into the I/O model dependent part. If at all possible CR and EMIT used TYPE such that TYPE becomes a natural vectoring point.

Rubout is best left to the EXPECT code. Remember, a Linux itself knows the rubout key for any of its 500+ known terminal types and isn't it nice in MSDOS that F3 gets the previous command back for you? If EXPECT builds up a line from separate key strokes, and if you ever want to change the rubout key, just change the RUBOUT user variable.

7.2 MULTI-USER

The name `USER` reflects that more than one user could use the dictionary and users could share the background storage, provided certain precautions are taken. These precautions are

1. Variables that can be different for different users, must be defined as an offset to an area, that is different for each user: the *user area* .
2. Provisions that maintains the integrity of the dictionary. Different scratch pads for each user.
3. Different stacks, user area's and terminal input buffer's for each user.
4. A means to switch applications.

Almost nothing from this is realized in the fig-Forth model. In fact only 1, but there is the pointer to that area is in the bootup parameters, a design error.

Still `USER` variables seem to serve a useful purpose. They are initialized during start up, by changing it one could modify the system. This however is only seemingly. Because the initialized memory is saved anyway, the initial values would be stored even if they were ordinary variables. So we are left with the disadvantage that we have to store them back before saving the system. Remains the advantage that they can be restored by typing `COLD` . This too is hardly an advantage because with the fast mass storage you would rather type `BYE` and '`forth`' as a much safer way to restart your Forth. Furthermore `COLD` performs a buggy `FORGET` , it cannot reinitalise the dictionary for a system that has active glossaries, that remain after a boot.

Program Index

This index lists programs words.

Forth Word Index

This index lists forth words.

Concept Index

This index lists concepts.

Short Contents

Table of Contents